# High Performance Linear Algebra

Lecture 1: Introduction and Overview

Ph.D. program in High Performance Scientific Computing

**Fabio Durastante** **Pasqua D'Ambra** **Salvatore Filippone**

November 10, 2025 — 14:00 to 16:00

Dipartimento
di Matematica
Università di Pisa

# Table of Contents

1 Introduction


- Introduction
  - Why high-performance Linear Algebra?
  - A gallery of problems
- What does it mean large-scale?
- Where do we solve such *large* problems?
  - The TOP500 list
  - The EuroHPC Joint Undertaking: www.eurohpc-ju.europa.eu
- What tools are we going to use?
  - (Modern) Fortran
  - Git
    - An example of Git usage

# About this course: general information



First some *bureaucratic* information about the course:

- Course webpage: fdurastante.github.io/courses/hpla2025.html
- Lecture slides: fdurastante.github.io/courses/hpla2025.html#lectures

The **exam** will consist in a **project work** to be presented at the end of the course. This will involve the implementation and performance analysis of some linear algebra algorithms, or the performance analysis of existing libraries, possibly in relation to a specific application. The choice of the project topic will also depend on your Ph.D. research topic, so to make it more interesting and useful for you.

# What is Linear Algebra?

1 Introduction

**Linear Algebra** is a branch of mathematics concerned with:

- **Vector spaces** and linear transformations
- Systems of **linear equations**, **matrices**, **vectors**
- Key concepts: determinants, eigenvalues, eigenvectors, singular values

**Applications:** computer graphics, machine learning, optimization, physics

**Numerical Linear Algebra** focuses on:

- Solving LA problems using numerical methods on computers
- Development of **efficient**, **stable**, and **accurate** algorithms
- Essential for *large-scale* problems where exact solutions are impractical

# Problem 1: Linear Systems

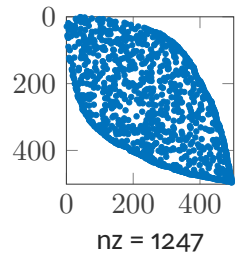
## 1 Introduction

Consider the Poisson equation (PDE):

$$-\Delta u = f \quad \text{in } \Omega, \qquad u = 0 \quad \text{on } \partial\Omega$$

**Discretization approach:**

- Divide domain into grid: $N = n_1 \times n_2 \times n_3$ points
- Use finite difference approximation for derivatives
- Results in sparse linear system: $A\mathbf{u} = \mathbf{f}$
- $A \in \mathbb{R}^{N \times N}$ is **sparse**
- Most elements are zero
- $N$ is typically very large




Sparse matrix pattern

# Problem 2: Eigenvalue Problems

## 1 Introduction

Find scalar $\lambda$ and vector $\mathbf{v}$ such that:

$$P\mathbf{v} = \lambda\mathbf{v}$$

**Example: Markov Chains**

- Transition matrix $P \in \mathbb{R}^{N\times N}$ ($P_{i,j} \geq 0$, rows sum to 1)
- Evolution: $\mathbf{p}_{\ell+1} = P\mathbf{p}_\ell$
- Stationary distribution $\boldsymbol{\pi}$ satisfies:

$$\boldsymbol{\pi}^\top = \boldsymbol{\pi}^\top P, \quad \boldsymbol{\pi}^\top \mathbf{1} = 1$$

- Finding $\boldsymbol{\pi}$ is an eigenvalue problem for large $N$

# Problem 3: Matrix Equations

## 1 Introduction

**Sylvester equation:** $AX + XB = C$

**Application: Model Reduction in Control Theory**

LTI dynamical system:

$$\dot{\mathbf{x}}(t) = A\mathbf{x}(t) + B\mathbf{u}(t), \quad \mathbf{y}(t) = C\mathbf{x}(t)$$

**Balanced truncation approach:**

1. Compute Gramians via Lyapunov equations:

$$AP + PA^\top + BB^\top = 0 \quad \text{(controllability)}$$
$$A^\top Q + QA + C^\top C = 0 \quad \text{(observability)}$$

2. Solve Sylvester equation: $AT + TS = B$
3. Efficient algorithms needed for large dimensions

# Problem 4: Machine Learning

## 1 Introduction

**Linear Regression:**

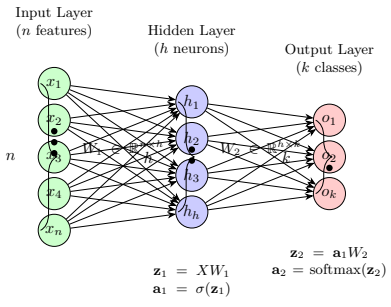
- Data: $X \in \mathbb{R}^{m \times n}$, targets: $\mathbf{y} \in \mathbb{R}^m$
- Find coefficients: $\min_{\boldsymbol{\beta}} \|X\boldsymbol{\beta} - \mathbf{y}\|_2^2$

**Neural Networks:**

- Weights as matrices: $W_1 \in \mathbb{R}^{n \times h}$, $W_2 \in \mathbb{R}^{h \times k}$
- Forward pass:

$$\mathbf{a}_1 = \sigma(XW_1)$$
$$\mathbf{a}_2 = \text{softmax}(\mathbf{a}_1 W_2)$$

- Training relies on matrix operations




Neural network

# Take Home Message



**Key Point**

Applied mathematics is fundamentally about solving combinations of linear algebra problems.

**Modern challenges:**

- Ever larger problem sizes
- Need for reliable results in reasonable time
- Requirements: *efficient*, *scalable*, *parallel* algorithms

$\Rightarrow$ This motivates high-performance numerical linear algebra!

# Recommended Reading



Recommended books on Linear Algebra and Numerical Linear Algebra include:

- Golub and Van Loan [4] - a classic covering matrix factorizations, eigenvalue problems, and singular value decomposition.
- Other notable works: [2], [8].
- Axler [1] offers an operative introduction to the theory of linear algebra.
- For comprehensive theory, see Horn and Johnson [5, 6].

We will focus on numerical and implementation aspects, with references for deeper theoretical insights.

# Table of Contents

2 What does it mean large-scale?


- Introduction
  - Why high-performance Linear Algebra?
  - A gallery of problems
- What does it mean large-scale?
- Where do we solve such *large* problems?
  - The TOP500 list
  - The EuroHPC Joint Undertaking: www.eurohpc-ju.europa.eu
- What tools are we going to use?
  - (Modern) Fortran
  - Git
    - An example of Git usage

# What does “large-scale” mean?



In the previous section we have seen examples of problems in numerical linear algebra, where a recurrent theme is that the problem sizes are *large*.

**But how large is large?**

**The answer: It depends!**

It depends on:

- The problem we are dealing with
- The algorithm we are using
- The hardware we are using
- The time we have to solve the problem

# What does “large-scale” mean?



In the previous section we have seen examples of problems in numerical linear algebra, where a recurrent theme is that the problem sizes are *large*.

**But how large is large?**
Furthermore, it’s a matter of *when* we are asking this question:

- 20 years ago: different answer
- Today: different answer
- 20 years from now: yet again different!

# Problem size: different perspectives



The notion of “size” varies by problem type:

**Linear systems:**

- First approximation: number of unknowns
- **Sparse matrices**: combined information
  - Number of non-zero elements
  - Overall matrix dimensions
- **Dense matrices**: number of rows and columns

# Current capabilities

2 What does it mean large-scale?

**Sparse linear systems:**

- Solved with relative ease: several millions of unknowns
- Current frontier: hundreds of billions of unknowns

**Eigenvalue problems:**

- Compute few eigenvalues/eigenvectors
- Matrices with several millions of rows and columns

**Matrix equations:**

- More complicated situation
- Need to exploit special structure for large-scale problems

# Exploiting structure: low-rank solutions

## 2 What does it mean large-scale?

For large matrix equations, we need solutions with special structure.

**Example: Sylvester equation with low-rank solution**

$$T = T_1 T_2^\top, \quad \text{where } T_1 \in \mathbb{R}^{m \times r}, T_2 \in \mathbb{R}^{n \times r}$$

with $r \ll m, n$

**Key principle**

**Exploiting clever structures** in the problem permits us to solve problems of larger size than we would be able to without these structures.

**Computer science analogy:** Building **data structures** that permit us to store and manipulate large amounts of data more efficiently.

# Table of Contents

3 Where do we solve such *large* problems?


- Introduction
  - Why high-performance Linear Algebra?
  - A gallery of problems
- What does it mean large-scale?
- Where do we solve such *large* problems?
  - The TOP500 list
  - The EuroHPC Joint Undertaking: www.eurohpc-ju.europa.eu
- What tools are we going to use?
  - (Modern) Fortran
  - Git
    - An example of Git usage

# Solving large problems: parallel computers

3 Where do we solve such *large* problems?

To deal with problems which are large in the sense we have just discussed, we need to use **parallel computers**.

**Key idea**

Parallel computers perform multiple calculations **simultaneously** by using multiple processors or cores working together.

# Memory organization

## 3 Where do we solve such *large* problems?

Parallel computers are classified by memory organization:

- **Shared memory systems**:
  - All processors share common memory space
  - Easy data access and communication
  - Limited by memory size and contention

- **Distributed memory systems**:
  - Each processor has local memory
  - Communication via message passing
  - Allows larger memory, but requires complex programming

# Parallel architectures

## 3 Where do we solve such *large* problems?

**Common parallel computing architectures:**

- **Multicore processors**:
  - Multiple cores on single chip
  - Each core executes independent thread
- **Clusters**:
  - Interconnected computers (nodes)
  - Communication through network
- **Supercomputers**:
  - Extremely powerful systems
  - Thousands of processors working in parallel
  - Designed for high-speed complex calculations

# The TOP500 list: top500.org

3 Where do we solve such *large* problems?

The **TOP500** list ranks the 500 most powerful supercomputers worldwide.

- Updated biannually (June and November)
- Ranks based on LINPACK benchmark performance
- Provides insights into trends in high-performance computing

**Current leader (as of 2025):** El Capitan (USA) with a performance of over 1 exaFLOP ($10^{18}$ floating-point operations per second).

# TOP500 List (June 2025) - Part 1

3 Where do we solve such *large* problems?

| Rank | System | Cores | Rmax (PFlop/s) | Rpeak (PFlop/s) | Power (kW) |
|---|---|---|---|---|---|
| 1 | **El Capitan**, HPE Cray EX255a, AMD EPYC 24C, DOE/NNSA/LLNL, United States | 11,039,616 | 1,742.00 | 2,746.38 | 29,581 |
| 2 | **Frontier**, HPE Cray EX235a, AMD EPYC 64C, DOE/SC/ORNL, United States | 9,066,176 | 1,353.00 | 2,055.72 | 24,607 |
| 3 | **Aurora**, HPE Cray EX, Intel Xeon Max 9470, DOE/SC/ANL, United States | 9,264,128 | 1,012.00 | 1,980.01 | 38,698 |

# TOP500 List (June 2025) - Part 2

3 Where do we solve such *large* problems?

| Rank | System | Cores | Rmax (PFlop/s) | Rpeak (PFlop/s) | Power (kW) |
|---|---|---|---|---|---|
| 4 | **JUPITER Booster**, BullSequana XH3000, NVIDIA GH200, EuroHPC/FZJ, Germany | 4,801,344 | 793.40 | 930.00 | 13,088 |
| 5 | **Eagle**, Microsoft NDv5, Xeon Platinum 8480C, Microsoft Azure, United States | 2,073,600 | 561.20 | 846.84 | — |
| 6 | **HPC6**, HPE Cray EX235a, AMD EPYC 64C, Eni S.p.A., Italy | 3,143,520 | 477.90 | 606.97 | 8,461 |

# TOP500 List (June 2025) - Part 3

3 Where do we solve such *large* problems?

| Rank | System | Cores | Rmax (PFlop/s) | Rpeak (PFlop/s) | Power (kW) |
|---|---|---|---|---|---|
| 7 | **Supercomputer Fugaku**, Fujitsu, A64FX 48C 2.2GHz, RIKEN CCS, Japan | 7,630,848 | 442.01 | 537.21 | 29,899 |
| 8 | **Alps**, HPE Cray EX254n, NVIDIA Grace 72C, CSCS, Switzerland | 2,121,600 | 434.90 | 574.84 | 7,124 |
| 9 | **LUMI**, HPE Cray EX235a, AMD EPYC 64C, EuroHPC/CSC, Finland | 2,752,704 | 379.70 | 531.51 | 7,107 |
| 10 | **Leonardo**, BullSequana XH2000, Xeon Platinum, EuroHPC/CINECA, Italy | 1,824,768 | 241.20 | 306.31 | 7,494 |

# High Performance Linpack (HPL) Benchmark

## 3 Where do we solve such *large* problems?

The computers in this table are ranked according to *Rmax*, the maximum sustained performance; but how is this measured? This is the High Performance Linpack (HPL) benchmark, which is run according to the following rules:

1. Generate a (random) linear system $Ax = b$ of size $N$ and solve for $x$;
2. Measure the time for the solution process $T$ and define a computation rate $R(N)$ according to the formula
$$R = \frac{2}{3}\frac{N^3}{T};$$
3. Let $N$ grow and repeat the process, until you get the best possible execution rate value *Rmax*.

# Importance of Linear Algebra in Benchmarking

## 3 Where do we solve such *large* problems?

Linear algebra problems have been used to benchmark supercomputers for a very long time, influencing their design in multiple ways.
Key observations:

- Supercomputers have a huge number of cores.
- Operating them consumes a lot of power.
- They are equipped with accelerators, specifically **graphical processing units** (GPUs).

# Sustained Performance and Historical Context

## 3 Where do we solve such *large* problems?

The *sustained* rate of execution on the HPL benchmark shows that the number one machine, El Capitan, is capable of executing $1.7 \times 10^{18}$ arithmetic operations per second!

- **Linear algebra** is a primary tool for **benchmarking supercomputers**.
- *Dense linear algebra* problems are **compute-bound**, enabling hardware to operate close to peak performance.

# Evolution of HPL Benchmark

3 Where do we solve such *large* problems?

The Linpack benchmark originated from tests in the LINPACK User's Guide [3].

- It has evolved into a standardized benchmark for comparing computing systems.
- The current HPL benchmark allows vendors to choose problem size and software configuration for optimal performance.
- Continuous interaction between supercomputing advances and linear algebra has driven innovations in algorithms and software.

# The EuroHPC Joint Undertaking

## 3 Where do we solve such *large* problems?

**EuroHPC JU** is a European initiative to develop a world-class supercomputing ecosystem in Europe.

- Established in 2018
- Partnership between the European Union, European countries, and private sector
- Aims to provide access to high-performance computing resources for research, industry, and public sector

**Key objectives:**

- Deploy and operate supercomputers in Europe
- Foster research and innovation in HPC technologies
- Support development of HPC applications across various sectors

# EuroHPC Supercomputers


EuroHPC
Joint Undertaking

# The EuroHPC Pre-exascale Machines

## 3 Where do we solve such *large* problems?

- **JUPITER** (Germany) - First European exascale system
  - NVIDIA GH200 Grace Hopper GPUs
  - 793.4 PFlop/s (Rmax)
- **LUMI** (Finland) - One of world's fastest and most energy-efficient
  - AMD Instinct MI250X GPUs
  - 379.7 PFlop/s (Rmax)
- **Leonardo** (Italy) - General-purpose HPC system
  - NVIDIA A100 GPUs
  - 241.2 PFlop/s (Rmax)

# The EuroHPC Petascale Machines

## 3 Where do we solve such *large* problems?

- **MELUXINA** (Luxembourg) - Modular supercomputing architecture
  - NVIDIA A100 GPUs
  - 18.2 PFlop/s (Rmax)
- **Vega** (Slovenia) - First EuroHPC system in Eastern Europe
  - NVIDIA A100 GPUs
  - 6.9 PFlop/s (Rmax)
- **Karolina** (Czech Republic) - Accelerated computing platform
  - NVIDIA A100 GPUs
  - 15.2 PFlop/s (Rmax)
- **Discoverer** (Bulgaria) - Supporting research and innovation
  - NVIDIA A100 GPUs
  - 3.0 PFlop/s (Rmax)
- **MareNostrum 5** (Spain) - Upgrade of iconic BSC system
  - NVIDIA Hopper GPUs
  - 314.0 PFlop/s (Rmax)

# Table of Contents

4 What tools are we going to use?


- Introduction
  - Why high-performance Linear Algebra?
  - A gallery of problems
- What does it mean large-scale?
- Where do we solve such *large* problems?
  - The TOP500 list
  - The EuroHPC Joint Undertaking: www.eurohpc-ju.europa.eu
- What tools are we going to use?
  - (Modern) Fortran
  - Git
    - An example of Git usage

# Programming Distributed Memory Systems

## 4 What tools are we going to use?

In this course, we focus on **distributed memory systems**:

- Most common in High-Performance Computing (HPC)
- Composed of many nodes, each with local memory
- Communication via message-passing libraries (e.g., MPI)

Before diving into the programming model, let's discuss the **tools** we'll use to write efficient parallel code.

# Tools for High-Performance Linear Algebra

## 4 What tools are we going to use?

**Modern Fortran**

- Long-standing language for scientific computing
- Well-suited for numerical computations
- Still widely used in scientific applications

**Software Version Control: git**

- Track changes to code
- Collaborate effectively with others
- Essential for team development

# Parallel Programming Tools

4 What tools are we going to use?

**MPI, OpenMP, OpenACC, CUDA and other tools**

- **MPI (Message Passing Interface)**: Distributed memory parallelism
- **OpenMP**: Shared memory parallelism for many-core processors
- **OpenACC and CUDA**: Accelerator/GPU programming
  - Modern supercomputers are equipped with GPUs
  - Essential for leveraging full system capabilities

**Job Scheduler: Slurm**

- Manage execution of jobs on the cluster
- Resource allocation and job queuing

# What is Fortran?

4 What tools are we going to use?

**Fortran** (“Formula Translation”) is one of the oldest high-level programming languages:

- Originally developed in the 1950s by IBM
- Designed for scientific and engineering applications
- Easy translation of mathematical formulas into code

**Modern versions include:**

- Fortran 90, 95, 2003, 2008, 2018, 2023
- Features: modular programming, array operations, OOP, parallel computing

**Note**

Most concepts discussed can be ported to C/C++ or other compiled languages. For more on Fortran, see fortran-lang.org and [7].

# Why Fortran for HPC?

## 4 What tools are we going to use?

**Key strengths:**

- **High performance** in numerical computations
- Highly optimized for array and matrix operations
- Efficient machine code generation
- Preferred choice for HPC applications

**Programming paradigms supported:**

- Procedural, modular, and object-oriented programming
- Parallel programming features (coarrays, MPI, OpenMP)
- Scalable code for distributed and shared memory systems

**Applications:** Climate modeling, computational fluid dynamics, numerical linear algebra

# Fortran Compilers

4 What tools are we going to use?

**Available compilers:**

- **GNU Fortran** (`gfortran`) - Part of GCC
- **Intel Fortran** (`ifort`) - Optimized for Intel architectures
- **Cray Fortran** (`ftn`) - For Cray supercomputers
- **LLVM Fortran** (`flang`)
- **PGI Fortran** (`pgfortran`)
- **NAG compiler** (`nagfor`)

**Our choice:** `gfortran`

- Widely available and default on many systems
- Up to date with latest Fortran standards
- Free and open source

# Installing gfortran

4 What tools are we going to use?

## Checking installation

To check if `gfortran` is installed:

```
gfortran --version
```

**Installation options:**

- **Ubuntu/Debian:**
  `sudo apt-get install gfortran`
- **macOS (via Homebrew):**
  `brew install gcc`
- **Using Spack:** Download from spack.io or GitHub

# Basic gfortran Usage

## 4 What tools are we going to use?

**Basic compilation syntax:**

```
gfortran -o output_file source_file.f90
```

**Common options:**

- `-o output_file` - Specify output executable name
- `-Wall` - Enable all compiler warnings
- `-g` - Generate debug information
- `-O0, -O1, -O2, -O3` - Optimization levels
- `-fcheck=all` - Enable runtime checks
- `-frecursive` - Enable recursion
- `-fPIC` - Position-independent code for shared libraries

# Your First Fortran Program

## 4 What tools are we going to use?

Create a file `hello.f90`:

```
program hello
    use iso_fortran_env, only: output_unit
    implicit none
    write(output_unit,'("Hello, World!")')
end program hello
```

**Compile and run:**

```
gfortran -o hello hello.f90
./hello
```

**Output:**

```
Hello, World!
```

**Note:** `implicit none` enforces explicit variable declaration (good practice!)

# What is Version Control?

4 What tools are we going to use?

**Version control** is a system that records changes to files over time:

- Recall specific versions later
- Track changes to files and code
- Enable collaboration without conflicts
- Revert to previous versions when needed

**Benefits:**

- Multiple developers work simultaneously
- Compare changes over time
- Collaborate more effectively

# Types of Version Control Systems

4 What tools are we going to use?

- **Centralized Version Control Systems (CVCS)**
  - Central server stores the repository
  - Developers check out files from central location
  - Examples: Subversion (SVN), CVS
  - **Drawback:** Server failure stops all work
- **Distributed Version Control Systems (DVCS)**
  - Complete repository copy on each developer's machine
  - Enables offline work
  - Better collaboration capabilities
  - Examples: Git, Mercurial, Bazaar

# What is Git?

4 What tools are we going to use?

**Git** is a distributed version control system:

- Created by Linus Torvalds in 2005 for Linux kernel development
- Designed for speed and efficiency
- Handles projects from small to very large

**Key features:**

- Track changes to files
- Collaborate with others
- Manage different versions of codebase
- Powerful branching and merging capabilities

**Current status:** *De facto* standard for version control in software development

# Basic Git Workflow

## 4 What tools are we going to use?

**Initialize a new repository:**

```
git init my_project
cd my_project
```

Create or add files to the repository, e.g., `hello.f90` **Check repository status:**

```
git status
```

**Stage changes for commit:**

```
git add hello.f90
```

**Commit changes with a message:**

```
git commit -m "Add hello.f90 program"
```

**View commit history:**

```
git log
```

# Cloning a Remote Repository or add a Remote



**Clone a remote repository:**

```
git clone <repository-url>
cd <repository-name>
```

**Add a remote to an existing repository:**

```
git remote add origin <repository-url>
```

**Push local commits to remote:**

```
git push origin main
```

**Pull changes from remote repository:**

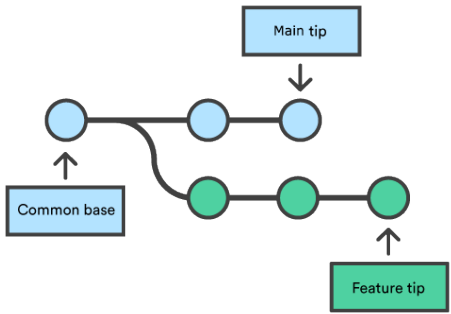
```
git pull origin main
```

# Branching and Merging

## 4 What tools are we going to use?

**Branching** allows you to create a separate line of development:

- Isolate features or bug fixes
- Experiment without affecting the main codebase

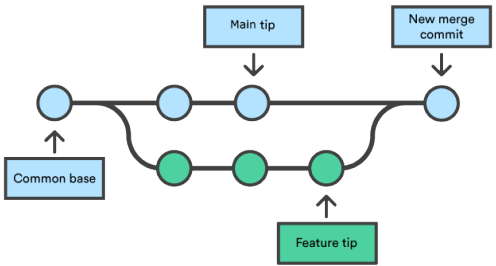


# Branching and Merging

## 4 What tools are we going to use?

**Merging** combines changes from different branches:

- Integrate new features or fixes
- Resolve conflicts when changes overlap



# Branching and Merging: Example



**Create a new branch:**

```
git checkout -b new-feature
```

**Make changes and commit:**

```
git add .
git commit -m "Implement new feature"
```

**Switch back to main branch:**

```
git checkout main
```

**Merge changes from new-feature branch:**

```
git merge new-feature
```

# Services that use Git

4 What tools are we going to use?

**Popular platforms for hosting Git repositories:**

- **GitHub** (github.com)
- **GitLab** (gitlab.com)
- **Bitbucket** (bitbucket.org)

We also have a Gitea instance installed at the Math Department: git.phc.dm.unipi.it.

## Exercise

Explore the features of the mentioned Git hosting platforms and create an account on GitHub. This will require you to setup SSH keys for secure access to your repositories.

```
ssh-keygen -t ed25519 -C "your_email@example.com"
ssh-add ~/.ssh/id_ed25519
cat ~/.ssh/id_ed25519.pub
```

After you have done it, tell me the username and I'll add you to the course organization.

# Summary

5 Summary

**Key takeaways from this lecture:**

- High-performance computing relies heavily on linear algebra
- The TOP500 list ranks the most powerful supercomputers using the HPL benchmark
- We will use Modern Fortran and Git for programming and version control

**Next steps:**

- Set up your Fortran development environment
- Familiarize yourself with Git and version control
- Explore CI/CD tools for automating workflows

**Next lecture** will cover: introduction to parallel computing from a theoretical standpoint, including models and paradigms, but in relation to linear algebra problems.

# References

6 Bibliography


[1] S. J. Axler. *Linear Algebra Done Right*. Undergraduate Texts in Mathematics. New York: Springer, 1997. ISBN: 0387982582. URL: http://linear.axler.net/.

[2] J. W. Demmel. *Applied Numerical Linear Algebra*. Philadelphia, PA: SIAM, 1997. ISBN: 0-89871-389-7.

[3] J. J. Dongarra et al. *LINPACK User's Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1979. ISBN: 0-89871-172-X. DOI: 10.1137/1.9781611971811.

[4] G. H. Golub and C. F. Van Loan. *Matrix computations*. Fourth. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, MD, 2013, pp. xiv+756. ISBN: 978-1-4214-0794-4; 1-4214-0794-9; 978-1-4214-0859-0.

[5] R. A. Horn and C. R. Johnson. *Matrix analysis*. Second. Cambridge University Press, Cambridge, 2013, pp. xviii+643. ISBN: 978-0-521-54823-6.

# References

6 Bibliography


[6] R. A. Horn and C. R. Johnson. *Topics in matrix analysis*. Corrected reprint of the 1991 original. Cambridge University Press, Cambridge, 1994, pp. viii+607. ISBN: 0-521-46713-6.

[7] M. Metcalf et al. *Modern Fortran Explained: Incorporating Fortran 2023*. 6th ed. Numerical Mathematics and Scientific Computation. Oxford, UK: Oxford University Press, 2024. ISBN: 978-0198876588.

[8] L. N. Trefethen and D. Bau. *Numerical Linear Algebra*. SIAM, 1997.

# High Performance Linear Algebra

## Lecture 2: Performance Modeling and the Roofline Model

Ph.D. program in High Performance Scientific Computing

**Fabio Durastante** **Pasqua D'Ambra** **Salvatore Filippone**

November 14, 2025 — 9.00:11.00

Dipartimento
di Matematica
Università di Pisa

# Summary of previous lecture



In the **previous lecture** we have discussed the following topics:

- Introduction to High Performance Scientific Computing
- Overview of Linear Algebra and its importance in scientific computing
- The machines from the TOP500 list
- What tools are we going to use in this course
  - Programming language: Fortran
  - Software Versioning: Git and GitHub

# Table of Contents

2 Continuous Integration and Deployment (CI/CD)


- ▶ Continuous Integration and Deployment (CI/CD)
- ▶ General Parallel Programming Issues
  - Basic concepts
  - Parallel Performance Metrics
  - Parallelism: Performance metrics
  - Scalability of a parallel system
  - Speed-up and efficiency
  - Amdahl's law
  - Gustafson's law
- ▶ Paradigms, models and tools for parallel programming
  - Algorithmic paradigms
  - Programming models
  - Programming tools

# What is CI/CD?



**Continuous Integration (CI)** and **Continuous Deployment (CD)** are practices in software development that automate the process of integrating code changes and deploying applications.

**Continuous Integration (CI):**

- Developers frequently merge code changes into a shared repository
- Automated builds and tests run to detect issues early

**Continuous Deployment (CD):**

- Automatically deploys code changes to production after passing tests
- Ensures rapid delivery of new features and bug fixes

# Benefits of CI/CD



**Key benefits:**

- **Early bug detection:** Automated tests catch issues before they reach production
- **Faster development cycles:** Rapid integration and deployment of changes
- **Improved collaboration:** Teams work together more effectively with shared codebase
- **Higher quality software:** Consistent testing and deployment processes

**Popular CI/CD tools:**

- GitHub Actions
- GitLab CI/CD
- Jenkins
- Travis CI

# An example of CI/CD with GitHub Actions



**Setting up a simple CI workflow:** we will use GitHub Actions to automatically build and test our Fortran code whenever we push changes to the repository
First, ensure your Fortran project has a `Makefile` with appropriate build and test targets. This can be as simple as:

```
all:
    gfortran -o hello hello.f90
test:
    ./hello
```

We recall that a **Makefile** is a file that defines a set of tasks to be executed. It is commonly used to automate the build process of software projects.

# Makefile basics



A **Makefile** consists of rules with the following structure:

```
target: dependencies
    command
```

**Example:**

```
hello: hello.f90
    gfortran -o hello hello.f90
```

Here, `hello` is the target, `hello.f90` is the dependency, and the command compiles the Fortran source file into an executable named `hello`.
For **small projects**, a simple Makefile like this is sufficient to automate the build and test process. For **larger projects**, it is better to also have the Makefile programmatically generated using tools like CMake or Autotools.

# Creating a workflow on GitHub Actions



**GitHub Actions** allows you to automate workflows directly in your GitHub repository.

**Key components:**

- **Workflows:** Automated processes defined in YAML files
- **Jobs:** A set of steps that execute on the same runner
- **Steps:** Individual tasks within a job (e.g., running commands, setting up environments)

In GitHub, workflows are stored in the `.github/workflows/` directory of your repository as YAML (`.yml`, Yet Another Markup Language) files.

# An example of CI/CD with GitHub Actions



Create a file `.github/workflows/ci.yml` in your repository:

```yaml
name: CI
on:
  push:
    branches:
      - main
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4
      - name: Set up Fortran
```

# An example of CI/CD with GitHub Actions



```
      run: |
        sudo apt-get update
        sudo apt-get install -y gfortran
    - name: Test
      run: make test
```

**Explanation:**

- Triggered on **pushes** to the `main` branch

```
on:
  push:
    branches:
      - main
```

- Runs on the latest Ubuntu environment

# An example of CI/CD with GitHub Actions



- Defines a job named `build` with several steps:
  - Checkout code from the repository

```yaml
      - name: Checkout code
        uses: actions/checkout@v4
```

  - Install `gfortran`

```yaml
      - name: Set up Fortran
        run: |
          sudo apt-get update
          sudo apt-get install -y gfortran
```

  - Build the project using `make`

```yaml
      - name: Build
        run: make
```

# An example of CI/CD with GitHub Actions



— Run tests using `make test`

```
      - name: Test
        run: make test
```

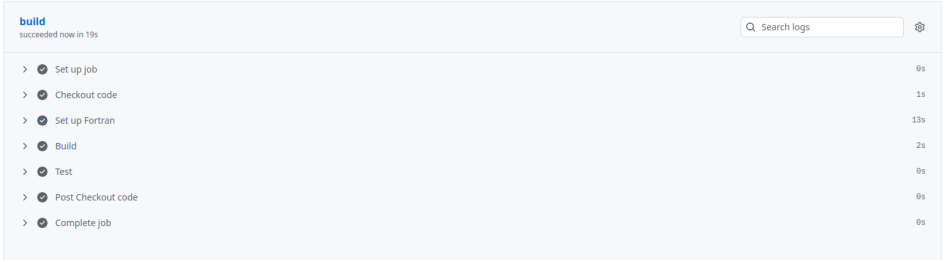
You can change the “manual installation” of `gfortran` with a pre-built action from the GitHub Marketplace, such as setup-fortran:

```
  - name: Setup Fortran
    uses: fortran-lang/setup-fortran@v1.8.0
    with:
      compiler: gcc
      version: 'latest'
      update-environment: true
```

# If everything has gone well:

## 2 Continuous Integration and Deployment (CI/CD)

From the top menu of your GitHub repository, click on the **Actions** tab. You should see your workflow running, and if everything is set up correctly, it should complete successfully, indicating that your Fortran code has been built and tested automatically.




**Example:** github.com/High-Performance-Linear-Algebra/hello-fortran-world

# Learning by doing:

## 2 Continuous Integration and Deployment (CI/CD)

Explore the setup-fortran action and modify the provided example workflow to include additional steps, such as:

- Running on different operating systems (e.g., Windows, macOS)
- Running on different Fortran compilers (e.g., Intel Fortran, NVIDIA HPC SDK)

**Question:** How would you modify the Makefile so that it does not call `gfortran` directly, but uses instead the compiler available in the environment?

**Moving to CMake:** To automate the search for the compiler, configuration, and building of projects, a good practice is to use CMake.

# Further informations on the GitHub Actions



For more information on GitHub Actions, refer to the official documentation:

- GitHub Actions Documentation
- Introduction to GitHub Actions
- Workflow syntax for GitHub Actions

They can help you explore more advanced features and customize your CI/CD workflowsm, such as:

- Running tests on multiple operating systems
- Integrating with other services
- Deploying applications automatically
- Setting up notifications for build status
- Deploying Documentation, artifacts, and more

# Table of Contents

3 General Parallel Programming Issues


- ▶ Continuous Integration and Deployment (CI/CD)
- ▶ General Parallel Programming Issues
  - Basic concepts
  - Parallel Performance Metrics
  - Parallelism: Performance metrics
  - Scalability of a parallel system
  - Speed-up and efficiency
  - Amdahl's law
  - Gustafson's law
- ▶ Paradigms, models and tools for parallel programming
  - Algorithmic paradigms
  - Programming models
  - Programming tools

# General Parallel Programming Issues



In **this lecture** we will discuss some general issues related to parallel programming:

- Performance metrics
- Scalability
- Speedup and Efficiency
- Amdahl's and Gustafson's Laws
- Programming Models
- Parallel Architectures
- Roofline Model

# What do we mean by parallelism?



**Definition**

We call *parallelism* the ability to have multiple operations completing their execution at the same time.

- “Operation” may mean a machine instruction, a floating-point operation, or something else depending on context.
- In scientific/engineering applications, the key metric is often the number of floating-point operations (FLOPs), typically the limiting factor for execution speed.

# What do we mean by parallelism?



## Keywords: FLOP and FLOP/s

A *FLOP* is a floating-point operation, typically an addition, subtraction, multiplication, or division between two floating-point numbers. The number of FLOPs required to solve a problem is often used as a measure of the problem's **computational complexity**. The amount of FLOP/s (floating-point operations per second) a computer can perform is a key measure of its performance, and is often used to rank supercomputers (e.g., the TOP500 list).

# Levels of parallelism in a computing system



- Within a single machine instruction specifying multiple operations (e.g., SSE on x86, fused multiply–add on many architectures).
- Within a single processor completing more than one instruction per clock cycle; many modern RISC processors have multiple execution units and are *superscalar*.
- Within a single silicon chip hosting multiple CPUs; *multicore* processors[1], a core being each complete CPU.
- Within a single computer containing multiple processors.
- Using multiple computers connected through some sort of communication device.

[1] This usage is slightly confusing, since we are calling *processor* both a single CPU and a chip hosting multiple cores; context should avoid confusion.

# Programmer's perspective



- The first two kinds of parallelism (*single machine* and *single processor*) are mostly handled by the compiler and optimized libraries implementing heavy computational kernels.
- They are almost transparent to application programmers.
- We will concentrate on the **last three kinds of parallel computing systems**:
  - *Single chip* (multicore) systems.
  - *Single computer* with multiple processors.
  - *Multiple computers* connected by a network.

# Parallel systems: memory configurations



In classifying high-performance parallel computers, the discriminating factor is the memory subsystem configuration. Two main kinds:

1. *Shared memory* systems.
2. *Distributed memory* systems.

# Flynn's Taxonomy



## Overview

Since the early 1970s, the **Flynn taxonomy** [1, 2] has been the standard classification scheme for computer architectures.

**Four categories:**

| | |
|---|---|
| SISD | **S**ingle **I**nstruction **S**ingle **D**ata |
| SIMD | **S**ingle **I**nstruction **M**ultiple **D**ata |
| MISD | **M**ultiple **I**nstruction **S**ingle **D**ata |
| MIMD | **M**ultiple **I**nstruction **M**ultiple **D**ata |

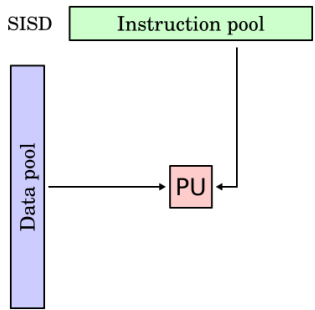
# Flynn's Taxonomy: SISD



## Single Instruction Single Data

Sequential computers where a single stream of data is processed by a single stream of instructions.

- Traditional von Neumann architecture
- One instruction operates on one data element at a time
- No parallelism at the architecture level



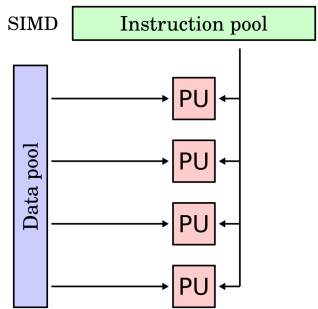
# Flynn's Taxonomy: SIMD



## Single Instruction Multiple Data

Vector processors capable of handling multiple data with a single instruction.

- Data presented in the form of a vector
- **Notable example:** Cray-1 computer with vector registers of length 64
- Modern examples: SSE/AVX instructions on x86, AltiVec on Power processors



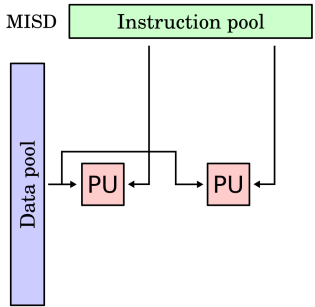
# Flynn's Taxonomy: MISD



## Multiple Instruction Single Data

Multiple instruction streams operating on the same data.

- No significant examples of such architectures have been built
- Flynn classified ancient plug-board machines in this category
- Some embedded devices use this for **fault tolerance**:
  - Same instruction executed redundantly in multiple streams
  - Results verified for accordance



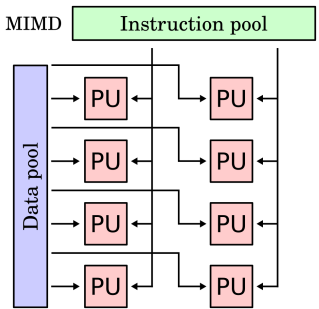
# Flynn's Taxonomy: MIMD



## Multiple Instruction Multiple Data

Multiple instruction streams concurrently operate on different (sub)sets of data.

- **The vast majority** of parallel computers built in the last 30 years
- Includes both:
  - Shared memory multiprocessors
  - Distributed memory multiprocessors
- Most flexible and powerful category



# Modern Supercomputers: Beyond Flynn



- Flynn's taxonomy still useful for **rough categorization**
- Modern landscape is much more complex
- Current supercomputers are **hybrid combinations** of different architectures

The Marenostrum 5 supercomputer at the Barcelona Supercomputing Center (BSC).

# Anatomy of a Modern Supercomputer

## 3 General Parallel Programming Issues

A typical modern supercomputer consists of:

1. **Multiple nodes** connected through a network interconnect
2. **Multiple processors per node**, possibly in NUMA configuration
   - NUMA = Non-Uniform Memory Access
3. **Multicore processors** (multiple cores per processor)
4. **SIMD units per core**
   - SSE/AVX on x86
   - AltiVec on Power processors
   - Vector instructions of size 2 or 4
5. **Accelerators** (e.g., GPUs)
   - Own memory system
   - Connected via PCI or high-speed interconnect (e.g., NVLink)



## The Challenge

Exploiting the computational power of modern supercomputers demands:

- Programming models capable of matching their **hierarchical structure**
- Algorithms that can handle their **heterogeneity**
- Understanding of performance metrics for parallel programs

**Coming up:** Basic concepts of parallel programming and performance evaluation

# Performance metrics in parallel computing



- Many alternatives exist: hardware architectures, programming paradigms, applications.
- It is necessary to **define metrics** to evaluate the performance of a parallel system.

# Performance metrics in parallel computing



- Many alternatives exist: hardware architectures, programming paradigms, applications.
- It is necessary to **define metrics** to evaluate the performance of a parallel system.

**No single criterion fits all**

- **Computer scientist:** pure algorithmic speed-up.
- **Computational scientist:** time to completion; maximum problem size that can be analyzed.
- **System administrator:** maximize system utilization.

# Beware of benchmarks



- No substitute for testing with the workload of interest.
- Benchmarks are indicators only as good as their relation to the intended usage.
- Procurement differs: single critical application vs computing center serving many users.

# What do we mean by scalability?



- A parallel system = implementation of a parallel algorithm on a given parallel architecture.
- Scalability theory organizes performance evaluation while accounting for usage aspects.
- We must choose appropriate metrics depending on goals and constraints.

# Questions a metric should answer



- How do we measure the raw performance of a system?
- How do we compare measurements obtained on different machines?
- How does the metric respond to the programming paradigm employed?
- Do we want raw performance or value for money?

**Note**

Given the variety of questions, only general criteria exist; there is no single, precise measurement procedure.

# Definitions of scalability



- A parallel system is scalable if it delivers the same performance per processor while increasing the number of processors and/or the problem size.
- A program is scalable if its performance improves when increasing the processors employed from $p - 1$ to $p$.

# Workload, execution times, and I/O



- Define problem size $W$ as the number of basic operations for the best-known sequential algorithm.

[2] See e.g. ROOT from CERN: root.cern and CAPIO: github.com/High-Performance-IO/capio.

# Workload, execution times, and I/O



- Define problem size $W$ as the number of basic operations for the best-known sequential algorithm.

**How do we know what is the *best* sequential algorithm?**

There exist a branch of computer science, called *computational complexity theory*, that studies the **inherent difficulty** of computational problems and classifies them according to the resources needed to solve them. However, in practice, we often rely on empirical performance measurements and established benchmarks to determine the best-known algorithms for specific problems.

[2] See e.g. ROOT from CERN: root.cern and CAPIO: github.com/High-Performance-IO/capio.

# Workload, execution times, and I/O



- Define problem size $W$ as the number of basic operations for the best-known sequential algorithm.
- **Serial time $T_s$:** start-to-end time on one processor.
- **Parallel time $T_p$** on $p$ processors: time until the last processor completes.
- **I/O** may be included or measured separately depending on goals; if I/O is essential, consider parallel I/O.[2]

[2] See e.g. ROOT from CERN: root.cern and CAPIO: github.com/High-Performance-IO/capio.

# Execution-time models and benchmarking



- $T_s = f(W)$
- $T_p = f(W, p, \text{arch})$
- Absolute assessment is application-dependent; for example, the TOP500 rules measure **dense linear algebra factorization performance** (remind the TOP500 list).



**Definition**

$S(W,p) = T_s(W)/T_p(W,p)$

- Linear: $S = p$
- Sub-linear: $S < p$
- Super-linear: $S > p$

Linear speed-up is the *ideal target*; in practice communication and overheads limit it.

# Why speed-up is often sub-linear



- Startup costs
- Communication latency/bandwidth limits
- Synchronization overheads

**Example:** consider a workload with $W$ operations and communication overhead $C$ per processor. The parallel time can be modeled as:

$$T_p(W, p) = \frac{W}{p} + C \cdot p$$

The speed-up is then:

$$S(W, p) = \frac{W}{\frac{W}{p} + C \cdot p} = \frac{p}{1 + \frac{C \cdot p^2}{W}}$$

With fixed $W$, the fraction of time spent communicating typically grows with $p$.

# When super-linear speed-up may occur



1. Better use of **memory hierarchy**: partitioning reduces working set per process, improving cache and memory behavior (coming back to this in a few slides).
2. **Search problems:** parallel decomposition changes exploration order, finding solutions earlier; sometimes even $S > 1$ on a single core if extra memory helps.

**Memory hierarchy:** In many computing systems, memory is organized in a hierarchy of levels, each with different speeds and sizes. The fastest level is the CPU cache, followed by main memory (RAM), and then slower storage devices like hard drives or SSDs. When a problem is divided among multiple processors, each processor may work on a smaller subset of the data, which can fit better into the faster levels of the memory hierarchy. This improved data locality can lead to reduced memory access times and increased overall performance, resulting in super-linear speed-up in some cases.

# When super-linear speed-up may occur



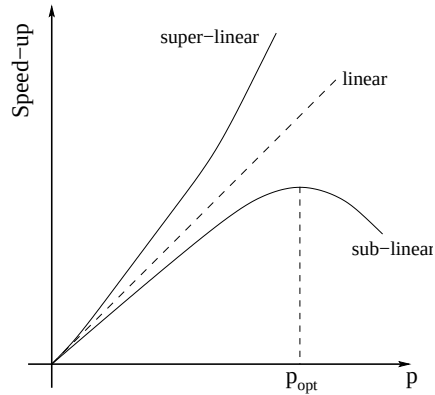
1. Better use of **memory hierarchy**: partitioning reduces working set per process, improving cache and memory behavior (coming back to this in a few slides).
2. **Search problems:** parallel decomposition changes exploration order, finding solutions earlier; sometimes even $S > 1$ on a single core if extra memory helps.

**Note:** Super-linear speed-up is rare and often specific to certain problem structures or algorithmic techniques. For the linear algebra problems we will consider, it is highly uncommon.

# Speed-up vs number of processors



- Linear, sub-linear, and occasional super-linear trends.
- For fixed $W$, the gap from linear typically widens as $p$ increases due to rising overheads.

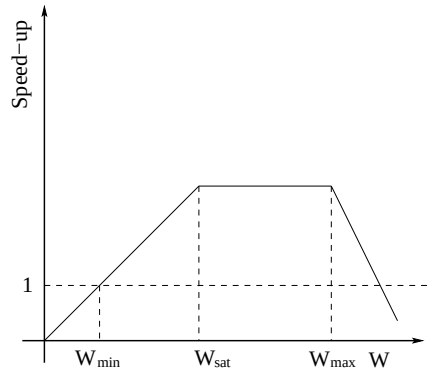



Speed-up vs processors

# Speed-up vs problem size



- Benefit only within a range $[W_{\min}, W_{\max}]$.
- Small $W$: communication dominates ($S < 1$).
- Peak near $W_{\text{sat}}$; then saturation.
- For very large $W$, node memory pressure can degrade performance sharply.




Speed-up vs problem size

# Efficiency



**Definition**

$$E(W,p) = \frac{S(W,p)}{p},$$

or, substituting the definition of speed-up

$$E(W,p) = \frac{S(W,p)}{p} = \frac{T_s(W)}{T_p(W,p)\cdot p}$$

- Typically $E$ in $(1/p, 1]$, excluding rare super-linear cases.

# Two models of scalability



We can think of two complementary models of scalability:

- **Amdahl's law:** fixed workload $W$; increasing $p$.
- **Gustafson's law:** scale $W$ with $p$ to keep execution time constant.

For the first, the focus is on how speed-up degrades with more processors for a fixed problem size. For the second, the focus is on how larger problems can be solved in the same time as more processors are added.

## Strong-scaling vs weak-scaling

- **Strong-scaling:** how $T_p$ changes with $p$ at fixed $W$ (Amdahl).
- **Weak-scaling:** how $T_p$ changes with $p$ when $W$ scales with $p$ (Gustafson).

# Amdahl's model



- Serial fraction $f_s$: time inherently serial divided by $T_s(W)$
- Parallel fraction $f_p = 1 - f_s$

**Parallel time**

$$T_p(W, p) = T_s(W) f_s + T_s(W) f_p / p$$

# Amdahl's speed-up and limit



## Speed-up

$$S(W,p) = \frac{_s(W)}{_p(W,p)} = \frac{p}{1+(p-1)f_s}$$

## Asymptotic limit

$$\lim_{p\to\infty} S(W,p) = 1/f_s$$

- Example: $f_s = 5\% \Rightarrow S \leq 20$ regardless of $p$.

# Implications of Amdahl's law



- Reducing the *serial fraction* $f_s$ is critical for absolute performance.
- At fixed $W$, added sync/comm overhead may cap speed-up even if all code paths are parallelized.
- For many applications in large-scale linear algebra, we scale $W$ with $p$, making Amdahl less constraining in practice.

# Gustafson's law: scaled-speed perspective



- With serial fraction $\alpha$ and parallel fraction $(1-\alpha)$, scale the parallel work with $n$ processors:

$$W(n) = \alpha W + (1-\alpha)\, nW, \quad S'_n = \frac{W(n)}{W} = \alpha + (1-\alpha)\, n$$

- Better reflects solving larger problems with more processors.
- Caveat: $\alpha$ may change as the problem scales.

## Putting it together

- Workload = serial part + parallel part + parallelization/communication overhead.
- Realistic speed-up estimates must include all three and are application dependent.

# Table of Contents

4 Paradigms, models and tools for parallel programming


- ▶ Continuous Integration and Deployment (CI/CD)
- ▶ General Parallel Programming Issues
  - Basic concepts
  - Parallel Performance Metrics
  - Parallelism: Performance metrics
  - Scalability of a parallel system
  - Speed-up and efficiency
  - Amdahl's law
  - Gustafson's law
- ▶ Paradigms, models and tools for parallel programming
  - Algorithmic paradigms
  - Programming models
  - Programming tools

# Throughput vs parallel applications



- Parallel machines may run many independent serial jobs to maximize throughput.
- Related to work-pool ideas but outside our main focus on single parallel applications.

# Paradigms, models, and tools



**Paradigm** Logical structure imposed on a parallel algorithm.

**Model** How parallelism is expressed in code.

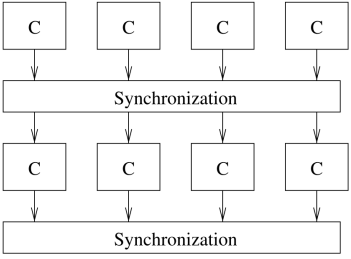
**Tool** Software instruments (compiler, libraries, etc.).

The right choice depends on both software and hardware; shifting abstraction levels is essential.

# Algorithmic paradigms (1/2)



**Phase parallel** Alternate independent compute phases and synchronization.




Phase parallel

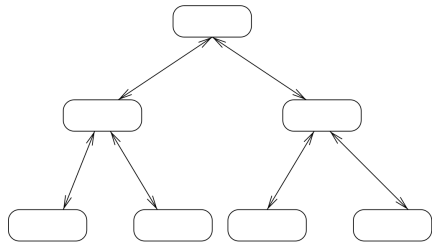
Phase parallel

# Algorithmic paradigms (1/2)



**Phase parallel** Alternate independent compute phases and synchronization.

**Divide and conquer** Recursively split into subproblems; combine results.



Divide and Conquer

Divide and conquer

# Algorithmic paradigms (1/2)



**Phase parallel** Alternate independent compute phases and synchronization.

**Divide and conquer** Recursively split into subproblems; combine results.

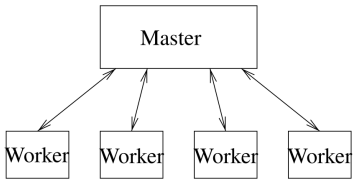
**Owner computes** Partition data; each task processes its own partition.

# Algorithmic paradigms (2/2)



Master–worker A controller distributes work and collects results.

PDE-based simulations often use phase-parallel + owner-computes via domain decomposition. Design optimization may mix master–worker with complex simulation kernels.




Master–Worker

Master–worker

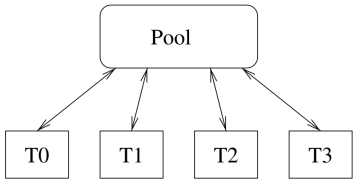
# Algorithmic paradigms (2/2)



**Master–worker** A controller distributes work and collects results.

**Work pool** Tasks pull jobs from a shared queue; may generate new jobs.

PDE-based simulations often use phase-parallel + owner-computes via domain decomposition. Design optimization may mix master–worker with complex simulation kernels.




Work pool

Work pool

# Programming models



**Implicit parallelism** Compiler extracts parallelism; often low efficiency.

**Data parallel** Single control flow applied to partitioned data; logically shared memory (e.g., HPF constructs like FORALL).

**Message passing** Processes interact by explicit messages; SPMD common; natural for owner-computes; very flexible but programmer manages communication.

# Programming models



Shared variable Logically shared memory with multiple control flows and private data; attractive with compiler support via directives.

- Data-parallel HPF largely faded in practice.
- Message passing dominates for scalability to many nodes.
- Shared-memory models are effective, especially with compiler support.



## Overview

This set of tools represents the backbone and provides the actual implementations of the parallel programming models we will be using in this course to make Linear Algebra algorithms run on parallel computers.

**Four main tools:**

| | |
|---|---|
| MPI | Message Passing Interface |
| OpenMP | Open Multi-Processing |
| OpenACC | Open Accelerators |
| CUDA | Compute Unified Device Architecture |

# MPI: Message Passing Interface



## What is MPI?

A standardized and portable message-passing system for process communication in parallel computing environments.

**Key features:**

- Widely used in HPC applications
- Point-to-point and collective communication
- Synchronization primitives
- Data distribution mechanisms
- Natural fit for distributed memory systems

# MPI implementations



- Multiple implementations exist, both open-source and commercial.
- Popular open-source implementations:
  - MPICH: www.mpich.org
  - Open MPI: www.open-mpi.org
- Commercial implementations often optimized for specific hardware, e.g., Cray MPI, Intel MPI, IBM Spectrum MPI.

**Note**

MPI is a specification; different implementations may have varying performance characteristics and features.

**Same code different MPI implementations**

The same MPI code can be compiled and run with different MPI implementations, allowing flexibility and portability across various HPC systems (at least on paper).

# OpenMP: Shared Memory Parallelism



## What is OpenMP?

An API for multi-platform shared memory multiprocessing programming.

**Key features:**

- Compiler directives for parallel regions
- Library routines and environment variables
- Primary use: parallelizing loops and code sections
- Concurrent execution on multiple threads
- Ideal for intra-node parallelism

**Info:** www.openmp.org

# OpenACC: Directive-Based Accelerator Programming



## What is OpenACC?

A directive-based programming tool for heterogeneous systems (CPUs and GPUs).

**Key features:**

- High-level directives for parallelism
- Automatic data movement between host and device
- Easier offloading to accelerators
- Portable across different accelerator architectures

**Info:** www.openacc.org

# CUDA: NVIDIA GPU Programming

4 Paradigms, models and tools for parallel programming

## What is CUDA?

A parallel computing platform and API developed by NVIDIA for general-purpose computing on GPUs.

**Key features:**

- Direct access to GPU parallel processing power
- Low-level control over GPU resources
- Extensive libraries and tools ecosystem
- Specific to NVIDIA GPUs

**Info:** developer.nvidia.com/cuda-zone



## Combining tools

These programming tools are not mutually exclusive and can be used together in a single application.

**Common pattern: MPI+X**

- **MPI** for inter-node communication
- **X** for intra-node parallelism, where X can be:
  - OpenMP (CPU threads)
  - OpenACC (directives for accelerators)
  - CUDA (direct GPU programming)

# MPI+X: The current standard



- Virtually all large-scale HPC libraries and applications are based on the MPI+X framework
- Active research into alternatives to MPI+X exists
- In this course, we focus on MPI+X due to its widespread adoption

**Research opportunity**

Porting the ideas and algorithms we discuss to alternative frameworks could be an interesting avenue of research.

# Summary of Lecture 2

5 Conclusion and summary

Scalability concepts: speed-up, efficiency, Amdahl's and Gustafson's laws

Parallel programming paradigms: phase parallel, divide and conquer, owner computes, master–worker, work pool

Programming models: implicit parallelism, data parallel, message passing, shared variable

Programming tools: MPI, OpenMP, OpenACC, CUDA: the MPI+X framework

**Next up:** Modern memory hierarchy and Roofline model: for performance analysis Measuring memory bandwidth with STREAM benchmark, Intra-node parallelism with OpenMP and starting the implementation of basic linear algebra kernels.

# References

6 Bibliography


[1] M. Flynn. “Very high-speed computing systems”. In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909. DOI: 10.1109/PROC.1966.5273.

[2] M. J. Flynn. “Some Computer Organizations and Their Effectiveness”. In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071.

# High Performance Linear Algebra

## Lecture 3: Intra-node parallelism and starting with BLAS

Ph.D. program in High Performance Scientific Computing

**Fabio Durastante** **Pasqua D'Ambra** **Salvatore Filippone**

November 17, 2025 — 14.00:16.00

Dipartimento
di Matematica
Università di Pisa

# Summary of previous lecture

1 Summary of previous lecture

- Taxonomy of computer architectures
- Performance metrics: FLOP/s, speedup, efficiency, scalability
- Performance modeling: weak and strong scaling

# Table of Contents

2 The roofline model


▶ The roofline model

▶ Intra-node parallelism
Intra-node parallelism: advanced architectures
Intra-node parallelism: tools
OpenMP
Using CMake and doing CI with OpenMP

# Modern Memory Hierarchy



- Computer architectures organized around a **memory hierarchy**
- Designed to balance speed, capacity, and cost

**Memory Hierarchy Levels**

1. Registers and cache (L1, L2, L3) — extremely fast
2. Main memory (RAM) — moderate speed
3. Secondary storage (SSD/HDD) — slower
4. Tertiary storage — archival

**Key parameter:** Memory bandwidth — rate of data transfer between memory and processor



## The Problem

Processor speeds have grown much faster than memory bandwidth improvements

- **Memory wall:** memory latency and bandwidth become the primary bottleneck
- Need tools to understand and visualize this limitation
- Enter: the *Roofline Model* [7]

# The Roofline Model: Concept



**Definition**

A visual performance model relating computational throughput to memory bandwidth

**Key hardware characteristics:**

- Peak floating-point performance: Perf (FLOP/s)
- Peak memory bandwidth: BW (Bytes/s)

**Key application characteristic:**

- Operational Intensity (OI): FLOP/Byte
- Ratio of floating-point ops to bytes accessed from memory

# Roofline Model: The Relationship



**Fundamental equation**

$$\text{Perf} = \frac{\text{FLOP}}{s} = \frac{\text{FLOP}}{\text{Byte}} \cdot \frac{\text{Byte}}{s} = \text{OI} \cdot \text{BW}$$

- Performance depends linearly on both OI and BW
- Plotted as log-log graph: performance vs operational intensity
- Creates a characteristic “roofline” shape

# Roofline Model: Visual Representation



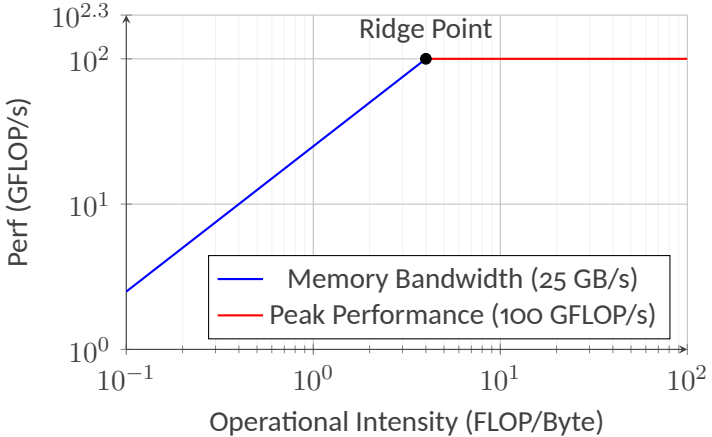



**Figure:** Roofline model: 100 GFLOP/s peak, 25 GB/s bandwidth

# Understanding the Roofline Plot



**Two regions:**

1. Memory-bound (left)
   - Linear increase with OI
   - Limited by bandwidth
2. Compute-bound (right)
   - Horizontal line
   - Limited by peak FLOP/s

**Ridge point:**

- Intersection of two regions
- Minimum OI to reach peak performance
- In example: 4 FLOP/Byte

# Using the Roofline Model



**Applications:**

- Analyze kernel performance on given architecture
- Identify performance bottlenecks
- Guide optimization efforts

**Optimization strategy**

Compare kernel's OI to ridge point:

- Below ridge → memory-bound → improve data locality
- Above ridge → compute-bound → optimize computations

# Roofline and Linear Algebra Evolution



- Algorithmic optimization improves OI and data locality
- Example: Evolution of BLAS (Basic Linear Algebra Subprograms)
  - Level 1: vector operations (low OI)
  - Level 2: matrix-vector operations (medium OI)
  - Level 3: matrix-matrix operations (high OI)
- Higher-level BLAS operations:
  - Reuse data in fast memory
  - Reduce memory traffic
  - Approach compute-bound regime

More details on BLAS in upcoming lectures

# Measuring Memory Bandwidth: STREAM



## STREAM Benchmark [4, 5]

Measures sustainable memory bandwidth (GB/s) for simple vector kernels

**Four kernels:**

| | |
|---|---|
| COPY | Copy vector from one location to another |
| SCALE | Scale vector by constant factor |
| SUM | Add two vectors |
| TRIAD | Scaled vector addition |

- Simple, easy to understand
- Provides reliable bandwidth measure
- Widely used in HPC community

Info: `http://www.cs.virginia.edu/stream/`

# Measuring Memory Bandwidth: Example



Let us run try the STREAM benchmark on your machine:

- Download the STREAM benchmark from `http://www.cs.virginia.edu/stream/`

```
mkdir -p stream && cd stream
wget -r -np -nH --cut-dirs=2 -e robots=off -R "index.html*" \
    https://www.cs.virginia.edu/stream/FTP/Code/
```

- There is a `Makefile` provided; you can compile with `make`
- The standard configuration requires `g77`, but you can edit the `Makefile` to use `gfortran`, or any other compiler you have available:

```
FF = gfortran
FFLAGS = -O2
```

- Run the benchmark by doing: `./stream_f.exe`

# Example output of STREAM benchmark



```
-------------------------------------------------
 Double precision appears to have 16 digits of accuracy
 Assuming 8 bytes per DOUBLE PRECISION word
-------------------------------------------------
 -------------------------------------------------
 STREAM Version $Revision: 5.6 $
 -------------------------------------------------
 Array size =    2000000
 Offset     =          0
 The total memory requirement is   45 MB
 You are running each test  10 times
 --
 The *best* time for each test is used
 *EXCLUDING* the first and last iterations
 -------------------------------------------------
```

# Example output of STREAM benchmark



```
-------------------------------------------------
Printing one line per active thread....
-------------------------------------------------------
Your clock granularity/precision appears to be      1 microseconds
-------------------------------------------------------
Function     Rate (MB/s)  Avg time    Min time  Max time
Copy:       19300.7949      0.0017      0.0017      0.0019
Scale:      16737.4645      0.0019      0.0019      0.0020
Add:        20691.3250      0.0024      0.0023      0.0025
Triad:      19599.5514      0.0025      0.0024      0.0025
-------------------------------------------------------
Solution Validates!
-------------------------------------------------------
```

# How to obtain correct results from STREAM



- Ensure the array size is large enough to exceed cache sizes
- Compile with optimizations enabled (e.g., `-O2` or higher)
- Run multiple iterations and take the best time
- Validate results to ensure correctness

**Note**

Reported bandwidth may vary based on system load, compiler optimizations, and other factors. Always run multiple trials for reliable measurements.

# How to obtain correct results from STREAM: Example



We can extract the right way to perform the test by looking at the size of the level 3 cache of our machine and ensuring that the array size is large enough to exceed it. This number can be found by running the command:

```
lscpu | grep "L3"
```

On my machine, this returns:

```
L3 cache:                              36 MiB (1 instance)
```

So I should set the array size to be larger than 36 MiB. Since each double-precision number takes 8 bytes, I can calculate the minimum number of elements needed:

```
MIN_SIZE=$(echo "36 * 1024 * 1024 / 8" | bc)
echo $MIN_SIZE
```

This gives me 4,718,592 elements. To be safe, I can set the array size to 5,000,000 elements in the STREAM benchmark code before compiling and running it

# How to obtain correct results from STREAM: Modifying the Makefile



## Using `awk`

A nice way to automate the modification of the array size in the STREAM benchmark code is to use `awk` to edit the source file directly from the command line.

```
L3CACHE=$(lscpu | awk -F: '/L3 cache/ {match($2, /[0-9]+/); print
↪  substr($2, RSTART, RLENGTH)}')
MIN_SIZE=$(echo "${L3CACHE} * 1024 * 1024 / 8" | bc)
echo $MIN_SIZE
```

Then, you can modify the `FFLAGS` variable in the Makefile to use the new array size:

```
FFLAGS="-O3 -march=native -mtune=native
↪  -DSTREAM_ARRAY_SIZE=${MIN_SIZE}"
```

# Measuring Peak Performance



## Estimation formula

$$\text{Peak FLOP/s} = \text{Cores} \times \text{Clock (GHz)} \times \text{FLOP/Cycle}$$

**Example:** x86 processor with AVX2

- 8 double-precision FLOP per cycle
- 4 cores at 3 GHz
- Peak: $4 \times 3 \times 8 = 96$ GFLOP/s

### Note

This is theoretical peak; actual performance may be lower due to: bandwidth limitations, cache misses, other overheads. It always best to get this number from the manufacturer datasheet when possible.

# Table of Contents

3 Intra-node parallelism

- The roofline model

- Intra-node parallelism
  - Intra-node parallelism: advanced architectures
  - Intra-node parallelism: tools
  - OpenMP
  - Using CMake and doing CI with OpenMP

# From Moore's Law to Parallelism

3 Intra-node parallelism

- For decades, performance grew via Moore's Law
  - Higher clock frequencies
  - Instruction Level Parallelism (ILP): pipelining, out-of-order execution, branch prediction
- Early 2000s: this trend hit fundamental limits

**Moore's Law**

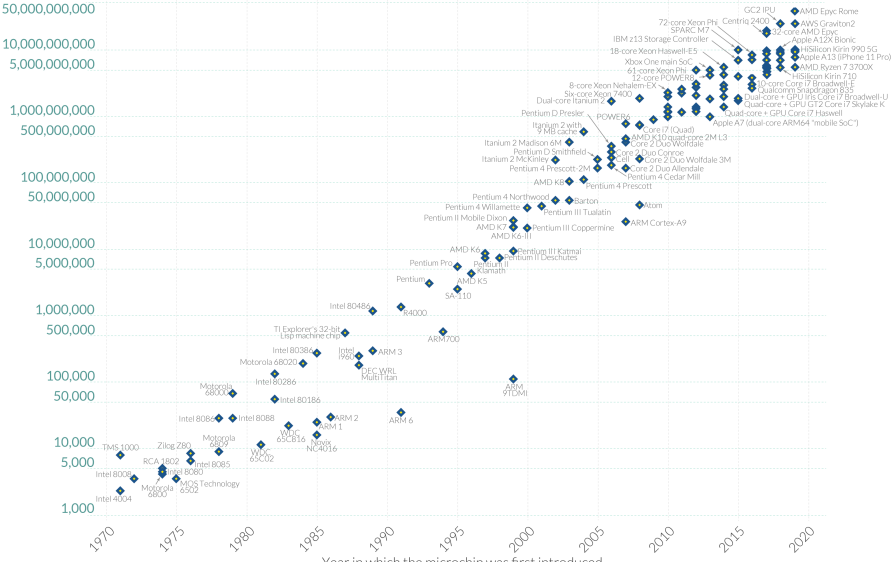
Number of transistors on a microchip doubles approximately every two years, leading to increased computational power and decreased relative cost (Gordon E. Moore, 1965)

# Moore's Law: The number of transistors on microchips has doubled every two years

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years.
This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Our World in Data

## Transistor count

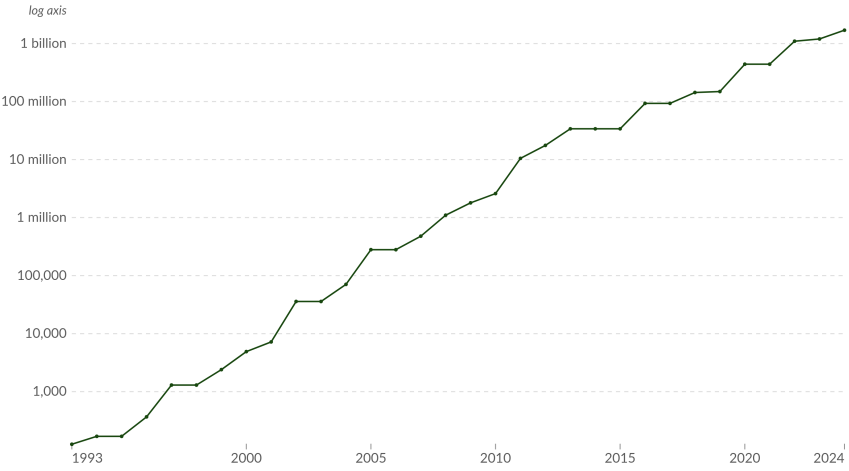



Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

# Computational capacity of the fastest supercomputers

Our World in Data

The number of floating-point operations[1] carried out per second by the fastest supercomputer in any given year. This is expressed in gigaFLOPS, equivalent to $10^9$ floating-point operations per second.




**Data source:** Dongarra et al. (2024)

OurWorldinData.org/technological-change | CC BY

**1. Floating-point operation** A floating-point operation (FLOP) is a type of computer operation. One FLOP represents a single arithmetic operation involving floating-point numbers, such as addition, subtraction, multiplication, or division.

# Hard Limits to ILP



**Concurrence Limit**

- ILP techniques are sophisticated but limited
- Modern processors: max 4-5 instructions per cycle
- Available concurrence is much larger

**Power Limit**

- Power consumption $\propto$ frequency$^3$
- Critical for mobile devices (battery life)
- Critical for supercomputers (operational costs)
- Top500 systems: $\sim$30 MW (small town!)

# The Shift to Thread Level Parallelism (TLP)



- Industry shifted from ILP to TLP techniques
- Birth of **multicores** / **Chip Multi-Processors (CMP)**
- Multiple independent cores on the same die
- Each core handles different instructions and data streams

## Key Advantages

- Higher concurrence levels
- Power consumption $\propto$ number of cores (linear)
- Lower frequency + more cores = better performance + less power

# The Multicore Era

3 Intra-node parallelism

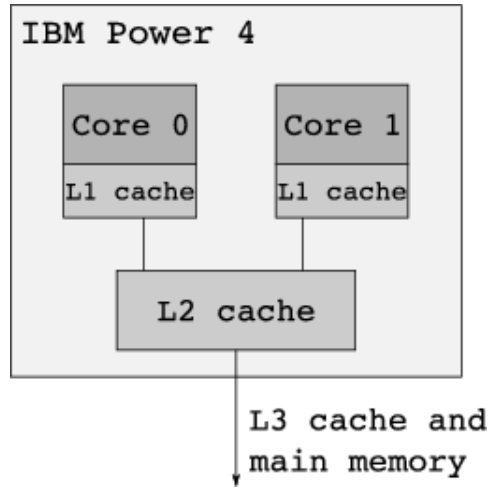
- Multicore processors are now ubiquitous
- Evolution driven by increasing core counts per chip
- Paradigm shift: parallel programming is essential

**Performance no longer comes from faster cores, but from more cores working together**

# POWER4: first mainstream multicore (2001)



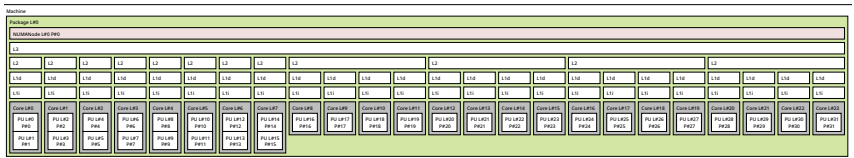
- Two general-purpose cores on the same die
- Per-core private L1 caches
- Shared L2; off-chip shared L3
- Cores access DRAM via a shared memory bus
- Template for many subsequent multicore designs



# Modern CPU examples

## 3 Intra-node parallelism

- AMD EPYC 9655P (2023): 96 cores, 192 threads
- Intel Xeon w9-3595X (2024): 60 cores, 120 threads
- Intel i9-14900HX (hybrid, 24 cores / 32 threads)
  - 8 P-cores (16 threads), each with dedicated L2
  - 16 E-cores (16 threads), L2 shared across clusters of 4
  - L3 shared among all cores; L1 private per core



# CPU cache hierarchy (L1, L2, L3)



- Caches use SRAM (fast, low latency, small, costly)
- DRAM in main memory is larger but slower
- Multi-level design balances speed, capacity, and cost
- L1: smallest/fastest, usually split I/D caches, per-core
- L2: larger/slower than L1, per-core or per-cluster
- L3: largest on-chip, shared across cores
- Miss path: L1 → L2 → L3 → DRAM (increasing latency)

## Exercise: topology

Use `lscpu` and the following command to inspect your CPU topology:

```
lstopo --no-attrs --no-factorize --no-collapse --no-cpukinds --no-legend
↪  topology.pdf
```

# Memory-bound vs compute-bound workloads



**Memory-bound**

- Low arithmetic intensity; little/no data reuse
- Performance limited by memory bandwidth
- Parallel speedups saturate early
- Examples:
  - SpMV: $\mathcal{O}(\text{nnz})$ FLOPs on $\mathcal{O}(\text{nnz})$ data
  - BLAS-1: $\mathcal{O}(n)$ FLOPs on $\mathcal{O}(n)$ data
  - BLAS-2: $\mathcal{O}(n^2)$ FLOPs on $\mathcal{O}(n^2)$ data

**Compute-bound**

- High arithmetic intensity; strong data reuse
- Performance limited by peak FLOP/s
- Scales well across cores (cache-friendly)
- Examples:
  - BLAS-3 (e.g., GEMM): $\mathcal{O}(n^3)$ FLOPs on $\mathcal{O}(n^2)$ data
  - Dense factorizations leveraging BLAS-3

# Operational intensity and the memory wall



- Example (EPYC 9655P): peak 710 GFLOP/s vs 614 GB/s bandwidth
- Roofline knee: 710/614 ≈ 1.16 FLOP/byte
  - OI < 1.16: memory-bound (bandwidth limits performance)
  - OI > 1.16: compute-bound (FLOP/s limits performance)
- As core counts grow, static bandwidth limits memory-heavy kernels
- Remedies: improve cache reuse, increase bandwidth, or both

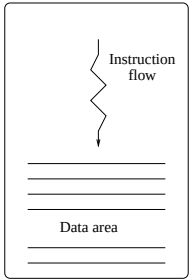
# Multiprogramming and processes



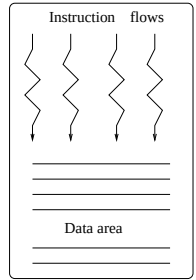
- Modern systems support multiprogramming: many programs appear to run concurrently.
- Microscopic view: you **cannot** execute **more programs than available cores**.
- Macroscopic view: time sharing makes many programs seem concurrent.
- A process is a running *instance of a program* **plus** its *data*.
- Processes are dynamic; multiple processes can run the same program.
- **Each process has a private address space** (its data are private).

# Processes and threads: visual






- Code + private memory + execution context.
- OS schedules processes on cores.
- No shared memory by default.




- Execution streams within a process.
- Share address space and program data.
- Own stack and registers; often one per core.

# Programming tools for threads



- POSIX threads (pthreads): low-level API, fine-grained control, portable.
- OpenMP: high-level, directive-based, widely used in C/C++/Fortran.
- Typical workflow: start with OpenMP; use pthreads only when necessary.

Will start describing some **OpenMP basics**, and decline it in the context of linear algebra routines.

# OpenMP: overview



- De-facto standard API for shared-memory parallel programming.
- Languages: Fortran, C, C++; introduced in 1997.
- Maintained by the OpenMP Architecture Review Board (openmp.org).

# OpenMP: overview



- De-facto standard API for shared-memory parallel programming.
- Languages: Fortran, C, C++; introduced in 1997.
- Maintained by the OpenMP Architecture Review Board (openmp.org).

Components:

- Compiler directives (pragmas)
- Run-time library routines
- Environment variables

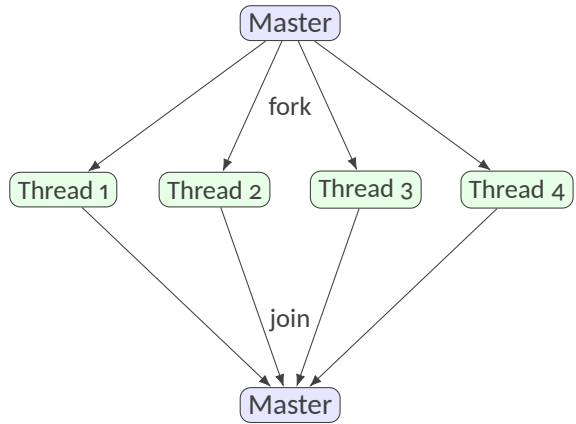
Directives behave as:

1. Actual instructions for OpenMP-aware compilers
2. Comments for non-supporting compilers (keeps serial behavior)

# Fork-join execution model



- Serial region executed by a single master thread.
- Hitting a parallel region: fork into multiple threads.
- Threads share address space; may coordinate via shared data.
- End of region: threads join back to one thread.



# Original focus: loop parallelism



- Split workload of loops (e.g., `do`) across threads.

1. Enter a loop/region: activate multiple threads and partition iterations.
2. Threads may communicate via shared variables/memory.
3. On completion: synchronize; deactivate all but one thread and continue serially.

# Original focus: loop parallelism



- Split workload of loops (e.g., `do`) across threads.

1. Enter a loop/region: activate multiple threads and partition iterations.
2. Threads may communicate via shared variables/memory.
3. On completion: synchronize; deactivate all but one thread and continue serially.

- Programming model: threads with shared logical address space.
- Natural fit for shared-memory systems; not mandated by the standard.
- Attempts to map the same model to distributed-memory exist, but limited success in practice.

# OpenMP today



- Standard evolves regularly; 6.0 recently released, 5.2 widely supported.
- Key additions:
  - Irregular and data-driven workload dispatching
  - Transformations to improve memory hierarchy usage and work sharing
  - Support for SIMD extensions and accelerators

## OpenMP in practice

- Will show concrete OpenMP code next.
- Often combined with MPI for hybrid/nested parallelism.
- Further reading: [1, 2, 3, 6]

# OpenMP example: let us start from an hello world



The standard Fortran hello world program:

```
program hello
    use, intrinsic :: iso_fortran_env,
    ↪  only: output_unit
    write (output_unit, '("Hello,
    ↪  world!")')
end program hello
```

which can be compiled and run as:

```
gfortran -o hello hello.f90
./hello
```

Getting the output:

```
Hello, world!
```

# OpenMP example: let us start from an hello world



The standard Fortran hello world program:

```
program hello
    use, intrinsic :: iso_fortran_env,
    ↪  only: output_unit
    write (output_unit, '("Hello,
    ↪  world!")')
end program hello
```

which can be compiled and run as:

```
gfortran -o hello hello.f90
./hello
```

Getting the output:

```
Hello, world!
```

We now want to implement the same program using OpenMP, and getting an output from each thread.

```
program hello
    use, intrinsic :: iso_fortran_env,
    ↪  only: output_unit
    use omp_lib
    integer :: tid, nthreads
    nthreads = omp_get_max_threads()
    !$omp parallel private(tid)
    tid = omp_get_thread_num()
    write (output_unit, '("Hello, world!
    ↪  from thread ", I0)') tid
    !$omp end parallel
end program hello
```

# Compiling the OpenMP hello world



To compile the OpenMP version, we need to add the '-fopenmp' flag:

```
gfortran -o hello hello.f90 -fopenmp
./hello
```

Getting the output (on my Laptop):

```
Hello, world! from thread 3
Hello, world! from thread 20
Hello, world! from thread 31
Hello, world! from thread 1
Hello, world! from thread 2
Hello, world! from thread 5
Hello, world! from thread 7
⋮
Hello, world! from thread 29
Hello, world! from thread 16
```

- Each thread prints its ID.
- Order of output may vary due to thread scheduling.
- By default, uses all available threads.
- Control number of threads via `OMP_NUM_THREADS=<num>` environment variable.

# Compiling the OpenMP hello world



To compile the OpenMP version, we need to add the '-fopenmp' flag:

```
gfortran -o hello hello.f90 -fopenmp
./hello
```

Getting the output (on my Laptop):

```
Hello, world! from thread 3
Hello, world! from thread 20
Hello, world! from thread 31
Hello, world! from thread 1
Hello, world! from thread 2
Hello, world! from thread 5
Hello, world! from thread 7
⋮
Hello, world! from thread 29
Hello, world! from thread 16
```

- Each thread prints its ID.
- Order of output may vary due to thread scheduling.
- By default, uses all available threads.
- Control number of threads via `OMP_NUM_THREADS=<num>` environment variable.
- Let us have a better look at the code, line by line.

# OpenMP hello world: code walkthrough



```
program hello
    use, intrinsic ::
    ↪  iso_fortran_env, only:
    ↪  output_unit
    use omp_lib
    integer :: tid, nthreads
    nthreads =
    ↪  omp_get_max_threads()
    !$omp parallel private(tid)
    tid = omp_get_thread_num()
    write (output_unit, '("Hello,
    ↪  world! from thread ",
    ↪  I0)') tid
    !$omp end parallel
end program hello
```

- `use omp_lib`: imports OpenMP functions/constants
- `nthreads = omp_get_max_threads()`: gets max available threads
- *`!$omp parallel private(tid)`*: starts parallel region; each thread has private `tid`
- `tid = omp_get_thread_num()`: each thread gets its unique ID
- *`!$omp end parallel`*: ends parallel region; threads synchronize

# Compilation flag for other compilers



- **GCC / GFortran**: `-fopenmp`
- **Intel ICC / IFORT**: `-qopenmp` or `-openmp`
- **Clang / Flang**: `-fopenmp` (requires OpenMP library)
- **PGI / NVIDIA HPC SDK**: `-mp`

**Note**

Ensure the compiler supports OpenMP and is properly configured.

## Mixing compilers

There exist a few cases where mixing compilers is possible (e.g., Intel and GCC), but in general it is not recommended to mix different compilers when dealing with OpenMP code.

# Using CMake to build a Fortran project



As we have seen from the previous slide, and from the question on managing different compilers in the previous lecture, it is often useful to use a **build system** to manage the complexity of building a project.

# Using CMake to build a Fortran project



As we have seen from the previous slide, and from the question on managing different compilers in the previous lecture, it is often useful to use a **build system** to manage the complexity of building a project.
There exists several build systems:

Make / GNU Make / Autotools: classic, widely used, but low-level
https://www.gnu.org/software/make/

CMake: popular, cross-platform, higher-level
https://cmake.org/

Ninja: fast, modern, often used as a backend for CMake
https://ninja-build.org/

Meson: high-level, fast, modern
https://mesonbuild.com/

# Using CMake to build a Fortran project



To build a project with CMake, the first step is represented by the creation of a `CMakeLists.txt` file in the root directory of the project.

# Using CMake to build a Fortran project



To build a project with CMake, the first step is represented by the creation of a `CMakeLists.txt` file in the root directory of the project.
Let us go step by step through a minimal example.

1. Create a folder for the project and enter it:

```
mkdir hello_openmp
cd hello_openmp
```

# Using CMake to build a Fortran project



To build a project with CMake, the first step is represented by the creation of a `CMakeLists.txt` file in the root directory of the project.
Let us go step by step through a minimal example.

1. Create a folder for the project and enter it:

```
mkdir hello_openmp
cd hello_openmp
```

2. Create a git repository inside:

```
git init
git branch -m main
```

# Using CMake to build a Fortran project



To build a project with CMake, the first step is represented by the creation of a `CMakeLists.txt` file in the root directory of the project.
Let us go step by step through a minimal example.

1. Create a folder for the project and enter it:
   ```
   mkdir hello_openmp
   cd hello_openmp
   ```
2. Create a git repository inside:
   ```
   git init
   git branch -m main
   ```
3. Create the Fortran source file `hello.f90` with the OpenMP code seen before.

# Using CMake to build a Fortran project



To build a project with CMake, the first step is represented by the creation of a `CMakeLists.txt` file in the root directory of the project.
Let us go step by step through a minimal example.

1. Create a folder for the project and enter it:
   ```
   mkdir hello_openmp
   cd hello_openmp
   ```
2. Create a git repository inside:
   ```
   git init
   git branch -m main
   ```
3. Create the Fortran source file `hello.f90` with the OpenMP code seen before.
4. Create the `CMakeLists.txt` file:
   ```
   touch CMakeLists.txt
   ```

# Editing the CMakeLists.txt file



The content of the `CMakeLists.txt` file should be as follows:

```
cmake_minimum_required(VERSION 3.23)

project(hello-openmp LANGUAGES Fortran)
find_package(OpenMP REQUIRED COMPONENTS
↪  Fortran)

# Executable from the single source file
add_executable(hello-openmp hello-openmp.f90)
# Link OpenMP
target_link_libraries(hello-openmp PRIVATE
↪  OpenMP::OpenMP_Fortran)
```

- Specify minimum CMake version
- Define project name and language
- Find OpenMP package for Fortran
- Add executable target
- Link OpenMP libraries to the target

# The CMake instructions explained



The roject name and the programming language used, it also take further optional arguments:

```
project(<PROJECT-NAME>
        [VERSION
        ↪  <major>[.<minor>[.<patch>[.<tweak>]]]]
        [COMPAT_VERSION
        ↪  <major>[.<minor>[.<patch>[.<tweak>]]]]
        [SPDX_LICENSE <license-string>]
        [DESCRIPTION <description-string>]
        [HOMEPAGE_URL <url-string>]
        [LANGUAGES <language-name>...])
```

Specify:

- project name
- version
- compatible version
- license (SPDX format)
- description
- homepage URL
- programming languages used

# The CMake instructions explained



Another important command is igure external packages or libraries that the project depends on.

```
find_package(<PackageName> [version] [EXACT]
↪   [REQUIRED]
              [QUIET] [COMPONENTS components...]
              [OPTIONAL_COMPONENTS components...]
              [NO_DEFAULT_PATH])
```

Specify:

- package name
- version
- whether it is required
- components to find
- whether to suppress messages
- whether to avoid default search paths

# The CMake instructions explained



Another important command is igure external packages or libraries that the project depends on.

```
find_package(<PackageName> [version] [EXACT]
↪  [REQUIRED]
             [QUIET] [COMPONENTS components...]
             [OPTIONAL_COMPONENTS components...]
             [NO_DEFAULT_PATH])
```

You can pass suggestion on where to find the package using the `CMAKE_PREFIX_PATH` environment variable or the `-DCMAKE_PREFIX_PATH=<path>` option when invoking CMake.

Specify:

- package name
- version
- whether it is required
- components to find
- whether to suppress messages
- whether to avoid default search paths

# The CMake instructions explained



The next command is `add_executable()`, which is used to define an executable target:

```
add_executable(<name> [WIN32] [MACOSX_BUNDLE]
               [EXCLUDE_FROM_ALL]
               source1 source2 ... sourceN)
```

An executable target is a binary file that can be run on the system, it can be created from *one* or *more* source files.

Specify:

- target name
- platform-specific options
- whether to exclude from default build
- source files

The last command is `target_link_libraries()`, which is used to specify libraries to link against a target.

```
target_link_libraries(<target>
               <PRIVATE|PUBLIC|INTERFACE> <item>...
               [<PRIVATE|PUBLIC|INTERFACE> <item>...]...)
```

Specify:

- target name
- libraries to link
- linkage type

# Private, Public, and Interface linkage



When using `target_link_libraries()`, you can specify the linkage type:

- **PRIVATE**: the library is used only for the target itself.
- **PUBLIC**: the library is used for both the target and any targets that link against it.
- **INTERFACE**: the library is used only for targets that link against the target, not for the target itself.

**Example**

```
target_link_libraries(my_executable
    PRIVATE libA
    PUBLIC libB
    INTERFACE libC)
```

In this example, `libA` is linked only to `my_executable`, `libB` is linked to both `my_executable` and any targets that link against it, and `libC` is linked only to targets that link against `my_executable`.

# Configuring and building



To configure and build the project with CMake the steps are:

1. Create a build folder: `mkdir build`
2. Move to the build folder and launch the `cmake` program

```
cd build
cmake .. # You could also try doing ccmake .. for an interactive configuration
```

3. Build the project using the generated build system, for example:

   Make run `make`
   Ninja run `ninja`

   this will compile the code and generate the executable in the `build` folder.

# Make a commit



If everything works, we can make a commit of the results.

- it is a good idea to create a `.gitignore` file to avoid committing build *artifacts*.

For doing this, you run

```
touch .gitignore
```

and then with your favourite editor write inside it

```
build/
```

Everything which is listed here is going to be ignored by git.

# Make a commit



If everything works, we can make a commit of the results.

- it is a good idea to create a `.gitignore` file to avoid committing build *artifacts*.

For doing this, you run

```
touch .gitignore
```

and then with your favourite editor write inside it

```
build/
```

Everything which is listed here is going to be ignored by git. Now we can add all the files and make a commit:

```
git add .
git commit -m "Initial commit: OpenMP hello world with CMake"
```

# Continuous Integration (CI) with GitHub Actions



We can adapt our last example of continuous integration (CI) with GitHub Actions from the previous lecture to build and test our OpenMP project. We need to create a workflow file in the `.github/workflows` folder.

1. Create the folders:
   ```
   mkdir -p .github/workflows
   ```
2. Create the workflow file:
   ```
   touch .github/workflows/CI.yml
   ```
3. Edit the file (starting from the one seen in the previous lecture).

# Editing the CI.yml file



The content of the `CI.yml` file should be as follows:

```yaml
name: CI
on:
  push:
    branches:
      - main
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4
      - name: Setup CMake (latest)
        uses: lukka/get-cmake@latest
      - name: Setup Fortran
        uses: fortran-lang/setup-fortran@v1.8.0
```

- Define workflow name and trigger on push to main branch
- Set up Ubuntu environment
- Checkout code, **set up CMake** and Fortran compiler

```yaml
    with:
      compiler: gcc
      version: 'latest'
      update-environment: true
```

# Editing the CI.yml file



The content of the `CI.yml` file should be as follows:

```
- name: Configure (CMake)
  run: cmake -S . -B build
  ↪  -DCMAKE_BUILD_TYPE=Release
- name: Build (CMake)
  run: cmake --build build --config Release
  ↪  -- -j
- name: Run program
  env:
    OMP_NUM_THREADS: '4'
  run: |
    ./build/hello-openmp || (echo
    ↪  "Executable not found" && ls -la
    ↪  build && exit 1)
```

- **Configure** and **build** project using CMake
- Run the compiled OpenMP program with 4 threads

# Summary and Next Steps

4 Summary

- OpenMP is a widely used API for shared-memory parallel programming.
- It provides a simple and flexible way to parallelize code using compiler directives.
- CMake can be used to manage the build process of Fortran projects with OpenMP.

# Summary and Next Steps

4 Summary

- OpenMP is a widely used API for shared-memory parallel programming.
- It provides a simple and flexible way to parallelize code using compiler directives.
- CMake can be used to manage the build process of Fortran projects with OpenMP.

## Next Steps

- The Basic Linear Algebra Subprograms (BLAS) provide standardized building blocks for dense linear algebra operations.
- Using BLAS enables code reuse, portability, and performance optimizations across different hardware architectures.
- Explore more advanced OpenMP features (e.g., task parallelism, SIMD).
- Use Fortran and OpenMP features to look through BLAS implementations.

# References

5 Bibliography


[1] G. J. Barbara Chapman and R. van der Pas. *Using OpenMP*. MIT Press, Cambridge, MA, 2007, p. 384. ISBN: 9780262533027.

[2] O. A. R. Board. *OpenMP Application Programming Interface Specification 5.2*. Ed. by B. de Supinski and M. Klemm. 2021. ISBN: 979-8497370195. URL: https://www.openmp.org/specifications/.

[3] O. A. R. Board. *OpenMP Application Programming Interface Specification 6.0*. 2024. URL: https://www.openmp.org/specifications/.

[4] J. D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers". In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), pp. 19–25.

# References

5 Bibliography


[5] J. D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Tech. rep. A continually updated technical report. http://www.cs.virginia.edu/stream/. Charlottesville, Virginia: University of Virginia, 1991-2007. URL: http://www.cs.virginia.edu/stream/.

[6] Y. ( H. Timothy G. Mattson and A. E. Koniges. *The OpenMP Common Core*. MIT Press, Cambridge, MA, 2019, p. 320. ISBN: 9780262538862.

[7] S. Williams, A. Waterman, and D. Patterson. “Roofline: an insightful visual performance model for multicore architectures”. In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. URL: https://doi.org/10.1145/1498765.1498785.

# High Performance Linear Algebra

## Lecture 4: Starting with BLAS, BLAS Level 1: AXPY

Ph.D. program in High Performance Scientific Computing

**Fabio Durastante** **Pasqua D'Ambra** **Salvatore Filippone**

November 19, 2025 — 14.00:16.00

Dipartimento
di Matematica
Università di Pisa

# Table of Contents

1 Building Blocks for Dense Linear Algebra


- ▶ Building Blocks for Dense Linear Algebra
- ▶ The Basic Linear Algebra Subprograms (BLAS)
- ▶ Level 1 BLAS: Vector operations
  - AXPY
  - An object oriented packaging
  - Implementation with OpenMP
    - Performance Analysis: roofline model
    - Performance Analyis: varying the number of threads

# Motivation: Cholesky factorization example



## Symmetric Matrix

A matrix $A \in \mathbb{R}^{n \times n}$ is called symmetric if $A = A^\top$, meaning that it is equal to its transpose.

## Eigenvalue and Eigenvector

Given a square matrix $A \in \mathbb{R}^{n \times n}$, a non-zero vector $\mathbf{v} \in \mathbb{R}^n$ is called an eigenvector of $A$ if there exists a scalar $\lambda \in \mathbb{R}$ such that:

$$A\mathbf{v} = \lambda\mathbf{v}$$

The scalar $\lambda$ is referred to as the eigenvalue corresponding to the eigenvector $\mathbf{v}$. All eigenvalues of a symmetric matrix are real.

# Motivation: Cholesky factorization example



## Positive Definite Matrix

A symmetric matrix $A \in \mathbb{R}^{n \times n}$ is called positive definite if for all non-zero vectors $\mathbf{x} \in \mathbb{R}^n$:

$$\mathbf{x}^\top A \mathbf{x} > 0$$

This implies that all eigenvalues of $A$ are positive.

### Examples of symmetric positive definite matrices

- Covariance/correlation matrices in statistics and machine learning.
- Normal equations: $A^\top A$ from least squares; SPD if $A$ has full column rank.
- Gram/kernel matrices: $K_{ij} = k(x_i, x_j)$ with strictly PD kernels (e.g., Gaussian/RBF).
- Precision (inverse covariance) matrices in Gaussian Markov random fields.

# Motivation: Cholesky factorization example



- The Cholesky factorization is a method for decomposing a positive definite matrix $A$ into the product of an upper triangular matrix $U$ and its transpose:

$$A = U^\top U$$

- It is useful for solving systems of linear equations, and inverting matrices.
- It is computationally efficient, requiring approximately $\frac{1}{3}n^3$ operations for an $n \times n$ matrix.

**Theorem (Existence and uniqueness)**

Every symmetric positive definite matrix $A$ has a unique Cholesky factorization $A = U^\top U$, where $U$ is an upper triangular matrix with positive diagonal entries.

# Motivation: Cholesky factorization example



Consider the Cholesky factorization $A = U^\top U$:

**Algorithm**

1: **for** $j = 1$ to $n$ **do**
2: **for** $i = 1$ to $j - 1$ **do**
3: $u_{ij} \leftarrow \frac{1}{u_{ii}} \left( a_{ij} - \sum_{k=1}^{i-1} u_{ki} u_{kj} \right)$
4: **end for**
5: $u_{jj} \leftarrow \sqrt{a_{jj} - \sum_{k=1}^{j-1} u_{kj}^2}$
6: **end for**

- Easy to translate to any language
- But…“reinventing the wheel”
- Similar patterns appear repeatedly
- Lots of code duplication

# The key observation



Similar code patterns resurface over and over again
in linear algebra algorithms

**Natural strategy**

*"Define a set of operators such that any algorithm
can be expressed as their application to the data at hand."*

- Some languages provide native operators (MATLAB, Fortran, Julia)
- Algorithms = sequences of primitive operator calls

# Benefits of standardized building blocks



1. **Code reuse**
   - Write once, use many times
   - Amortize cost of high-quality implementation
2. **Standardized interfaces**
   - Explore alternative implementations
   - Preserve overall code behavior
3. **Architecture-aware optimizations**
   - Exploit cache hierarchies
   - Use block/submatrix operations (not just vectors)
4. **Portability across systems**
   - Same interface, optimized per platform

# Scope of application



- Cholesky is just one example
- Same reasoning applies to:
  - Dense linear algebra (LU, QR, eigensolvers, ...)
  - Sparse linear algebra (SpMV, iterative solvers, ...)
  - Many other numerical algorithms

- Encapsulation enables:
  - Performance tuning without changing user code
  - Leveraging hardware accelerators (GPUs, vector units)
  - Evolution of implementations over time

This is the foundation of BLAS and LAPACK

# Table of Contents

2 The Basic Linear Algebra Subprograms (BLAS)


- ▶ Building Blocks for Dense Linear Algebra
- ▶ The Basic Linear Algebra Subprograms (BLAS)
- ▶ Level 1 BLAS: Vector operations
  - AXPY
  - An object oriented packaging
  - Implementation with OpenMP
    - Performance Analysis: roofline model
    - Performance Analyis: varying the number of threads

# The Basic Linear Algebra Subprograms (BLAS)



- Set of low-level routines for common linear algebra operations
- Designed to be efficient and portable
- Building block for higher-level libraries (LAPACK, ScaLAPACK, PSBLAS, PETSc)
- Available in many programming languages (C, Fortran, Python)

**Focus of this section**

Dense BLAS: routines for dense matrices and vectors

# BLAS organization: three levels



**Level 1:** Vector operations

- Examples: dot product, vector addition, scaling
- Complexity: $\mathcal{O}(n)$
- Memory-bound

**Level 2:** Matrix-vector operations

- Examples: matrix-vector multiplication, rank-1 updates
- Complexity: $\mathcal{O}(n^2)$
- Memory-bound

**Level 3:** Matrix-matrix operations

- Examples: matrix-matrix multiplication (GEMM)
- Complexity: $\mathcal{O}(n^3)$
- Compute-bound (high data reuse)

# Popular BLAS implementations



| | |
|---|---|
| OpenBLAS: | Open-source implementation of BLAS and LAPACK |
| ATLAS: | Automatically Tuned Linear Algebra Software; open-source, self-optimizing |
| Intel MKL: | High-performance library optimized for Intel processors |
| cuBLAS: | GPU-accelerated BLAS for NVIDIA GPUs |
| BLIS: | Portable, high-performance, modern BLAS framework |

**Key takeaway**

Same interface, different implementations $\Rightarrow$ performance portability

# Finding BLAS with CMake



- CMake provides a built-in module to find BLAS libraries
- Use `find_package(BLAS REQUIRED)` to locate BLAS
- Link against the found BLAS library using `target_link_libraries(<target> PRIVATE ${BLAS_LIBRARIES})`
- Information are available on the webpage: FindBLAS module documentation.

**Example CMake snippet**

```
find_package(BLAS REQUIRED)
target_link_libraries(<target> PRIVATE ${BLAS_LIBRARIES})
```

# Table of Contents

3 Level 1 BLAS: Vector operations


- Building Blocks for Dense Linear Algebra
- The Basic Linear Algebra Subprograms (BLAS)
- Level 1 BLAS: Vector operations
  - AXPY
  - An object oriented packaging
  - Implementation with OpenMP
    - Performance Analysis: roofline model
    - Performance Analyis: varying the number of threads

# Level 1 BLAS: Overview

## 3 Level 1 BLAS: Vector operations

| types | name | ( size arguments ) | description | equation | flops | data |
|---|---|---|---|---|---|---|
| s, d, c, z | axpy | ( n, alpha, x, incx, y, incy ) | update vector | $y = y + \alpha x$ | $2n$ | $2n$ |
| s, d, c, z, cs, zd | scal | ( n, alpha, x, incx ) | scale vector | $y = \alpha y$ | $n$ | $n$ |
| s, d, c, z | copy | ( n, x, incx, y, incy ) | copy vector | $y = x$ | 0 | $2n$ |
| s, d, c, z | swap | ( n, x, incx, y, incy ) | swap vectors | $x \leftrightarrow y$ | 0 | $2n$ |
| s, d | dot | ( n, x, incx, y, incy ) | dot product | $= x^T y$ | $2n$ | $2n$ |
| c, z | dotu | ( n, x, incx, y, incy ) | (complex) | $= x^T y$ | $2n$ | $2n$ |
| c, z | dotc | ( n, x, incx, y, incy ) | (complex conj) | $= x^H y$ | $2n$ | $2n$ |
| sds, ds | dot | ( n, x, incx, y, incy ) | (internally double precision) | $= x^T y$ | $2n$ | $2n$ |
| s, d, sc, dz | nrm2 | ( n, x, incx ) | 2-norm | $= \|x\|_2$ | $2n$ | $n$ |
| s, d, sc, dz | asum | ( n, x, incx ) | 1-norm | $= \|\mathrm{Re}(x)\|_1 + \|\mathrm{Im}(x)\|_1$ | $n$ | $n$ |
| s, d, c, z | i_amax | ( n, x, incx ) | ∞-norm | $= \mathrm{argmax}_i( \lvert\mathrm{Re}(x_i)\rvert + \lvert\mathrm{Im}(x_i)\rvert )$ | $n$ | $n$ |
| s, d, c, z | rotg | ( a, b, c, s ) | generate plane (Given's) rotation (c real, s complex) | | $O(1)$ | $O(1)$ |
| s, d, c, z † | rot | ( n, x, incx, y, incy, c, s ) | apply plane rotation (c real, s complex) | | $6n$ | $2n$ |
| cs, zd | rot | ( n, x, incx, y, incy, c, s ) | apply plane rotation (c & s real) | | $6n$ | $2n$ |
| s, d | rotmg | ( d1, d2, a, b, param ) | generate modified plane rotation | | $O(1)$ | $O(1)$ |
| s, d | rotm | ( n, x, incx, y, incy, param ) | apply modified plane rotation | | $6n$ | $2n$ |

# Level 1 BLAS: Vector operations



- Basic operations on vectors
- Examples:
  - Dot product: `DOT`
  - Vector addition: `AXPY`
  - Scaling: `SCAL`
  - Copy: `COPY`
  - Norms: `NRM2`
- **Memory-bound** operations

**Data types:**

- s: single real
- d: double real
- c: single complex
- z: double complex

Naming convention: `<data type><operation>`

# AXPY: Definition



**AXPY** (Add X times Y):

$$y \leftarrow \alpha x + y$$

- $\alpha$ scalar, $x, y$ vectors
- Level 1 BLAS (memory-bound)
- The output vector $y$ is overwritten
- Widely used in numerical algorithms (e.g., iterative methods)

Routine name: `daxpy` (double precision)

```
call daxpy(n, alpha, x, incx, y, incy)
```

- n: vector length
- alpha: scalar
- x, y: vectors
- incx, incy: strides (usually 1)

# Fortran example (double precision)



```
program axpy_example
    use iso_fortran_env, only: int64, real64, output_unit
    implicit none
    integer(kind=int64), parameter :: n = 5
    real(kind=real64) :: x(n), y(n), alpha
    integer(kind=int64) :: i
    ! Initialize the vectors and scalar
    x = [1.0, 2.0, 3.0, 4.0, 5.0]
    y = [10.0, 20.0, 30.0, 40.0, 50.0]
    alpha = 2.0
    ! Call the AXPY routine
    call daxpy(n, alpha, x, 1, y, 1)
    ! Print the result
    write(output_unit, '("Resulting vector y:")')
    do i = 1, n
        write(output_unit, '(F6.2)', advance='no') y(i)
    end do
    write(output_unit, '("")')
    return
end program axpy_example
```

# Compiling (OpenBLAS)



Install (Ubuntu): `apt-get install libopenblas-dev`

```
gfortran -o axpy_example axpy_example.f90 -lopenblas
./axpy_example
```

**Sample output**

```
Resulting vector y:
 12.00 24.00 36.00 48.00 60.00
```

# Compiling (OpenBLAS)



Install (Ubuntu): `apt-get install libopenblas-dev`

```
gfortran -o axpy_example axpy_example.f90 -lopenblas
./axpy_example
```

**Sample output**

```
Resulting vector y:
 12.00 24.00 36.00 48.00 60.00
```

There are quite a few inconvenient things!

# An object oriented packaging



Let us make an **object oriented packaging** of this BLAS operations using modern Fortran.

- Create a Git repository for our package

```
mkdir objblas
cd objblas
git init
git branch -m main
```

# An object oriented packaging



Let us make an **object oriented packaging** of this BLAS operations using modern Fortran.

- Create a Git repository for our package

```
mkdir objblas
cd objblas
git init
git branch -m main
```

- Create a `CMakeLists.txt` file for our project with content

```
cmake_minimum_required(3.28)
project(objblas LANGUAGES Fortran)
find_package(BLAS REQUIRED)
```

# An object oriented packaging



Let us make an **object oriented packaging** of this BLAS operations using modern Fortran.

- Create a Git repository for our package

```
mkdir objblas
cd objblas
git init
git branch -m main
```

- Create a `CMakeLists.txt` file for our project with content

```
cmake_minimum_required(3.28)
project(objblas LANGUAGES Fortran)
find_package(BLAS REQUIRED)
```

- Create a directory which will contain the code:

```
mkdir src
touch src/blas.f90
```

# An object oriented packaging



We will now create a Fortran module, which will package the BLAS library we are going to use, hence we write into the `src/blas.f90` file the following:

```
module blas
    use iso_fortran_env, only: real64, real32
    implicit none

     < interfaces >

contains

     < implementations >

end module blas
```

# Let us start with the implementations

## 3 Level 1 BLAS: Vector operations

```
subroutine daxpy_blas(alpha, x, y, incx, incy)
    use iso_fortran_env, only: real64
    implicit none
    real(real64), intent(in) :: alpha
    real(real64), intent(in) :: x(:)
    real(real64), intent(inout) :: y(:)
    integer, intent(in), optional :: incx, incy
    ! Local variables
    integer :: incx_, incy_
    incx_ = 1
    incy_ = 1
    if (present(incx)) incx_ = incx
    if (present(incy)) incy_ = incy
    call daxpy(size(x),alpha,x,incx_,y,incy_)
end subroutine daxpy_blas
```

- `intent()` tells the subroutine if the argumenti is an input, an output, or both,
- `optional` tells if the argument can be omitted, and `present` checks if it has been passed or not.
- We use `size(x)` to get the length of the vector.
- We can write similar subroutines for the other data types (single, complex, double complex).

# Now the interfaces

3 Level 1 BLAS: Vector operations

```
private

interface axpy
    module procedure daxpy_blas
    ! Other data types procedures
end interface axpy

public :: axpy
```

- `private` makes all the module contents private by default,
- `public :: axpy` makes the `axpy` interface public.

- The `interface` block allows to define multiple procedures with the same name but different argument types.
- Here we define the interface for `daxpy`, which maps to the implementation `daxpy_blas`.
- We can add other procedures for different data types (single, complex, double complex).

# CMake configuration



We need to tell CMake how to build our package, so we add the following lines to the `CMakeLists.txt` file:

```
add_library(objblas src/blas.f90)
target_link_libraries(objblas PUBLIC BLAS::BLAS)
```

- `add_library()` creates a library target named `objblas` from the source file.
- `target_link_libraries()` links the BLAS libraries to our package.

# CMake configuration



We need to tell CMake how to build our package, so we add the following lines to the `CMakeLists.txt` file:

```
add_library(objblas src/blas.f90)
target_link_libraries(objblas PUBLIC BLAS::BLAS)
```

- `add_library()` creates a library target named `objblas` from the source file.
- `target_link_libraries()` links the BLAS libraries to our package.

Now we need to write a tester: we create a `test` directory and inside it a `CMakeLists.txt` file and a `axpy_test.f90` file and add the following lines to the `test/CMakeLists.txt` file:

```
add_executable(test_axpy test/test_axpy.f90)
target_link_libraries(test_axpy PRIVATE objblas)
```

# Test program

## 3 Level 1 BLAS: Vector operations

```
program test_axpy
    use iso_fortran_env, only: real64,
    ↪  output_unit
    use blas
    implicit none
    integer, parameter :: n = 10
    real(real64) :: x64(n), y64(n),
    ↪  alpha64
    x64 = [1,2,3,4,5,6,7,8,9,10]
    y64 = 0.0_real64
    alpha64 = 2.0_real64
    call axpy(alpha64, x64, y64)
    write(output_unit,*) "Double Precision
    ↪  AXPY Result:"
    write(output_unit,*) y64
end program test_axpy
```

- We test the double precision AXPY operations through the `axpy` interface.
- We initialize vectors and scalars, call the `axpy` method from our package, and print the results.
- This modular approach makes it easy to extend and maintain the BLAS wrapper.

# Test program



```
program test_axpy
    use iso_fortran_env, only: real64,
    ↪  output_unit
    use blas
    implicit none
    integer, parameter :: n = 10
    real(real64) :: x64(n), y64(n),
    ↪  alpha64
    x64 = [1,2,3,4,5,6,7,8,9,10]
    y64 = 0.0_real64
    alpha64 = 2.0_real64
    call axpy(alpha64, x64, y64)
    write(output_unit,*) "Double Precision
    ↪  AXPY Result:"
    write(output_unit,*) y64
end program test_axpy
```

- We test the double precision AXPY operations through the `axpy` interface.
- We initialize vectors and scalars, call the `axpy` method from our package, and print the results.
- ✔ This modular approach makes it easy to extend and maintain the BLAS wrapper.

**Exercise**

Implement the single precision, complex, and double complex versions of AXPY in the `blas` module and test them in the test program.

# AXPY: A Starting Point for Parallelism

3 Level 1 BLAS: Vector operations

- AXPY is a simple routine, ideal for exploring parallelism
- The operation can be parallelized by splitting vectors into chunks
- Each chunk can be computed independently in parallel

## OpenMP Parallelization

- OpenMP: API for shared memory parallel programming
- Uses **compiler directives** (special comments)
- Supports C, C++, and Fortran
- Portable and scalable for multi-core processors

# OpenMP AXPY Example

3 Level 1 BLAS: Vector operations

```
program axpy_opm_example
    use iso_fortran_env, only: int64, real64, output_unit
    use omp_lib
    implicit none
    integer(kind=int64), parameter :: n = 5
    real(kind=real64) :: x(n), y(n), alpha
    integer(kind=int64) :: i
    ! Initialize the vectors and scalar
    x = [1.0, 2.0, 3.0, 4.0, 5.0]
    y = [10.0, 20.0, 30.0, 40.0, 50.0]
    alpha = 2.0
    ! Write the OpenMP directive to parallelize the for loop
    !$omp parallel do
    do i = 1, n
        y(i) = y(i) + alpha * x(i)
```

# OpenMP AXPY Example



```
    end do
    !$omp end parallel do
    ! Print the result
    write(output_unit, '("Resulting vector y:")')
    do i = 1, n
        write(output_unit, '(F6.2)', advance='no') y(i)
    end do
    write(output_unit, '("")')
    ! Return
    return
end program axpy_opm_example
```

- Include OpenMP: `use omp_lib`
- Directive: `!$omp parallel do`
- Compiler spawns threads to distribute loop iterations

# Compiling with OpenMP



Compile the OpenMP program:

```
gfortran -o axpy_omp_example axpy_omp_example.f90 -fopenmp
```

Run the program:

```
./axpy_omp_example
```

## Controlling Thread Count

Set number of threads via environment variable:

```
export OMP_NUM_THREADS=4
./axpy_omp_example
```

Or in code: `call omp_set_num_threads(4)`

# Querying Thread Information



Get the number of threads being used:

```
integer(kind=int64) :: nthreads
!$omp parallel
!$omp single
    nthreads = omp_get_num_threads()
!$omp end single
!$omp end parallel
write(output_unit, '("Number of threads: ", I2)') nthreads
```

- Use `omp_get_num_threads()` to query
- *`!$omp single`* ensures only one thread updates
- Must be called within a parallel region

# Two Key Questions



1. **How are threads scheduled?**
   - How are loop iterations distributed among threads?
2. **Who owns what data?**
   - Which variables are shared vs. private?

# OpenMP Scheduling Policies



Specified using `schedule` clause:

- `static`: Equal-sized chunks (default)
- `static, chunk_size`: Fixed chunk size
- `dynamic`: Iterations assigned as threads become available
- `guided`: Dynamic with decreasing chunk sizes
- `runtime`: Determined by `OMP_SCHEDULE` environment variable
- `auto`: Compiler decides

```
!$omp parallel do schedule(static, chunk_size)
do i = 1, n
    y(i) = alpha * x(i) + y(i)
end do
!$omp end parallel do
```

# Data Sharing Clauses



Control variable visibility between threads:

- `shared`: Single instance visible to all threads
- `private`: Each thread has its own uninitialized copy
- `firstprivate`: Like private, but initialized from original
- `lastprivate`: Private, with final value copied back

Example for AXPY:

```
!$omp parallel do shared(x, y, alpha) private(i) schedule(dynamic)
do i = 1, n
    y(i) = alpha * x(i) + y(i)
end do
!$omp end parallel do
```

# Performance Measurement Strategy

3 Level 1 BLAS: Vector operations

- Use `omp_get_wtime()` for accurate timing
- Run multiple iterations for reliable measurements
- Use sufficiently large problem sizes
- Read problem size from command line
- Use allocatable arrays for dynamic sizing

## Compilation with optimization

```
gfortran -O3 -march=native -mtune=alderlake -o axpy_omp axpy_omp_time.f90 -fopenmp
```

# Timing Code Example



```
do i = 1, 1000
    elapsed_time = 0.0
    t1 = omp_get_wtime() ! Start timer
    !$omp parallel do shared(x, y, alpha) private(j) schedule(static)
    do j = 1, n
        y(j) = alpha * x(j) + y(j)
    end do
    !$omp end parallel do
    t2 = omp_get_wtime() ! Stop timer
    elapsed_time = elapsed_time + (t2 - t1)
end do
```

- Average over many iterations
- Use `omp_get_wtime()` instead of `cpu_time`
- Measure wall-clock time

# Performance Analysis: roofline model



Let us analyze the performance of our OpenMP AXPY implementation using the roofline model.

First the characteristics of the AXPY operation:

- AXPY operation: $y \leftarrow \alpha x + y$
- Floating-point operations (FLOPs): $2n$ (1 multiplication + 1 addition per element)
- Data movement: $3n$ (read $x$, read $y$, write $y$)
- Operational intensity: $\frac{2n}{3 \times 8n} = \frac{2}{24} = \frac{1}{12}$ FLOPs/byte

To plot the roofline model, we need to measure/know:

- Peak computational performance (FLOPs/s)
- Memory bandwidth (bytes/s)

# Performance Analysis: roofline model



On my CPU (Intel® Core™ i9-14900HX) I have:

- Peak performance: 844.8 GFLOPs (double precision)
- Memory bandwidth: 89.6 GB/s (measured with *stream*)

The **operational intensity** of AXPY is $1/12$ FLOPs/byte, which is independent of $n$, this is a clear indication of a **memory-bound** operation.

# Performance Analysis: roofline model

3 Level 1 BLAS: Vector operations

On my CPU (Intel® Core™ i9-14900HX) I have:

- Peak performance: 844.8 GFLOPs (double precision)
- Memory bandwidth: 89.6 GB/s (measured with *stream*)

The **operational intensity** of AXPY is $1/12$ FLOPs/byte, which is independent of $n$, this is a clear indication of a **memory-bound** operation.

The memory-bound performance ceiling is given by:

$$\begin{aligned} \text{Performance}_{\text{max, memory-bound}} &= \text{Operational Intensity} \times \text{Memory Bandwidth} \\ &= \frac{1}{12} \times 89.6 \text{ GFLOPs} \approx 7.47 \text{ GFLOPs} \end{aligned}$$

# Measuring Performance



After running the different AXPY implementation with a large vector size (e.g., slightly larger than the L3 cache size) and measuring the execution time, we can compute the achieved performance:

# Measuring Performance



After running the different AXPY implementation with a large vector size (e.g., slightly larger than the L3 cache size) and measuring the execution time, we can compute the achieved performance:

Assuming we measured an execution time of $t$ seconds, the achieved performance is:

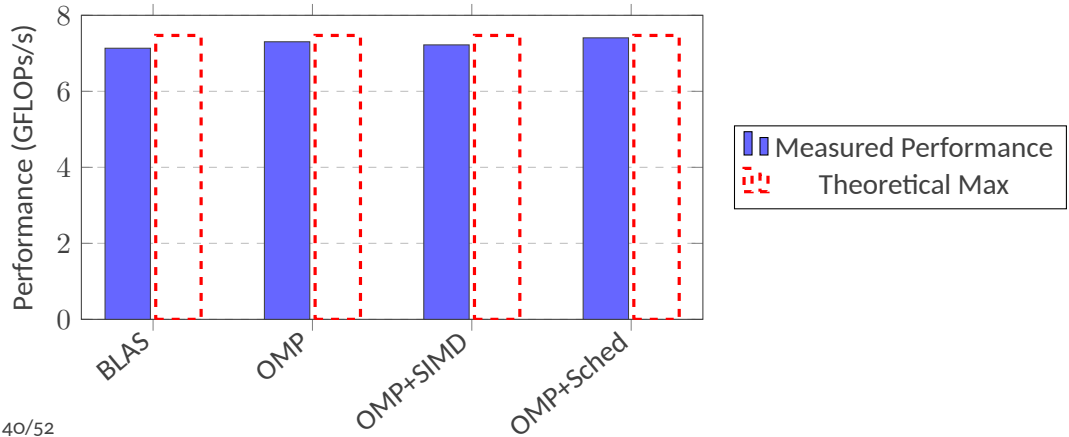
$$\text{Achieved Performance} = \frac{2n}{t} \text{ FLOPs/s}$$

```
do rep = 1, reps
      y = 0.0_dp ! Reset y for each repetition
      t0 = omp_get_wtime()
      call daxpy(n, alpha, x, 1, y, 1)
      t1 = omp_get_wtime() ! BLAS implementation
      time_blas = time_blas + (t1 - t0)
end do
```

# Visualizing results



I tested it my machine with $n = 6000000$ over $100$ repetitions and obtained the following:



# Some caveats



To obtain reasonable numbers from our implementations we **need to enable compiler optimizations**:

```
gfortran -O3 -march=native -mtune=native
```

- `-O3` enables high-level optimizations
- `-march=native` enables instructions for the host CPU
- `-mtune=native` optimizes for the host CPU microarchitecture

If we want to enable them in our **CMake project** we need to add the following lines to the `CMakeLists.txt` file:

```
set(CMAKE_Fortran_FLAGS_RELEASE "-O3 -march=native -mtune=native")
set(CMAKE_BUILD_TYPE Release CACHE STRING "Build type" FORCE)
```

# Full example code



The **full example code** for the OpenMP AXPY implementation with performance measurement is available at:

github.com/High-Performance-Linear-Algebra/objblas/tree/main

As usual, it can be obtained by doing

```
git clone git@github.com:High-Performance-Linear-Algebra/objblas.git
cd objblas
```

and the example run by doing

```
mkdir build
cd build
cmake ..
make # or ninja
./axpy_perf
```

# Varying the number of threads



We can analyze the performance of our OpenMP AXPY implementation by varying the number of threads used.

- Set the number of threads using the `OMP_NUM_THREADS` environment variable
- Measure execution time and compute achieved performance for each thread count
- Plot performance vs. number of threads to visualize scaling behavior

**Example command to run with different thread counts**

```
export OMP_NUM_THREADS=4
./axpy_perf
```

# Implementation of the AXPY scaling driver



Since we want to vary the number of threads, and we want to visualize the performance, we can implement a simple bash script which will do the job for us.

```
#!/bin/bash
BUILD_DIR=../../build
THREADS=(1 2 4 8 16 32)
SIZES=(1000000 5000000 10000000 50000000 100000000)
repetitions=50

for size in "${SIZES[@]}"; do
    for threads in "${THREADS[@]}"; do
        export OMP_NUM_THREADS=$threads
        echo "Running axpy_scaling with size=$size and threads=$threads"
        $BUILD_DIR/axpy_scaling $size $repetitions
    done
done
```

# Implementation of the AXPY scaling driver



The `axpy_scaling` program needs to read the size and number of repetitions from the command line, so we can implement it as follows:

```
integer :: reps, n
character(len=20) :: n_str, reps_str
if (command_argument_count() < 2) then
    write(error_unit, *) 'Usage: axpy_scaling <problem_size> <repetitions>'
    stop
end if
call get_command_argument(1, n_str)
call get_command_argument(2, reps_str)
read(n_str, *) n
read(reps_str, *) reps
```

# Implementation of the AXPY scaling driver



We allocate the arrays dynamically:

```
real(dp), allocatable :: x(:), y(:)
real(dp) :: alpha
integer :: stat
! Allocate and initialize data
allocate(x(n), y(n),stat=stat)
if (stat /= 0) then
    write(error_unit, *) 'Error allocating arrays of size ', n
    stop
end if

x = [(real(i, dp), i = 1, n)]
y = 0.0_dp
alpha = 2.0_dp
```

# Implementation of the AXPY scaling driver



Finally we can implement the timing loop as follows:

```
! Benchmark BLAS AXPY
time_blas = 0.0_dp
do rep = 1, reps
    y = 0.0_dp
    t0 = omp_get_wtime()
    call daxpy(n, alpha, x, 1, y, 1)
    t1 = omp_get_wtime()
    time_blas = time_blas + (t1 - t0)
end do
```

```
! Benchmark OpenMP AXPY
time_omp = 0.0_dp
do rep = 1, reps
    y = 0.0_dp
    t0 = omp_get_wtime()
    call axpy_omp(n, alpha, x, 1, y, 1)
    t1 = omp_get_wtime()
    time_omp = time_omp + (t1 - t0)
end do
```

We then compute the averages, and print the results to screen:

```
write(output_unit, *) 'Average time BLAS AXPY: ', time_blas/reps, ' seconds'
write(output_unit, *) 'Average time OpenMP AXPY: ', time_omp/reps, ' seconds'
```

# Writing to file for plotting



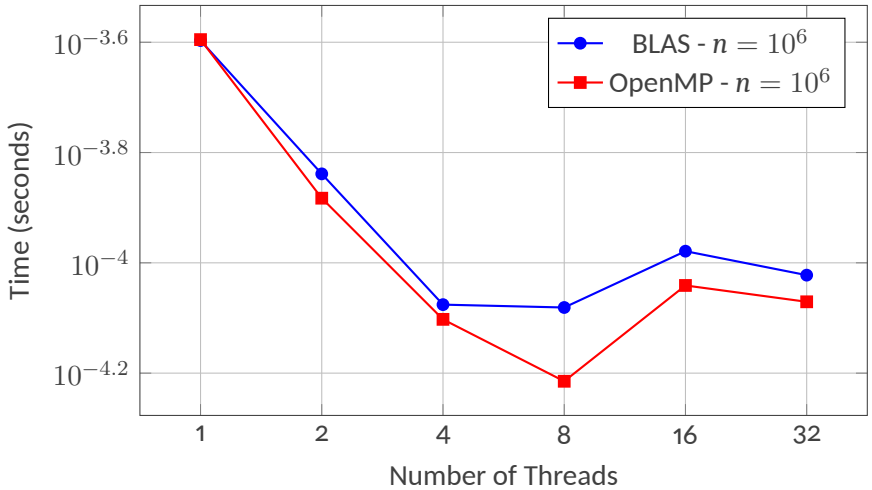
It is convenient to write the results to a file for later plotting. We can do it as follows:

```
open(unit=10, file='axpy_scaling_results.csv', status='unknown',
↪  action='write', position='append')
write(10, '(I10,I10,1X,F15.6,F15.6)') n, nthreads, time_blas, time_omp
close(10)
```

- `open` opens a file for writing
- `status='unknown'` creates the file if it doesn't exist, other possibilities for this argument are `'old'`, `'new'`, and `'replace'`
- `position='append'` adds data to the end of the file, other possibilities are `'rewind'` and `'replace'`, which start writing from the beginning of the file, and overwrite existing content.
- `write` formats and writes the data: problem size, thread count, and timings
- `close` closes the file

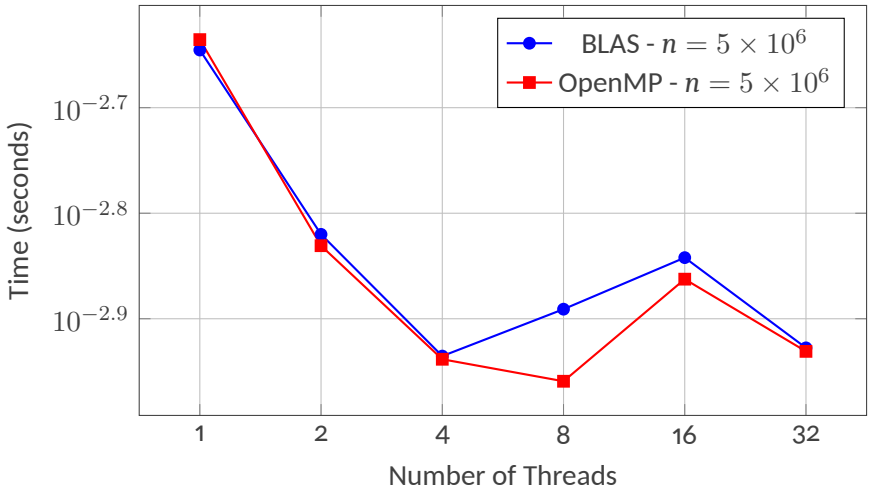
# Strong Scaling Results

3 Level 1 BLAS: Vector operations



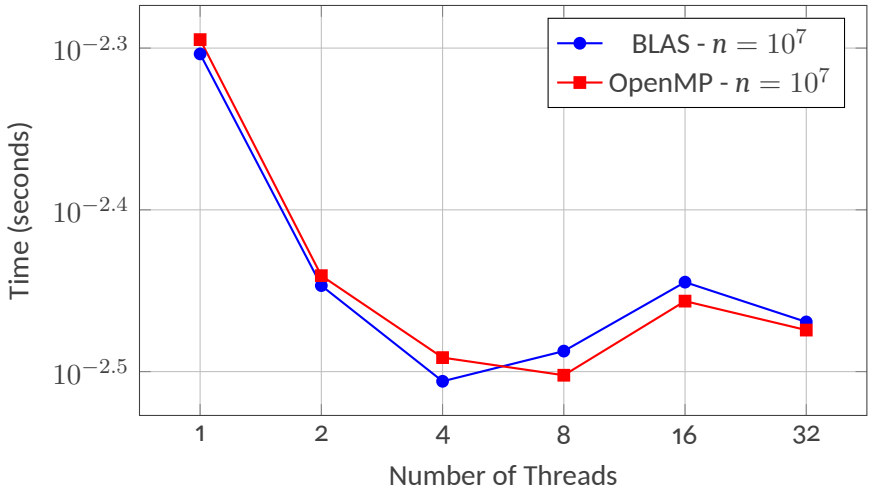
# Strong Scaling Results

3 Level 1 BLAS: Vector operations



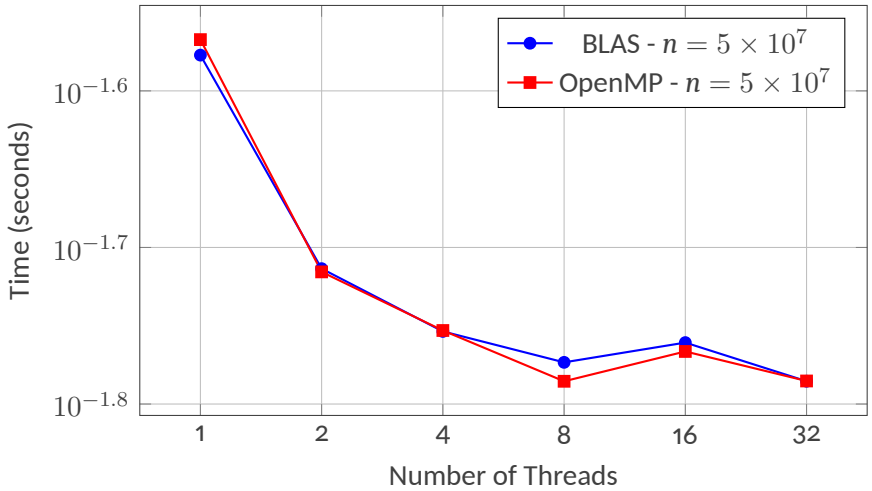
# Strong Scaling Results

## 3 Level 1 BLAS: Vector operations



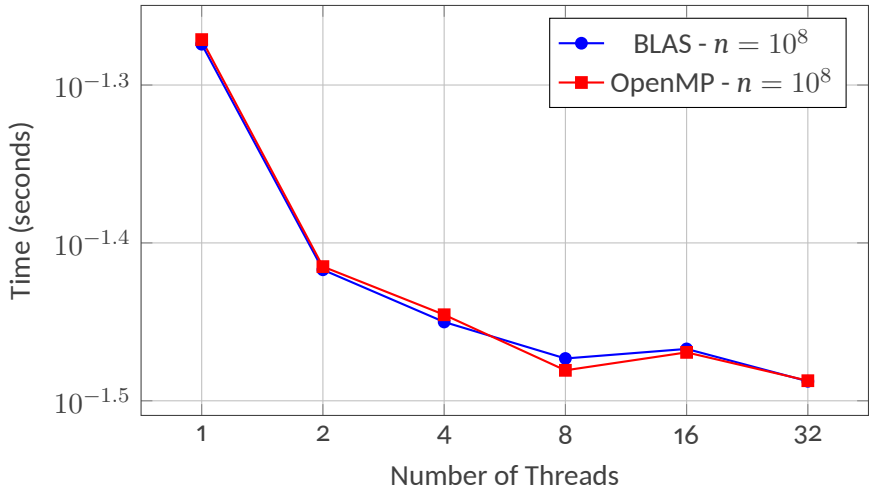
# Strong Scaling Results

3 Level 1 BLAS: Vector operations



# Strong Scaling Results

3 Level 1 BLAS: Vector operations



# Strong Scaling Results: Analysis



- For small problem sizes ($n = 10^6$), execution time increases with more threads
  - Thread creation overhead dominates computation
  - Memory bandwidth not fully utilized

# Strong Scaling Results: Analysis



- For small problem sizes ($n = 10^6$), execution time increases with more threads
  - Thread creation overhead dominates computation
  - Memory bandwidth not fully utilized
- For large problem sizes ($n \geq 5 \times 10^7$), execution time decreases with more threads
  - Computation becomes significant enough to benefit from parallelism
  - Better amortization of parallel overhead
  - Improved memory bandwidth utilization across cores

# Strong Scaling Results: Analysis



- For small problem sizes ($n = 10^6$), execution time increases with more threads
  - Thread creation overhead dominates computation
  - Memory bandwidth not fully utilized
- For large problem sizes ($n \geq 5 \times 10^7$), execution time decreases with more threads
  - Computation becomes significant enough to benefit from parallelism
  - Better amortization of parallel overhead
  - Improved memory bandwidth utilization across cores
- **Memory-bound nature persists:**
  - Scaling plateaus beyond 8-16 threads
  - Limited by memory bandwidth, not computation
  - Multiple threads saturate available bandwidth

# Strong Scaling Results: Analysis



- For small problem sizes ($n = 10^6$), execution time increases with more threads
- For large problem sizes ($n \geq 5 \times 10^7$), execution time decreases with more threads
- **Memory-bound nature persists:**

**Rule of thumb**

For memory-bound operations like AXPY, parallelism helps only when the problem size is large enough to amortize threading overhead and saturate memory bandwidth.

# Exercises

4 Conclusions

- What does it change if we use **single precision** instead of **double precision**?
- Investigate weak scaling behavior by increasing problem size proportionally with the number of threads.
- Implement **C**ontinuous **I**ntegration (CI) for the repository using GitHub Actions.
- We could test **one BLAS** implementation **against another** (e.g., OpenBLAS vs Intel MKL) and one compiler against another (e.g., GCC vs Intel). How would you implement this? A good idea would be to look at spack.io for package management.

# Summary and next-steps

4 Conclusions

- We have created a Fortran module wrapping BLAS AXPY routines with a clean interface.
- We implemented an OpenMP version of AXPY to explore parallelism.
- We analyzed performance using the roofline model, confirming AXPY is memory-bound.
- We studied strong scaling behavior by varying thread counts and problem sizes.

**Next Steps:**

- Look at inner products (DOT), norms and their parallel implementations.
- Start exploring Level 2 BLAS routines (matrix-vector operations).
- Look at more OpenMP pragmas and optimizations.

# High Performance Linear Algebra

Lecture 5: Continuing with BLAS, BLAS Level 1: DOT and level 2: GEMV

Ph.D. program in High Performance Scientific Computing

**Fabio Durastante** **Pasqua D'Ambra** **Salvatore Filippone**

November 24, 2025 — 14.00:16.00

Dipartimento
di Matematica
Università di Pisa

# Back from the past



- Last time we have seen:
  - Introduction to the general BLAS idea
  - BLAS Level 1: vector-vector operations
  - Performance considerations
- Today we will continue with:
  - BLAS Level 1: DOT, NRM2 and Givens rotations
  - BLAS Level 2: GEMV

# Table of Contents

2 BLAS Level 1: DOT


- BLAS Level 1: DOT
  - An interlude on Fortran functions
  - Make DOT parallel
  - Reductions, atomics and critical
  - OpenMP implementation
  - DOT: Roofline model
- BLAS Level 1: NRM2
- All the other Level 1 BLAS
  - Level 2 BLAS
- Summary and next lecture

# DOT: Dot Product



Another important operation is the dot product, which is defined as:

$$c = x^\top y = \sum_{i=1}^{n} x_i y_i$$

- The dot product is a scalar product of two vectors
- Sum of the products of corresponding elements
- BLAS routine (double precision): `ddot`

```
c = ddot(n, x, incx, y, incy)
```

where `incx` and `incy` are the increments for the input vectors $x$ and $y$.

# Fortran Functions: Declaration



- Functions return a single value
- Must declare return type
- Two ways to declare:

**Method 1: Classic Fortran style**

```
real(real64) function my_function(x, y)
    real(real64), intent(in) :: x, y
    my_function = x + y
end function my_function
```

**Method 2: Result clause**

```
function my_function(x, y) result(z)
    real(real64), intent(in) :: x, y
    real(real64) :: z
    z = x + y
end function my_function
```

# Fortran Functions: External Functions



- Some BLAS routines are **external** functions: compiled separately, **no** interface
- Must declare in calling program

```
program main
    use iso_fortran_env, only: real64
    implicit none
    real(real64) :: ddot  ! Declaration
    real(real64) :: x(3), y(3), result
    x = [1.0, 2.0, 3.0]
    y = [4.0, 5.0, 6.0]
    result = ddot(3, x, 1, y, 1)
end program main
```

**Important**

Without declaration and without `implicit none`, the compiler assumes `ddot` returns `real(real32)`, possibly causing errors.

# Fortran Functions vs Subroutines



**Functions**

- Return single value
- Used in expressions
- Example: `ddot`

```
c = ddot(n, x, 1, y, 1)
```

**Subroutines**

- Return via arguments
- Called with `call`
- Example: `daxpy`

```
call daxpy(n, a, x, 1, y, 1)
```

**BLAS Convention**

- Scalar results: functions (`ddot`, `dnrm2`)
- Vector/matrix results: subroutines (`daxpy`, `dgemv`)

# Passing Functions as Arguments



- Sometimes we need to pass a function to a subroutine
- Common in numerical algorithms (integration, optimization)
- Fortran provides mechanisms for this

## Example Use Cases

- Numerical integration: $\int_a^b f(x)dx$ for different functions $f$,
- Root finding: find $x$ such that $f(x) = 0$ for different functions $f$,
- Optimization: minimize $f(x)$ for different functions $f$.
- Compute $f(A)\mathbf{v}$ for different functions $f$ and matrix $A$.

# Method 1: External Procedure



Classic Fortran approach (using `external`)—which is still valid today, but less safe and should be avoided in modern code.

# Method 1: External Procedure



Classic Fortran approach (using `external`)—which is still valid today, but less safe and should be avoided in modern code.

Step 1: **Define the function**

```
real(real64) function my_func(x)
    real(real64), intent(in) :: x
    my_func = x**2
end function my_func
```

# Method 1: External Procedure



Classic Fortran approach (using `external`)—which is still valid today, but less safe and should be avoided in modern code.

**Subroutine that accepts function:**

```
subroutine integrate(f, a, b, result)
    real(real64), external :: f
    real(real64), intent(in) :: a, b
    real(real64), intent(out) :: result
    ! Integration code using f(x)
    result = (b-a) * f((a+b)/2.0)
end subroutine integrate
```

# Method 1: External Procedure



Classic Fortran approach (using `external`)—which is still valid today, but less safe and should be avoided in modern code.

**Subroutine that accepts function:**

```
subroutine integrate(f, a, b, result)
    real(real64), external :: f
    real(real64), intent(in) :: a, b
    real(real64), intent(out) :: result
    ! Integration code using f(x)
    result = (b-a) * f((a+b)/2.0)
end subroutine integrate
```

- `external` declares `f` as external function
- No type checking of arguments

# Method 2: Procedure Pointer (Modern Fortran)



The procedure in modern Fortran is to use **procedure interfaces** for type safety.

# Method 2: Procedure Pointer (Modern Fortran)



The procedure in modern Fortran is to use **procedure interfaces** for type safety.

**Define interface:**

```
abstract interface
    real(real64) function func_interface(x)
        real(real64), intent(in) :: x
    end function func_interface
end interface
```

# Method 2: Procedure Pointer (Modern Fortran)



The procedure in modern Fortran is to use **procedure interfaces** for type safety.

**Subroutine with procedure argument:**

```
subroutine integrate(f, a, b, result)
    procedure(func_interface) :: f
    real(real64), intent(in) :: a, b
    real(real64), intent(out) :: result
    result = (b-a) * f((a+b)/2.0)
end subroutine integrate
```

# Method 2: Procedure Pointer (Modern Fortran)



The procedure in modern Fortran is to use **procedure interfaces** for type safety.

**Subroutine with procedure argument:**

```
subroutine integrate(f, a, b, result)
    procedure(func_interface) :: f
    real(real64), intent(in) :: a, b
    real(real64), intent(out) :: result
    result = (b-a) * f((a+b)/2.0)
end subroutine integrate
```

- `procedure(interface_name)` provides type safety
- Compiler checks function signature
- **Recommended** for modern code

# Using the Function Argument



**Calling the subroutine:**

```
program main
    use iso_fortran_env, &
        only: real64
    implicit none
    real(real64) :: result
    call integrate(my_func, &
        0.0_real64, &
        1.0_real64, result)
    print *, "Result:", result
contains
    real(real64) function &
        my_func(x)
        real(real64), &
            intent(in) :: x
        my_func = x**2
    end function my_func
end program main
```

- Pass function name without parentheses
- Function **must** match expected signature, as defined by the `abstract interface`.

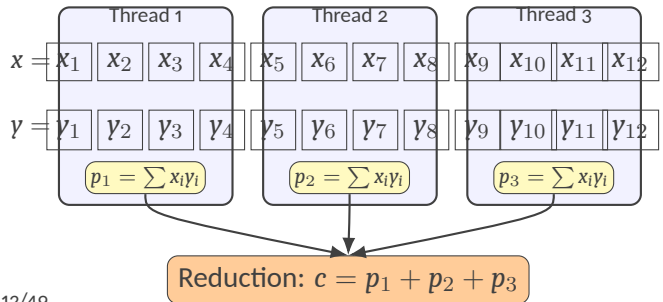
# DOT: Parallelization Strategy



## Question

What kind of parallelism can we exploit?

- The dot product is a **reduction** operation
- Strategy:




1. Split vectors into chunks
2. Each thread computes local dot
3. Reduce partial sums

# DOT: Parallelization Strategy



## Question

What kind of parallelism can we exploit?

- The dot product is a **reduction** operation
- Strategy:
  - Split vectors into chunks
  - Each thread computes local dot product
  - Reduce partial sums to get final result
- Good example for OpenMP parallelization

## Reduction operations

But how do we implement reductions in OpenMP?

# OpenMP Reduction Clause: Idea



**Pattern:** combine per-thread partial results into one final value.

- Syntax (loop form): `!$omp parallel do reduction(op:var1,var2,...)`
- Each listed variable gets:
  1. private copy (initialized),
  2. local accumulation,
  3. final merge.
- Avoid manual critical sections; scalable; fewer false sharing issues.
- Works for associative/commutative operations
  - floating point is *not associative*: order may changes result slightly!

**Supported intrinsic operators (Fortran)**

```
+ - * .and. .or. .xor. max min iand ior ieor
```

# Basic Examples

## 2 BLAS Level 1: DOT

```
! Sum of an array
total = 0.0_real64              ! Initialize shared reduction variable; each
↪  thread gets a private copy set to 0
!$omp parallel do reduction(+:total)
do i = 1, n                     ! Iterations divided among threads
    total = total + a(i)        ! Each thread accumulates into its private
    ↪  'total'
end do                          ! Runtime combines all private totals
↪  (addition) into the shared 'total'
```

# Basic Examples

## 2 BLAS Level 1: DOT

```
! Maximum over array
mval = -huge(mval)                ! Initialize with very small value; private
↪  copies get same initialization
!$omp parallel do reduction(max:mval)
do i = 1, n                       ! Iterations executed in parallel
    if (a(i) > mval) mval = a(i)  ! Track local maximum in each thread
end do                            ! Runtime computes global max from all
↪  thread-local maxima
```

# Basic Examples

## 2 BLAS Level 1: DOT

```
! Logical OR over flags
any_flag = .false.              ! Identity for .or.; private copies start as
↪  .false.
!$omp parallel do reduction(.or.:any_flag)
do i = 1, n                     ! Parallel traversal of flags
    any_flag = any_flag .or. flags(i)  ! Accumulate local logical OR
end do                          ! Final any_flag is OR of all thread-local
↪  results
```

# What the Runtime Does



1. Creates one private copy per thread (initialized suitably).
2. Executes loop chunks independently, updating private copies.
3. At implicit barrier: combines privates into the original variable: *in unspecified order*.

## Initialization Rules

- `+`: zero; `*`: one; logical: identity; `max/min`: extreme values.
- Ensure you **do NOT re-initialize** inside the loop.

## Numerical Note

Floating point reductions are order-dependent; expect tiny round-off differences vs serial.

# Common Pitfalls



- Forgetting initialization before the directive (needed for clarity; runtime overwrites).
- Writing to the reduction variable outside the loop: creates race conditions.
- Using non-associative custom operations without care (order not guaranteed).
- Large objects: reduction copies can be expensive; consider **manual chunking** or **atomic updates** if contention low.

# Atomic Updates



Atomic updates protect a single read-modify-write operation on one scalar or array element so that threads do not interleave it. In OpenMP add `!$omp atomic` before the assignment; only that memory operation is serialized, not the whole loop.

**When to use:**

- Few conflicting updates (low contention).
- Irregular indices (e.g. histogram, sparse gather).
- Operation not available as a reduction.

**Drawbacks:** High contention degrades scalability; for dense loops prefer a reduction (private copies + final combine).

**Rules:** Single assignment only; limited set of operators; protects exactly one memory location.

**Example:**

```
total = 0.0_real64
!$omp parallel do shared(total)
do i = 1, n
    !$omp atomic
    total = total + a(i)
end do
```

# Atomic with function calls



**Example:**

```
!$omp parallel do shared(total)
do i = 1, n
    !$omp atomic
    total = total + my_func(a(i))
end do
```

**Function:**

```
real(real64) function my_func(x)
    real(real64), intent(in) :: x
    my_func = x**2
end function my_func
```

- Function calls inside atomic regions can lead to undefined behavior if the function itself is not thread-safe.
- Ensure that any function called within an atomic region does not modify shared state or rely on non-thread-safe operations.
- It is only the update to the memory location of the variable `total` that will occur atomically.

# Atomic with function calls



**Example:**

```
!$omp parallel do shared(total)
do i = 1, n
    !$omp atomic
    total = total + my_func(a(i))
end do
```

**Function:**

```
real(real64) function my_func(x)
    real(real64), intent(in) :: x
!$omp critical (my_func_lock)
    my_func = x**2
!$omp end critical (my_func_lock)
end function my_func
```

- If the application developer does not intend to permit the threads to execute `my_func` at the same time, then the *`!$omp critical`* construct must be used instead,
- the critical construct provides a means to ensure that multiple threads do not attempt to update the same shared data simultaneously,
- When a thread encounters a critical construct, it waits until **no other thread is executing a critical region with the same name**.

# DOT: OpenMP Implementation

2 BLAS Level 1: DOT

```
program dot_omp
    use iso_fortran_env, only: output_unit, real64
    use omp_lib
    implicit none
    integer :: i
    integer, parameter :: n = 10000
    integer :: nthreads
    real(real64) :: x(n), y(n)
    real(real64) :: sum, c
    real(real64) :: start_time, end_time
    real(real64) :: ddot
    !$omp parallel
    !$omp single
    nthreads = omp_get_num_threads()
```

# DOT: OpenMP Implementation



```
!$omp end single
!$omp end parallel
write(output_unit,'("Number of threads: ",I0)') nthreads
! Initialize arrays
x = 1.0
y = 2.0
c = 0.0
start_time = omp_get_wtime()
!$omp parallel do private(i) shared(x,y) reduction(+:c)
do i = 1, n
    c = c + x(i) * y(i)
end do
!$omp end parallel do
end_time = omp_get_wtime()
```

# DOT: OpenMP Implementation



```
    write(output_unit,'("Dot product: ",F0.2)') sum
    write(output_unit,'("Time taken: ",E0.2)') end_time - start_time
    ! Check the result with blas
    call cpu_time(start_time)
    sum = ddot(n, x, 1, y, 1)
    call cpu_time(end_time)
    if (abs(c - sum) > 1.0e-12) then
        write(output_unit,'("Abs. Error: ",F0.2)') abs(c - sum)
    else
        write(output_unit,'("Result is correct")')
    end if
    write(output_unit,'("BLAS time: ",E0.2)') end_time - start_time
end program dot_omp
```

# DOT: OpenMP Implementation



- `!$omp parallel do` creates parallel region
- `reduction(+:c)` clause for reduction variable
- Each thread has private copy of `c`
- Values combined at end of parallel region

**Exercise**

A possible exercise is to implement the `ddot` function using, instead of the reduction clause, `atomic` or `critical` sections, and then compare the performance with the reduction version.

# Back to the roofline model

## 2 BLAS Level 1: DOT

- DOT operation: $c = \sum_{i=1}^{n} x_i y_i$
- Arithmetic: $2n$ flops (multiply + add per element)
- Memory traffic: $2n \times 8$ bytes (read $x_i$ and $y_i$)
- Arithmetic intensity:

$$AI = \frac{2n}{16n} = \frac{1}{8} \text{ flops/byte}$$

**Memory Bound**

DOT is **memory bound**: performance limited by **memory bandwidth**, not compute capability.

- Peak performance: $P = \min(\pi, \beta \times AI) = \beta \times 1/8$
- where $\beta$ is memory bandwidth (GB/s) and $\pi$ is peak compute (GFLOPS/s)

# Table of Contents

3 BLAS Level 1: NRM2


- BLAS Level 1: DOT
  - An interlude on Fortran functions
  - Make DOT parallel
  - Reductions, atomics and critical
  - OpenMP implementation
  - DOT: Roofline model
- BLAS Level 1: NRM2
- All the other Level 1 BLAS
  - Level 2 BLAS
- Summary and next lecture

# NRM2: 2-Norm of a Vector



The 2-norm is defined as:

$$c = \|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$$

BLAS routine: `dnrm2`

```
c = dnrm2(n, x, incx)
```

where `incx` is the increment for the input vector $x$.

# NRM2: OpenMP Implementation



Similar implementation to dot product:

```
c = 0.0
!$omp parallel do reduction(+:c) shared(x) private(i)
do i = 1, n
        c = c + x(i)**2
end do
!$omp end parallel do
c = sqrt(c)
```

- Same reduction pattern
- Square root applied after parallel region
- Same declaration note applies to `dnrm2`

# Back to the roofline model

## 3 BLAS Level 1: NRM2

- NRM2 operation: $c = \sqrt{\sum_{i=1}^{n} x_i^2}$
- Arithmetic: $2n$ flops (square + add per element)
- Memory traffic: $n \times 8$ bytes (read $x_i$)
- Arithmetic intensity:

$$AI = \frac{2n}{8n} = \frac{1}{4} \text{ flops/byte}$$

### Memory Bound

NRM2 is also **memory bound**: performance limited by **memory bandwidth**, not compute capability.

- Peak performance: $P = \min(\pi, \beta \times AI) = \beta \times 1/4$
- where $\beta$ is memory bandwidth (GB/s) and $\pi$ is peak compute (GFLOPS/s)

# Table of Contents

4 All the other Level 1 BLAS


- BLAS Level 1: DOT
  - An interlude on Fortran functions
  - Make DOT parallel
  - Reductions, atomics and critical
  - OpenMP implementation
  - DOT: Roofline model
- BLAS Level 1: NRM2
- All the other Level 1 BLAS
  - Level 2 BLAS
- Summary and next lecture

# Other Level 1 BLAS Routines

4 All the other Level 1 BLAS

- **SCAL**: Scale vector by a constant

$$y = \alpha x,$$

- **COPY**: Copy vector $x$ to $y$,
- **SWAP**: Swap vectors $x$ and $y$,
- **ASUM**: 1-norm of a vector,
- **IAMAX**: finds the index of the first element having maximum absolute value.

## Exercise

Implement these routines using OpenMP parallelization, the OpenMP instruction seen so far are sufficient for this purpose.

# Givens rotations

4 All the other Level 1 BLAS

Another important class of Level 1 BLAS routines are those implementing Givens rotations.

- Givens rotation zeroes elements in vectors/matrices.
- Used in QR factorization, least squares problems.
- Basic operation:

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

  where $c = \cos(\theta), s = \sin(\theta)$.

BLAS routines: `drotg` (generate), `drot` (apply).

```
call drotg(a, b, c, s)
```

# Givens rotations



Another important class of Level 1 BLAS routines are those implementing Givens rotations.

- Givens rotation zeroes elements in vectors/matrices.
- Used in QR factorization, least squares problems.
- Basic operation:

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

  where $c = \cos(\theta)$, $s = \sin(\theta)$.

BLAS routines: `drotg` (generate), `drot` (apply).

```
call drot(n, x, incx, y, incy, c, s)
```

This applies the rotation to vectors $x$ and $y$.

# Givens rotations: example

4 All the other Level 1 BLAS

```
program givens_example
    use iso_fortran_env, only: real64
    implicit none
    real(real64) :: a, b, c, s
    a = 3.0_real64
    b = 4.0_real64
    call drotg(a, b, c, s)
    print *, "Givens rotation parameters:"
    print *, "c =", c, ", s =", s
end program givens_example
```

- This program computes the Givens rotation parameters for the vector $(3, 4)$.
- The output will show the cosine and sine values used in the rotation.
- These parameters can then be used to zero out one of the elements in a vector or matrix.

# Using Givens rotations to zero out elements



**Example: Zeroing out elements in a vector**

```
subroutine zero_givens(x, c, s)
    use iso_fortran_env, only: real64
    real(real64), intent(inout) :: x(:)
    real(real64), allocatable,
    ↪  intent(out) :: c(:), s(:)
    integer :: n, i
    real(real64) :: a, b, cc, ss
    n = size(x)
    if (n <= 1) then
        allocate(c(0), s(0))
        return
    end if
    allocate(c(n-1), s(n-1))
```

- This subroutine takes a vector `x` and computes Givens rotations to zero out all elements except the first one.
- It allocates arrays `c` and `s` to store the cosine and sine values of the rotations.

# Using Givens rotations to zero out elements



**Example: Zeroing out elements in a vector**

```
    do i = 2, n
        a = x(1)
        b = x(i)
        ! build Givens for (a,b)
        call drotg(a, b, cc, ss)
        c(i-1) = cc
        s(i-1) = ss
        ! Apply rotation to the 2-vector
        ↪  [x(1); x(i)] using BLAS drot
        call drot(1, x(1:1), 1, x(i:i),
        ↪  1, cc, ss)
    end do
end subroutine zero_givens
```

- This subroutine takes a vector `x` and computes Givens rotations to zero out all elements except the first one.
- It allocates arrays `c` and `s` to store the cosine and sine values of the rotations.
- For each element in the vector (from the second to the last), it computes the Givens rotation parameters using `drotg`.
- It then applies the rotation to the pair of elements $x(1)$ and $x(i)$ using `drot`.

# Level 2 BLAS Overview

## 4 All the other Level 1 BLAS

- Level 2 BLAS routines perform matrix-vector operations.

**Level 2 BLAS: matrix-vector, $O(n^2)$ operations**

| types | name ( options | size | arguments | ) | description | equation | flops | data |
|---|---|---|---|---|---|---|---|---|
| s, d, c, z | gemv ( trans, | m, n, | alpha, A, ldA, x, incx, beta, y, incy | ) | general matrix-vector multiply | $y = \alpha A^* x + \beta y$ | $2mn$ | $mn$ |
| c, z | hemv ( uplo, | n, | alpha, A, ldA, x, incx, beta, y, incy | ) | Hermitian matrix-vector mul. | $y = \alpha A x + \beta y$ | $2n^2$ | $n^2/2$ |
| s, d † | symv ( uplo, | n, | alpha, A, ldA, x, incx, beta, y, incy | ) | symmetric matrix-vector mul. | $y = \alpha A x + \beta y$ | $2n^2$ | $n^2/2$ |
| s, d, c, z | trmv ( uplo, trans, diag, | n, | A, ldA, x, incx | ) | triangular matrix-vector mul. | $x = A^* x$ | $n^2$ | $n^2/2$ |
| s, d, c, z | trsv ( uplo, trans, diag, | n, | A, ldA, x, incx | ) | triangular solve | $x = A^{-*} x$ | $n^2$ | $n^2/2$ |
| s, d | ger ( | m, n, alpha, x, incx, y, incy, | A, ldA ) | | general rank-1 update | $A = A + \alpha x y^T$ | $2mn$ | $mn$ |
| c, z | geru ( | m, n, alpha, x, incx, y, incy, | A, ldA ) | | general rank-1 update (complex) | $A = A + \alpha x y^T$ | $2mn$ | $mn$ |
| c, z | gerc ( | m, n, alpha, x, incx, y, incy, | A, ldA ) | | general rank-1 update (complex conj) | $A = A + \alpha x y^H$ | $2mn$ | $mn$ |
| s, d † | syr ( uplo, | n, alpha, x, incx, | A, ldA ) | | symmetric rank-1 update | $A = A + \alpha x x^T$ | $n^2$ | $n^2/2$ |
| c, z | her ( uplo, | n, alpha, x, incx, | A, ldA ) | | Hermitian rank-1 update | $A = A + \alpha x x^H$ | $n^2$ | $n^2/2$ |
| s, d | syr2 ( uplo, | n, alpha, x, incx, y, incy, | A, ldA ) | | symmetric rank-2 update | $A = A + \alpha x y^T + \alpha y x^T$ | $2n^2$ | $n^2/2$ |
| c, z | her2 ( uplo, | n, alpha, x, incx, y, incy, | A, ldA ) | | Hermitian rank-2 update | $A = A + \alpha x y^H + y(\alpha x)^H$ | $2n^2$ | $n^2/2$ |

- These routines are essential for many numerical algorithms, including solving linear systems and eigenvalue problems.

# The GEMV Routine

4 All the other Level 1 BLAS

- One of the most important Level 2 BLAS routines is **GEMV** (General Matrix-Vector multiplication).
- It computes the operation:

$$y = \alpha Ax + \beta y$$

  where $A$ is a matrix, $x$ and $y$ are vectors, and $\alpha$ and $\beta$ are scalars.
- GEMV is widely used in various applications, including solving linear systems and performing transformations.

# GEMV: BLAS Interface

4 All the other Level 1 BLAS

$$y = \alpha Ax + \beta y$$

- Routine: `dgemv` (double precision)
- Prototype:

```
call dgemv(trans, m, n, alpha, A, lda, x, incx, beta, y, incy)
! trans = 'N','T','C'; lda = leading dimension of A
```

- Column-major storage; lda = first dimension of A as declared.
- Increments allow strided access (usually 1).

# GEMV: Example Program

4 All the other Level 1 BLAS

```
program gemv_blas
    use iso_fortran_env, only: real64, output_unit, error_unit
    implicit none
    integer :: m, n, lda
    real(real64) :: alpha, beta
    real(real64), allocatable :: A(:,:), x(:), y(:)
    character(len=100) :: m_str, n_str
    real(real64) :: start_time, end_time, elapsed_time
    integer :: i,j,info
    ! Read m and n from command line arguments
    if (command_argument_count() < 2) then
        write(error_unit, '("Usage: gemv_blas <m> <n>")')
        stop
```

# GEMV: Example Program

4 All the other Level 1 BLAS

```
end if
call get_command_argument(1, m_str, status=info)
call get_command_argument(2, n_str, status=info)
if (info /= 0) then
    write(error_unit, '("Error reading command line arguments")')
    stop
end if
read(m_str, *) m
read(n_str, *) n
! set parameters
lda = m
alpha = 1.0d0
beta = 1.0d0
allocate(A(lda, n), x(n), y(m),stat=info)
```

# GEMV: Example Program

## 4 All the other Level 1 BLAS

```
if (info /= 0) then
    write(error_unit, '("Error allocating memory")')
    stop
end if
! Initialize matrix A and vectors x and y
do i = 1, m
    do j = 1, n
        A(i, j) = real(i + j, kind=real64)
    end do
end do
do i = 1, n
    x(i) = real(i, kind=real64)
end do
do i = 1, m
```

# GEMV: Example Program

4 All the other Level 1 BLAS

```
        y(i) = real(i, kind=real64)
    end do
    ! Compute the matrix-vector product using BLAS gemv
    call cpu_time(start_time)
    call dgemv('N', m, n, alpha, A, lda, x, 1, beta, y, 1)
    call cpu_time(end_time)
    elapsed_time = end_time - start_time
    write(output_unit, '("BLAS dgemv time: ", E0.6)') elapsed_time
    ! Free allocated memory
    deallocate(A, x, y, stat=info)
    if (info /= 0) then
        write(error_unit, '("Error deallocating memory")')
        stop
```

# GEMV: Example Program

4 All the other Level 1 BLAS

```
    end if
end program gemv_blas
```

# Organizing Implementations

## 4 All the other Level 1 BLAS

- Place multiple GEMV variants in a module for reuse.

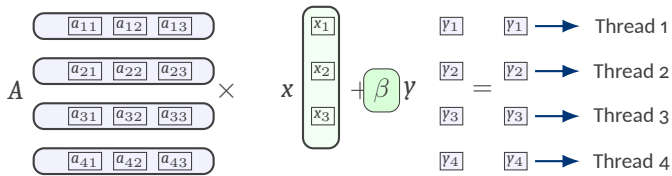
```
module gemvmod
    use iso_fortran_env
    use omp_lib
    implicit none
    private
    public :: gemv_openmp_n, gemv_openmp_n_block
contains
    ! Implementations follow
    ⟨ see next slides ⟩
end module gemvmod
```

# Parallel Formulation

## 4 All the other Level 1 BLAS

$$y_i = \alpha \sum_{j=1}^{n} A_{ij} x_j + \beta y_i, \quad i = 1, \ldots, m$$

- Natural outer-loop parallelism over rows i (independent updates).
- Uses dot products: reuse optimized `ddot`.



# Variant 1: Parallel Over Rows

4 All the other Level 1 BLAS

```
subroutine gemv_openmp_n(m, n, alpha, A, lda, x, beta, y)
    use iso_fortran_env, only: real64
    use omp_lib
    implicit none
    integer, intent(in) :: m, n, lda
    real(real64), intent(in) :: alpha, beta
    real(real64), intent(in) :: A(lda,*), x(*)
    real(real64), intent(inout) :: y(*)
    real(real64) :: ddot
    integer :: i
    real(real64) :: temp
    !$omp parallel do private(i,temp) shared(m,n,A,x,y,alpha,beta)
    do i = 1, m
```

# Variant 1: Parallel Over Rows

4 All the other Level 1 BLAS

```
        temp = ddot(n, A(i,1:n), 1, x, 1)
        y(i) = alpha*temp + beta*y(i)
    end do
    !$omp end parallel do
end subroutine
```

⚠ Strided row access in column-major layout: **less cache-friendly**.

## Memory Access Patterns: Cache Misses

Accessing data in a non-contiguous manner can lead to cache misses, as the CPU cache is optimized for spatial locality. When data is accessed sequentially, it is more likely to be present in the cache, leading to faster access times. However, when data is accessed in a strided or non-contiguous manner, it can result in **cache misses**, as the required data may not be present in the cache, leading to **slower access** times due to **fetching data from main memory**.

# Variant 1b: Dynamic Scheduling

4 All the other Level 1 BLAS

```
subroutine gemv_openmp_n_block(m, n, alpha, A, lda, x, beta, y)
    use iso_fortran_env, only: real64
    use omp_lib
    implicit none
    integer, intent(in) :: m, n, lda
    real(real64), intent(in) :: alpha, beta
    real(real64), intent(in) :: A(lda,*), x(*)
    real(real64), intent(inout) :: y(*)
    real(real64) :: ddot, temp
    integer :: i
    !$omp parallel do schedule(dynamic,32) private(i,temp) &
    !$omp     shared(m,n,A,x,y,alpha,beta)
    do i = 1, m
```

# Variant 1b: Dynamic Scheduling

4 All the other Level 1 BLAS

```
        temp = ddot(n, A(i,1:n), 1, x, 1)
        y(i) = alpha*temp + beta*y(i)
    end do
    !$omp end parallel do
end subroutine
```

- Dynamic chunks mitigate load imbalance; chunk size tunable.

## OpenMP schedule clause: chunk size reminder

- Syntax: `!$omp do schedule(kind[,chunk])`
- `chunk` = max iterations handed to a thread each time it receives work.
- `dynamic[,c]`: Threads pull blocks of size `c` from a queue until done. Good for irregular work; more overhead. Default `c` often = 1 if omitted.

# OpenMP schedule clause: chunk size effects



- Rule of thumb: pick $c$ so that per-chunk work dominates scheduling cost.
  - Too small: higher scheduling overhead, more contention.
  - Too large: potential load imbalance (idle threads at end).

```
!$omp parallel do
↪  schedule(static,64)
do i = 1, n
    work(i)
end do
```

- Static scheduling with large chunks.
- Low scheduling overhead, but potential load imbalance.

```
!$omp parallel do
↪  schedule(dynamic,8)
do i = 1, n
    work(i)
end do
```

- Dynamic scheduling with small chunks.
- Better load balance, but higher overhead.

```
!$omp parallel do
↪  schedule(guided,4)
do i = 1, n
    work(i)
end do
```

- Guided scheduling with minimum chunk size.
- Balances load and overhead adaptively.

# OpenMP schedule clause: chunk size effects



- Rule of thumb: pick c so that per-chunk work dominates scheduling cost.
  - Too small: higher scheduling overhead, more contention.
  - Too large: potential load imbalance (idle threads at end).

**Tip**

Benchmark several chunk sizes; optimal values depend on loop body cost variability and hardware.

# Table of Contents

5 Summary and next lecture


- BLAS Level 1: DOT
  - An interlude on Fortran functions
  - Make DOT parallel
  - Reductions, atomics and critical
  - OpenMP implementation
  - DOT: Roofline model
- BLAS Level 1: NRM2
- All the other Level 1 BLAS
  - Level 2 BLAS
- Summary and next lecture

# Summary and Next Lecture

5 Summary and next lecture

- OpenMP reductions simplify parallel accumulation patterns.
- **Atomic updates** provide fine-grained synchronization for low-contention cases.
- Level 1 BLAS routines (DOT, NRM2) are **memory-bound**.
- Level 2 BLAS routines (GEMV) involve matrix-vector operations; parallelism can be exploited over rows.
- Memory access patterns **significantly impact performance**; consider data layout and access order.

**Next lecture**

- Investigate better memory access patterns for GEMV.
- Level 3 BLAS: Matrix-Matrix operations (GEMM).
- Blocking techniques for cache efficiency.

# High Performance Linear Algebra

Lecture 6: Continuing with BLAS, BLAS Level 2: GEMV and level 3: GEMM

Ph.D. program in High Performance Scientific Computing

**Fabio Durastante** **Pasqua D'Ambra** **Salvatore Filippone**

November 26, 2025 — 14.00:16.00

Dipartimento
di Matematica
Università di Pisa

# Back from the past



- Last time we have seen:
  - BLAS Level 1: DOT, NRM2 and Givens rotations
  - BLAS Level 2: GEMV a first look
- Today we will continue with:
  - BLAS Level 2: GEMV: better memory access patterns
  - BLAS Level 3: GEMM

# Table of Contents

2 BLAS Level 2: GEMV continued


- BLAS Level 2: GEMV continued
  - Looking at the performance: roofline model
  - BLAS Level 2: TRSV
- BLAS Level 3
  - DGEMM with OpenMP
  - Tiled version
  - Tiled version with OpenMP

# Last lecture

## 2 BLAS Level 2: GEMV continued

We have implemented two variants of the GEMV kernel, in the following `module`:

```
module gemvmod
    use iso_fortran_env
    use omp_lib
    implicit none
    private
    public :: gemv_openmp_n, gemv_openmp_n_block
contains
    ! Implementations in the last lecture
end module gemvmod
```

- `gemv_openmp_n`: simple OpenMP parallelization over rows,
- `gemv_openmp_n_block`: blocked version with OpenMP parallelization over blocks of rows.

# GEMV: limitations of the previous implementations



- Both implementations have a limitation: they access the matrix $A$ in column-major order, which is not cache friendly.
- We can improve the memory access pattern by changing the way we parallelize the computation.

# GEMV: limitations of the previous implementations

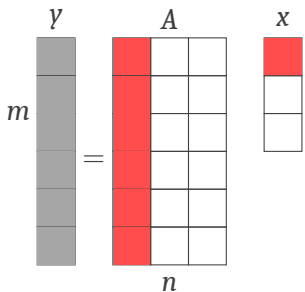


- Both implementations have a limitation: they access the matrix $A$ in column-major order, which is not cache friendly.
- We can improve the memory access pattern by changing the way we parallelize the computation.
- Instead of parallelizing over rows, we can **parallelize over columns**.

# GEMV: limitations of the previous implementations



- Both implementations have a limitation: they access the matrix $A$ in column-major order, which is not cache friendly.
- We can improve the memory access pattern by changing the way we parallelize the computation.
- Instead of parallelizing over rows, we can **parallelize over columns**.




We express the matrix-vector product as a linear combination of the columns of $A$:

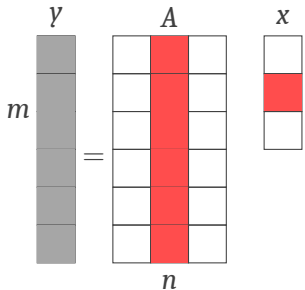
$$\mathbf{y} = x_1\mathbf{a}_{:,1} + x_2\mathbf{a}_{:,2} + \cdots + x_n\mathbf{a}_{:,n}$$

From an operative point of view, this means swapping the two loops in the naïve implementation.

# GEMV: limitations of the previous implementations



- Both implementations have a limitation: they access the matrix $A$ in column-major order, which is not cache friendly.
- We can improve the memory access pattern by changing the way we parallelize the computation.
- Instead of parallelizing over rows, we can **parallelize over columns**.




We express the matrix-vector product as a linear combination of the columns of $A$:

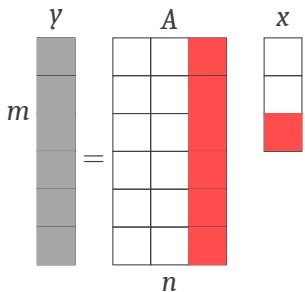
$$\mathbf{y} = x_1\mathbf{a}_{:,1} + x_2\mathbf{a}_{:,2} + \cdots + x_n\mathbf{a}_{:,n}$$

From an operative point of view, this means swapping the two loops in the naïve implementation.

# GEMV: limitations of the previous implementations



- Both implementations have a limitation: they access the matrix $A$ in column-major order, which is not cache friendly.
- We can improve the memory access pattern by changing the way we parallelize the computation.
- Instead of parallelizing over rows, we can **parallelize over columns**.




We express the matrix-vector product as a linear combination of the columns of $A$:

$$\mathbf{y} = x_1\mathbf{a}_{:,1} + x_2\mathbf{a}_{:,2} + \cdots + x_n\mathbf{a}_{:,n}$$

From an operative point of view, this means swapping the two loops in the naïve implementation.

# GEMV: parallelization over columns



- The matrix $A$ is stored in column-major order, this allows $A$ to be read sequentially, optimizing memory access.
- Each element of the vector $x$ is loaded into registers and reused efficiently.
- The vector $y$ is accessed in every iteration, but if its size is smaller than the cache capacity, it can be reused at the cache level.

```
y = beta * y ! Update y with beta * y
!$omp parallel do private(i) shared(A, x, alpha) reduction(+:y)
do i = 1, n
    call daxpy(m, alpha*x(i), A(1:m,i), 1, y, 1)
end do
!$omp end parallel do
```

# Limitation of the column-wise parallelization

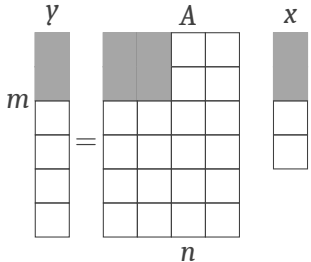


- The column-wise parallelization has a limitation:
  - The number of columns $n$ may be small compared to the number of available threads,
  - Array reductions are expensive: each thread needs to maintain a private copy of the output vector $y$ and then reduce them at the end of the computation,
  - For large $m$, maintaining multiple copies of $y$ can exceed cache capacity

# Limitation of the column-wise parallelization



- The column-wise parallelization has a limitation:
  - The number of columns $n$ may be small compared to the number of available threads,
  - Array reductions are expensive: each thread needs to maintain a private copy of the output vector $y$ and then reduce them at the end of the computation,
  - For large $m$, maintaining multiple copies of $y$ can exceed cache capacity
- To overcome this limitation, we can use a **blocked version** of the column-wise parallelization.

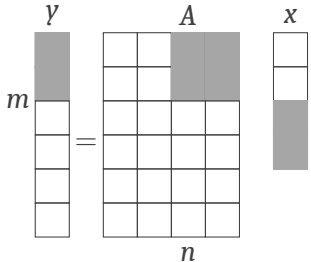



- This is a good approach if the matrix $A$ is large enough, and it allows us to take advantage of the cache hierarchy.
- The code is similar, but we need to add an outer loop that iterates over the blocks of the matrix $A$.
- This is an example of divide-and-conquer approach:
  - we divide the problem into smaller subproblems,
  - solve each subproblem by a sequential GEMV operation.

# Limitation of the column-wise parallelization



- The column-wise parallelization has a limitation:
  - The number of columns $n$ may be small compared to the number of available threads,
  - Array reductions are expensive: each thread needs to maintain a private copy of the output vector $y$ and then reduce them at the end of the computation,
  - For large $m$, maintaining multiple copies of $y$ can exceed cache capacity
- To overcome this limitation, we can use a **blocked version** of the column-wise parallelization.

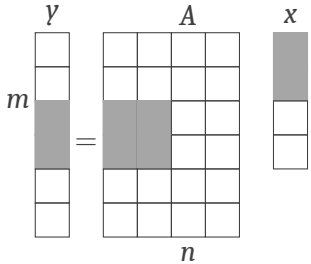



- This is a good approach if the matrix $A$ is large enough, and it allows us to take advantage of the cache hierarchy.
- The code is similar, but we need to add an outer loop that iterates over the blocks of the matrix $A$.
- This is an example of divide-and-conquer approach:
  - we divide the problem into smaller subproblems,
  - solve each subproblem by a sequential GEMV operation.

# Limitation of the column-wise parallelization



- The column-wise parallelization has a limitation:
  - The number of columns $n$ may be small compared to the number of available threads,
  - Array reductions are expensive: each thread needs to maintain a private copy of the output vector $y$ and then reduce them at the end of the computation,
  - For large $m$, maintaining multiple copies of $y$ can exceed cache capacity
- To overcome this limitation, we can use a **blocked version** of the column-wise parallelization.

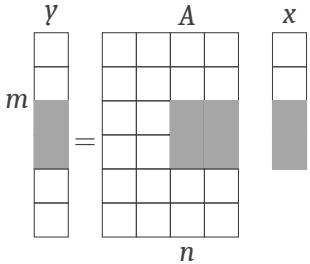



- This is a good approach if the matrix $A$ is large enough, and it allows us to take advantage of the cache hierarchy.
- The code is similar, but we need to add an outer loop that iterates over the blocks of the matrix $A$.
- This is an example of divide-and-conquer approach:
  - we divide the problem into smaller subproblems,
  - solve each subproblem by a sequential GEMV operation.

# Limitation of the column-wise parallelization



- The column-wise parallelization has a limitation:
  - The number of columns $n$ may be small compared to the number of available threads,
  - Array reductions are expensive: each thread needs to maintain a private copy of the output vector $y$ and then reduce them at the end of the computation,
  - For large $m$, maintaining multiple copies of $y$ can exceed cache capacity
- To overcome this limitation, we can use a **blocked version** of the column-wise parallelization.

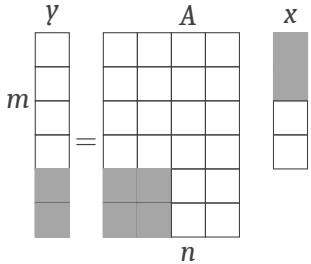



- This is a good approach if the matrix $A$ is large enough, and it allows us to take advantage of the cache hierarchy.
- The code is similar, but we need to add an outer loop that iterates over the blocks of the matrix $A$.
- This is an example of divide-and-conquer approach:
  - we divide the problem into smaller subproblems,
  - solve each subproblem by a sequential GEMV operation.

# Limitation of the column-wise parallelization



- The column-wise parallelization has a limitation:
  - The number of columns $n$ may be small compared to the number of available threads,
  - Array reductions are expensive: each thread needs to maintain a private copy of the output vector $y$ and then reduce them at the end of the computation,
  - For large $m$, maintaining multiple copies of $y$ can exceed cache capacity
- To overcome this limitation, we can use a **blocked version** of the column-wise parallelization.

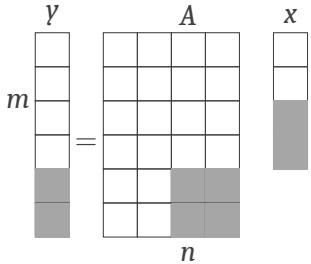



- This is a good approach if the matrix $A$ is large enough, and it allows us to take advantage of the cache hierarchy.
- The code is similar, but we need to add an outer loop that iterates over the blocks of the matrix $A$.
- This is an example of divide-and-conquer approach:
  - we divide the problem into smaller subproblems,
  - solve each subproblem by a sequential GEMV operation.

# Limitation of the column-wise parallelization



- The column-wise parallelization has a limitation:
  - The number of columns $n$ may be small compared to the number of available threads,
  - Array reductions are expensive: each thread needs to maintain a private copy of the output vector $y$ and then reduce them at the end of the computation,
  - For large $m$, maintaining multiple copies of $y$ can exceed cache capacity
- To overcome this limitation, we can use a **blocked version** of the column-wise parallelization.




- This is a good approach if the matrix $A$ is large enough, and it allows us to take advantage of the cache hierarchy.
- The code is similar, but we need to add an outer loop that iterates over the blocks of the matrix $A$.
- This is an example of divide-and-conquer approach:
  - we divide the problem into smaller subproblems,
  - solve each subproblem by a sequential GEMV operation.

# Divide and conquer GEMV code



```
! scale y by beta
y = beta * y
!$omp parallel default(none) &
!$omp    shared(A, x, y, m, n, lda, alpha, n_x_, n_y_) &
!$omp    private(i,j,ti,mb,nb,yloc)
allocate(yloc(m))
yloc = 0.0_real64
! Tile the i-j loops; collapse for better load balance
!$omp do collapse(2) schedule(static)
do i = 1, m, n_x_
    do j = 1, n, n_y_
        mb = min(n_x_, m - i + 1) ! handle edge tiles
```

# Divide and conquer GEMV code



```
        nb = min(n_y_, n - j + 1)
! perform the small GEMV into the thread--local yloc
        call dgemv('N', mb, nb, alpha, &
                   A(i, j), lda, &
                   x(j), 1, &
                   1.0_real64, yloc(i), 1)
    end do
end do
!$omp end do
! Safely accumulate thread--local yloc into global y
do ti = 1, m
    !$omp atomic
```

# Divide and conquer GEMV code



```
    y(ti) = y(ti) + yloc(ti)
end do
deallocate(yloc)
!$omp end parallel
```

- We need to choose **appropriate block sizes** `n_x_` and `n_y_`.
- We need to handle edge tiles when the matrix dimensions are not multiples of the block sizes.

# Description of the code

2 BLAS Level 2: GEMV continued

- We allocate a private copy of the output vector `yloc` for each thread.
- We tile the *i*-*j* loops, and we use the *!collapse(2)* clause to parallelize over the tiles.
- For each tile, we call a sequential `dgemv` to compute the partial result into the private `yloc`.
- Finally, we safely accumulate the private copies into the global output vector `y` using an *!atomic* operation.

**Compile**

We can add the module to our main project `objblas`:

```
add_library(objblas src/blas.f90 src/blasOMP.f90 src/gemvmod.f90)
target_link_libraries(objblas PUBLIC BLAS::BLAS OpenMP::OpenMP_Fortran)
```

# Roofline Model Refresher

## 2 BLAS Level 2: GEMV continued

We analyze GEMV

$$y = \alpha Ax + \beta y, \quad A \in \mathbb{R}^{m \times n},\ x \in \mathbb{R}^n,\ y \in \mathbb{R}^m$$

- Floating–point ops (precise):

$$\underbrace{2mn}_{Ax} + \underbrace{m}_{\beta y \text{ scale}} + \underbrace{m}_{\text{final add}} = 2mn + 2m \approx 2mn \quad (mn \gg m).$$

- Bytes moved (no reuse):

$$\underbrace{8mn}_{A} + \underbrace{8n}_{x} + \underbrace{16m}_{y \text{ read+write}} = 8mn + 8n + 16m.$$

- Operational intensity:

$$I = \frac{2mn + 2m}{8mn + 8n + 16m} \approx \frac{2mn}{8mn + 8n + 16m}.$$

# Roofline Model Refresher

## 2 BLAS Level 2: GEMV continued

We analyze GEMV

$$y = \alpha A x + \beta y, \quad A \in \mathbb{R}^{m \times n},\ x \in \mathbb{R}^n,\ y \in \mathbb{R}^m$$

Platform (Intel® Core™ i9-14900HX, approximate):

- Peak DP FLOPs $\approx$ 0.8–0.9 TFLOP/s
- Sustained memory bandwidth $\approx$ 89.6 GB/s
- Cache sizes:

```
L1d:                    896 KiB (24 instances)
L1i:                    1,3 MiB (24 instances)
L2:                     32 MiB (12 instances)
L3:                     36 MiB (1 instance)
```

Use roofline: Attainable GFLOP/s = $\min(\text{Peak}, \text{Bandwidth} \times I)$.

# Cache-Aware Problem Size Selection



Goal: choose (m,n) to exercise cache vs. memory bandwidth. Working set (no temporal reuse):

$$W(m,n) = 8mn + 8n + 16m \text{ bytes.}$$

Hierarchy (per core / shared, simplified):

- L1d: 32–48 KiB
- L2 (per P-core): 2 MiB; E-core cluster: 4 MiB
- LLC (L3): 36 MiB shared
- DRAM: 89.6 GB/s sustained

We classify regimes using full matrix footprint vs. cache levels (or effective tiled working set).

# Representative Matrix Sizes



**L1-working-set (fully fits when tiled)**:

- Example global size: m = n = 64
- Full A: $64^2 * 8B = 32KiB; x + yoverhead \approx 2$ KiB
- Entire working set $\approx$ 34 KiB (fits in 48 KiB L1d)

**L2/LLC-resident (fits in LLC, not L1)**:

- m = n = 2000
- A: 32 MB; x + y $\approx$ 0.032 MB
- Fits in 36 MB L3; stresses LLC bandwidth / latency

**DRAM-bound**:

- m = n = 20000
- A: 3.2 GB; exceeds LLC; compulsory DRAM traffic

# Tile Size Selection ($b_m, b_n$)

## $2\ BLAS Level 2: GEMV continued$

Tile footprint (data needed per tile GEMV assuming contiguous columns):

$$F(b_m, b_n) = 8\, b_m b_n + 8\, b_n + 16\, b_m \text{ bytes.}$$

Guidelines:

- Minimize capacity misses for A; keep (portion of) y hot.
- Reuse x entries across all rows of the tile.

# Tile Size Selection ($b_m, b_n$)

## $2\ BLASLevel2: GEMVcontinued$

**L1 tile**:

$$b_m = b_n = 32 \Rightarrow F \approx 8 \cdot 1024 + 8 \cdot 32 + 16 \cdot 32 \approx 8192 + 256 + 512 \approx 8.96 \text{ KiB}.$$

**L2 tile**:

$$b_m = b_n = 256 \Rightarrow F \approx 512 \text{ KiB} + \text{vector overhead} \approx 513 \text{ KiB}.$$

**Large / DRAM-stress tile**:

$$b_m = b_n = 512 \Rightarrow F \approx 2 \text{ MiB}.$$

Pick ($b_m$,$b_n$) so multiple thread–private $y$ tiles fit without eviction.

# Operational Intensity Examples



For square case $m = n$:

$$I(n) = \frac{2n^2 + 2n}{8n^2 + 8n + 16n} = \frac{2n^2 + 2n}{8n^2 + 24n}.$$

As $n \to \infty$: $I \to \frac{2}{8} = 0.25$ FLOP/Byte.
Examples (rounded):

- $n = 64$: $I \approx 0.24$
- $n = 2000$: $I \approx 0.25$
- $n = 20000$: $I \approx 0.25$

Conclusion: GEMV remains memory-bound on modern CPUs (low intensity). Optimization focuses on:

- Reducing data traffic (tiling, avoiding redundant y copies)
- Prefetch-friendly sequential column access
- Minimizing reduction overhead

# Performance Expectation vs. Roofline



Given peak $P$ and bandwidth $B$:

$$\text{Bound}_{\text{memory}} = B \cdot I, \quad \text{Bound}_{\text{compute}} = P.$$

With $I \approx 0.25$, $B = 89.6$ GB/s:

$$\text{Memory bound} \approx 22.4 \text{ GFLOP/s} \ll P.$$

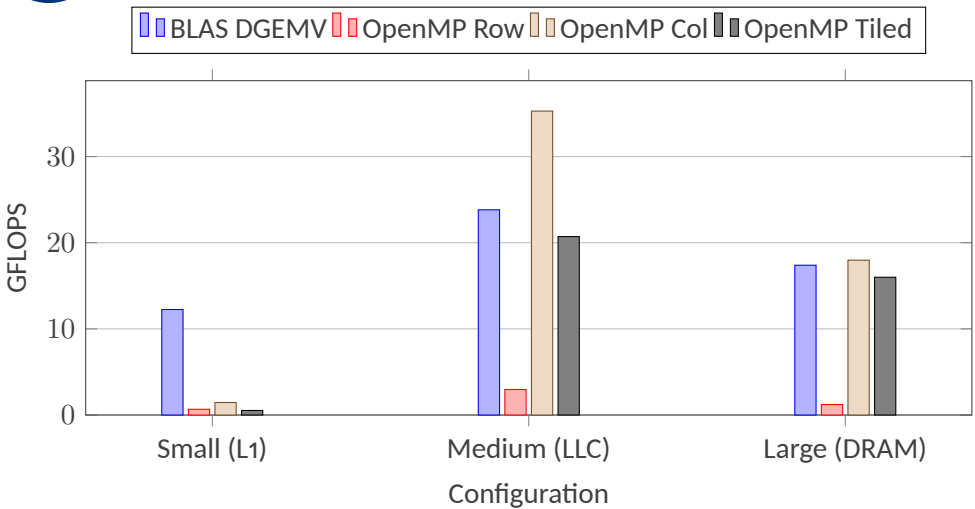
Thus:

- Optimized GEMV should approach 20–22 GFLOP/s.
- Large deviations imply poor locality or bandwidth saturation issues.
- Parallel scaling limited once bandwidth saturated.

Use roofline to validate improvement of column-wise and tiled implementations.

# Results and Observations

## 2 BLAS Level 2: GEMV continued



# Measured performance: consistency check

## 2 BLAS Level 2: GEMV continued

Matrix sizes used (matches earlier size slide):

$$\text{Small: } m = n = 64, \quad \text{Medium: } m = n = 2000, \quad \text{Large: } m = n = 20000.$$

Arithmetic intensity (AI) for square case

$$I(n) = \frac{2n^2 + 2n}{8n^2 + 24n} \xrightarrow[n\to\infty]{} 0.25 \text{ FLOP/Byte.}$$

Concrete values:

$$I(64) \approx 0.243,\ I(2000) \approx 0.250,\ I(20000) \approx 0.250.$$

DRAM roofline (bandwidth B = 89.6 GB/s):

$$R_{\text{mem}} = B \cdot I \approx 21.8\text{–}22.4 \text{ GFLOP/s (all sizes).}$$

Observed (from bar chart):

# Measured performance: consistency check



- Small (64): BLAS 13.3 GF/s (≈61% of DRAM roof); OMP row 0.68; OMP col 1.71; OMP tiled 0.48.
- Medium (2000): BLAS 23.4; OMP col 33.4; OMP tiled 20.5 (some values exceed DRAM roof).
- Large (20000): BLAS 17.1; OMP col 17.8; OMP tiled 16.2 (all below DRAM roof).

Flags:

- Column-wise medium case exceeds simple DRAM roof $\Rightarrow$ model underestimates attainable due to cache reuse.
- Small case far below roof $\Rightarrow$ kernel overhead $+$ underutilization dominate.
- Large case memory-bound as expected.

# Why some results exceed the DRAM roofline



The 22 GF/s bound assumes pure DRAM streaming. It is not a universal ceiling.

1. Cache residency  blocking:
   - Medium matrix (2000×2000): A = 32 MB fits in LLC (36 MB) ⇒ many accesses served from L3 after first pass.
   - Effective bandwidth becomes L3 (hundreds GB/s) not DRAM ⇒ higher feasible GFLOP/s.
2. Reuse pattern (column-wise outer-product):
   - Reuses each column of A sequentially with daxpy; x(j) stays in registers; y streamed once per column.
3. Simplified bytes-moved model overcounts:
   - Counts full read of A, x, y each repetition; ignores temporal locality of x and partial y residency.

# Why some results exceed the DRAM roofline



4. Library optimizations:
   - Vendor BLAS uses micro-kernels, software prefetching, packing improving cache-line reuse.
5. Timing / FLOP accounting alignment:
   - FLOPs formula correct (2mn + 2m), but if beta=0 path or fused operations shorten memory traffic, effective intensity rises.

Conclusion: Use a multi-ceiling roofline (L1/L2/L3/DRAM) to interpret results; DRAM roof alone is insufficient for cached regimes.

# Behavior of implementation variants

## 2 BLAS Level 2: GEMV continued

- Row-wise (ddot per row):
  - Very small per-thread work; function-call overhead; poor vector length; limited ILP → low GFLOP/s.
- Column-wise (daxpy per column / outer-product form):
  - Long contiguous daxpy operations → good SIMD utilization; favorable sequential column access; high cache reuse → best performance.
- Tiled version (current):
  - Private yloc per thread then atomic add for each element → m atomics per thread → heavy serialization.
  - Tile scheduling + allocation overhead further reduces throughput.
- BLAS:
  - Hand-tuned kernels, packing, minimized write-allocate misses, balanced threading.

**Key bottleneck now:** reduction scheme in tiled kernel (atomics) rather than GEMV arithmetic.

# OpenMP SIMD Directive



The *!$omp simd* directive instructs the compiler to vectorize the following loop using SIMD (Single Instruction, Multiple Data) instructions.

```
!$omp simd
do i = istart, iend
    y(i) = y(i) + alpha * A(i,j) * xj
end do
```

**Key features:**

- Enables automatic vectorization: multiple operations executed simultaneously
- Utilizes CPU vector registers (e.g., AVX-512 on modern Intel/AMD)
- No thread creation overhead — purely instruction-level parallelism
- Can combine with thread parallelism: *!$omp parallel do simd*

**Performance impact:** $4\times$–$8\times$ speedup typical with AVX2/AVX-512 for suitable loops.

# OpenMP SIMD Directive



The *!$omp simd* directive instructs the compiler to vectorize the following loop using SIMD (Single Instruction, Multiple Data) instructions.

```
!$omp simd
do i = istart, iend
    y(i) = y(i) + alpha * A(i,j) * xj
end do
```

**Requirements for effective SIMD:**

- Contiguous memory access (unit stride)
- No loop-carried dependencies
- Simple loop body (FMA-friendly operations)
- Alignment helps but not strictly required

**Performance impact:** $4\times$–$8\times$ speedup typical with AVX2/AVX-512 for suitable loops.

# Improved GEMV (row-partitioned, column-streaming, no atomics)



Goals: (1) column-major streaming of A, (2) reuse x(j) in registers, (3) avoid per-thread full y copies + atomic reduction.

```
subroutine gemv_openmp_blocked(m, n, alpha, A, lda, x, beta, y)
    use iso_fortran_env, only: real64
    use omp_lib
    implicit none
    integer, intent(in) :: m, n, lda
    real(real64), intent(in) :: alpha, beta
    real(real64), intent(in) :: A(lda,*), x(*)
    real(real64), intent(inout) :: y(*)
    integer :: i, j, tid, nth, istart, iend, base
    real(real64) :: xj
```

# Improved GEMV (row-partitioned, column-streaming, no atomics)



```
! Parallel scale y by beta (write each element once)
!$omp parallel default(none) shared(m,y,beta) private(i)
!$omp do schedule(static)
do i = 1, m
    y(i) = beta * y(i)
end do
!$omp end do
!$omp end parallel

!$omp parallel default(none) shared(m,n,A,lda,x,y,alpha) &
!$omp         private(tid,nth,istart,iend,j,i,xj,base)
tid = omp_get_thread_num()
nth = omp_get_num_threads()
```

# Improved GEMV (row-partitioned, column-streaming, no atomics)



```
! Contiguous row partition among threads
base   = (tid * m) / nth
istart = base + 1
iend   = ((tid + 1) * m) / nth

do j = 1, n
    xj = x(j)
    !$omp simd
    do i = istart, iend
        y(i) = y(i) + alpha * A(i,j) * xj
    end do
end do
```

# Improved GEMV (row-partitioned, column-streaming, no atomics)



```
    !$omp end parallel
end subroutine gemv_openmp_blocked
```

**Key differences vs previous tiled version:**

- Eliminates thread-private full copies (`yloc`) and expensive atomic accumulation.
- Each thread updates a disjoint contiguous slice of `y`: no write-sharing, no reduction.
- Outer loop over columns preserves sequential (contiguous) access to `A(:,j)` in column-major layout.
- Reuses scalar `x(j)` across whole row block (likely register-resident).

# Improved GEMV (row-partitioned, column-streaming, no atomics)



**Why this helps:**

- Reduction overhead removed → lower synchronization cost.
- Contiguous row blocks reduce TLB pressure and improve prefetch.
- `!$omp simd` on inner loop encourages vectorization across rows.

**Trade-offs / limits:**

- Parallelism tied to `m` (number of rows). If `m < threads` utilization poor.
- Load balance assumes uniform cost per row; acceptable for dense `A`.
- Still memory-bound; each `A(i,j)` touched exactly once; intensity capped at 0.25 FLOP/Byte.

# Improved GEMV (row-partitioned, column-streaming, no atomics)



**Possible refinements:**

- Hybrid blocking: partition rows, then process columns in chunks to keep `y` slice hot in cache.
- Use micro-kernel (unroll + FMA) for inner loop; block rows in multiples of SIMD width.
- If `n` very large, manual software prefetch on upcoming `A(i,j+pf)`.
- Replace scalar update with small packed panel (register block of `A`).

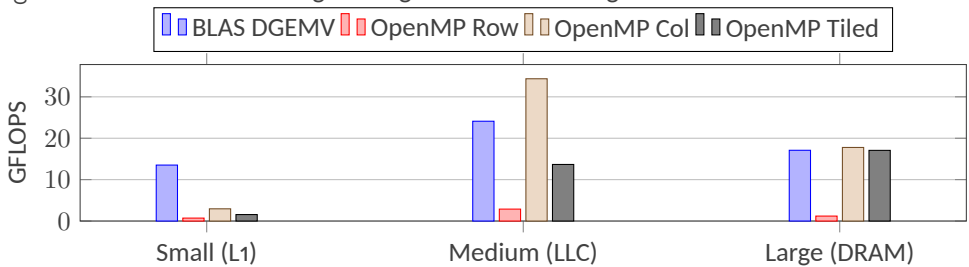
**When to prefer earlier tiled + reduction approach:**

- When `n` (columns) is huge and `m` moderately small: tiling over columns can improve `x` reuse granularity.
- When fusing multiple GEMVs sharing the same `y` (batch / multi-right-hand-side emulation).

# Improved GEMV (row-partitioned, column-streaming, no atomics)



**Summary:** This variant removes the dominant bottleneck (atomic reduction) while retaining cache-friendly column streaming over `A(:,j)` and register reuse of `x(j)`, approaching vendor `dgemv` behavior when `m` is large enough for thread scaling.



# TRSV: Triangular Solve with Vector



**Not all Level 2 BLAS operations parallelize well!**

- TRSV solves triangular systems $Ax = b$ where $A$ is:

**Lower triangular**

$$A = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

**Upper triangular**

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

Sequential dependency: must solve for $x_i$ before $x_{i+1}$ (forward) or $x_{i-1}$ (backward)

Options exist: iterative methods for sparse matrices, specialized parallel algorithms (covered later)

**Design principle:** Avoid triangular solves in parallel algorithms when possible!

# TRSV: Triangular Solve with Vector



```
subroutine fwd_subs(A, b, x)
implicit none
integer :: i, j, n
real(real64), intent(in) :: A(:,:)
real(real64), intent(in) :: b(:)
real(real64), intent(out) :: x(size(b))
n = size(b)
x(1) = b(1)/A(1,1)
do i = 2, n
    x(i) = b(i)
    do j = 1, i-1
        x(i) = x(i) - A(i,j)*x(j)
    end do
    x(i) = x(i)/A(i,i)
end do
end subroutine fwd_subs
```

Backward substitution algorithm.

# TRSV: Triangular Solve with Vector



```
subroutine bwd_subs(A, b, x)
implicit none
integer :: i, j, n
real(real64), intent(in) :: A(:,:)
real(real64), intent(in) :: b(:)
real(real64), intent(out) :: x(size(b))
n = size(b)
x(n) = b(n)/A(n,n)
do i = n-1, 1, -1
    x(i) = b(i)
    do j = i+1, n
        x(i) = x(i) - A(i,j)*x(j)
    end do
    x(i) = x(i)/A(i,i)
end do
end subroutine bwd_subs
```

Forward substitution algorithm.

# Table of Contents

3 BLAS Level 3


▶ BLAS Level 2: GEMV continued
Looking at the performance: roofline model
BLAS Level 2: TRSV

▶ BLAS Level 3
DGEMM with OpenMP
Tiled version
Tiled version with OpenMP

# BLAS Level 3: Overview



- Level 3 BLAS define operators that involve matrices
- Key operations:

  GEMM Computes $C = \alpha AB + \beta C$

  SYR2K Computes symmetric rank-2 update $C = \alpha AB^\top + \alpha BA^\top + \beta C$
- For $n \times n$ matrices: $O(n^3)$ arithmetic operations with $O(n^2)$ data accesses

Excellent arithmetic intensity compared to Level 1 and 2!

# GEMM: General Matrix Multiply

3 BLAS Level 3

## Mathematical formulation

$$C = \alpha AB + \beta C$$

- $A$ and $B$ can optionally be transposed or conjugated
- All three matrices may be strided
- Standard matrix multiplication: $\alpha = 1.0$, $\beta = 0.0$

**Element-wise formulation:**

$$C_{ij} = \alpha \sum_{l=1}^{k} A_{il} B_{lj} + \beta C_{ij}, \quad i = 1, \ldots, m, \; j = 1, \ldots, n$$

# GEMM: BLAS Interface

3 BLAS Level 3

```
call dgemm(transa, transb, m, n, k, alpha, A, lda,
           B, ldb, beta, C, ldc)
```

**Parameters:**

- `transa`, `transb`: transposition options for $A$ and $B$
- `m`, `n`, `k`: matrix dimensions
- `alpha`, `beta`: scalar multipliers
- `lda`, `ldb`, `ldc`: leading dimensions

# DGEMM: Example Usage

3 BLAS Level 3

```
program gemm_blass
    use iso_fortran_env, only: real64, output_unit, error_unit
    implicit none
    character(len=100) :: n_str, m_str, k_str
    integer :: n, m, k, info
    real(real64), allocatable :: a(:,:), b(:,:), c(:,:)
    ! Read from command line arguments n, m, k
    if (command_argument_count() < 3) then
        write(error_unit, *) "Usage: gemm_blass n m k"
        stop
    end if
    call get_command_argument(1, n_str)
    call get_command_argument(2, m_str)
    call get_command_argument(3, k_str)
    read(n_str, *) n
```

# DGEMM: Example Usage

3 BLAS Level 3

```
read(m_str, *) m
read(k_str, *) k
! Check if n, m, k are positive integers
if (n <= 0 .or. m <= 0 .or. k <= 0) then
    write(error_unit, '("n = ",I0,", m = ",I0,", k = ",I0," must be positive
    ↪  integers")') n,m,k
    stop
else
    write(output_unit, '("n = ",I0,", m = ",I0,", k = ",I0)') n,m,k
end if
! Allocate matrices
allocate(a(n,k), b(k,m), c(n,m), stat=info)
if (info /= 0) then
    write(error_unit, *) "Error allocating matrices"
    stop
```

# DGEMM: Example Usage

3 BLAS Level 3

```
    end if
    ! Initialize matrices
    call random_number(a)
    call random_number(b)
    call random_number(c)
    ! Perform matrix multiplication using BLAS
    call dgemm('N', 'N', n, m, k, 1.0d0, a, n, b, k, 1.0d0, c, n)
    ! Free matrices
    deallocate(a, b, c, stat=info)
    if (info /= 0) then
        write(error_unit, *) "Error deallocating matrices"
        stop
    end if
end program gemm_blass
```

# Naïve Implementation: $(i, j, l)$ ordering

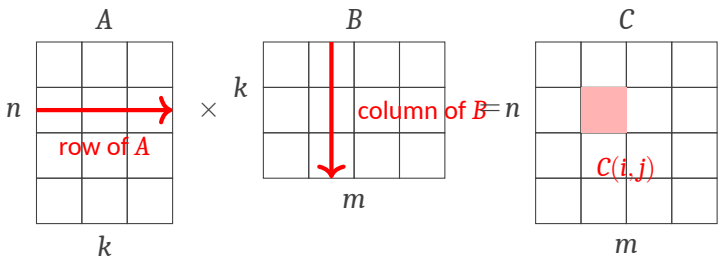


```
subroutine matmul_ijl(n,m,k,alpha,A,B,beta,C)
    use iso_fortran_env, only: real64
    implicit none
    integer, intent(in) :: n, m, k
    real(real64), intent(in) :: alpha, A(n,k), B(k,m), beta
    real(real64), intent(inout) :: C(n,m)
    integer :: i, j, l
    do i = 1, m
        do j = 1, n
            C(i,j) = beta * C(i,j)
            do l = 1, k
                C(i,j) = C(i,j) + alpha * A(i,l) * B(l,j)
            end do
        end do
    end do
end subroutine matmul_ijl
```

# Naïve Implementation: $(i, j, l)$ ordering



- </> Direct formula-to-code translation
- ! Poor memory access pattern for column-major storage



# Loop Reordering: $(j, i, l)$ ordering

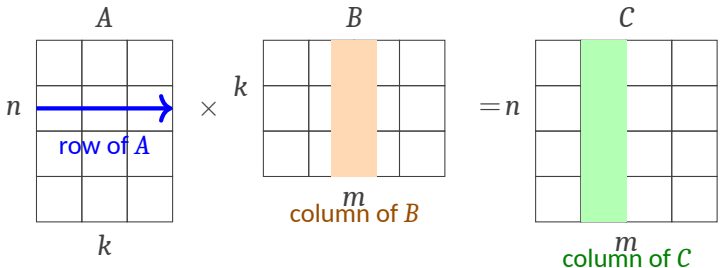
3 BLAS Level 3

```
subroutine matmul_jil(n,m,k,alpha,A,B,beta,C)
    use iso_fortran_env, only: real64
    implicit none
    integer, intent(in) :: n, m, k
    real(real64), intent(in) :: alpha, A(n,k), B(k,m), beta
    real(real64), intent(inout) :: C(n,m)
    integer :: i, j, l
    do j = 1, m
        do i = 1, n
            C(i,j) = beta * C(i,j)
            do l = 1, k
                C(i,j) = C(i,j) + alpha * A(i,l) * B(l,j)
            end do
        end do
    end do
end subroutine matmul_jil
```

# Loop Reordering: $(j, i, l)$ ordering



- Better: accesses $C$ in column-major order
- $\sim 2.6\times$ faster than $(i, j, l)$ ordering



# Optimal Loop Ordering: $(j, l, i)$

3 BLAS Level 3

**Why $(j, l, i)$ is best for Fortran column-major layout:**

- **Innermost loop ($i$):**
  - Reads $A(i, l)$ and $C(i, j)$ contiguously (unit stride)
  - Maximizes cache-line utilization
- **Middle loop ($l$):**
  - $B(l, j)$ constant, kept in register
  - Sequential access to columns of $A$ and $B$
- **Outer loop ($j$):**
  - Computes each column of $C$ in turn
  - Good spatial locality

**Key principle:** Loop order matters! Memory access pattern dominates performance.

# Optimal Implementation: $(j, l, i)$ ordering

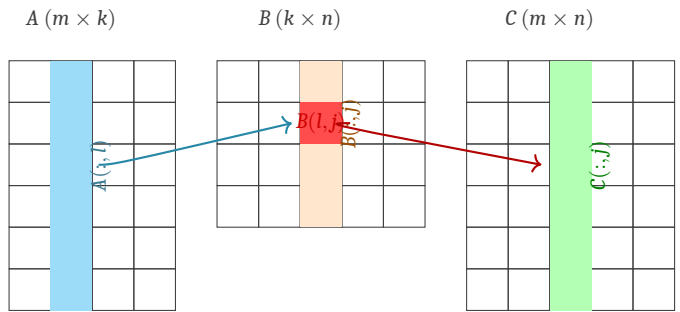
## 3 BLAS Level 3

```
subroutine matmul_jli(n,m,k,alpha,A,B,beta,C)
    use iso_fortran_env, only: real64
    implicit none
    integer, intent(in) :: n, m, k
    real(real64), intent(in) :: alpha, A(n,k), B(k,m), beta
    real(real64), intent(inout) :: C(n,m)
    integer :: i, j, l
    C = beta * C
    do j = 1, n
        do l = 1, k
            do i = 1, m
                C(i,j) = C(i,j) + alpha * A(i,l) * B(l,j)
            end do
        end do
    end do
end subroutine matmul_jli
```

# Optimal Implementation: $(j, l, i)$ ordering

3 BLAS Level 3

- Unit-stride access on all arrays
- $\sim$1.5$\times$ faster than $(j, i, l)$, $\sim$4$\times$ faster than $(i, j, l)$



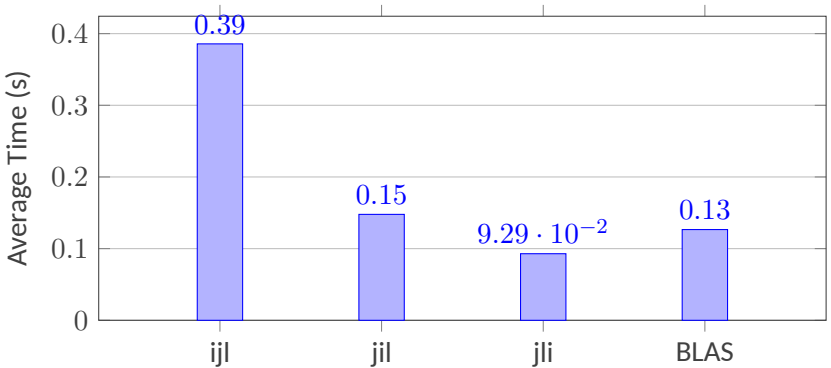

Outer: j

Middle: l

Inner: i

# Performance Comparison ($n = 10^3$, avg. over 100 runs)

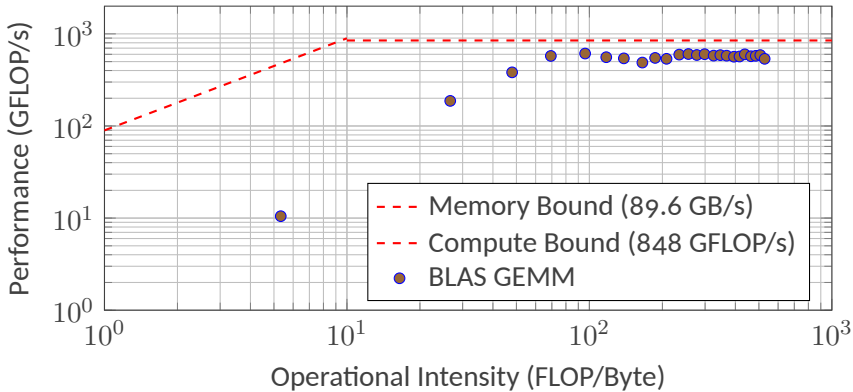
3 BLAS Level 3




- $(j, l, i)$ achieves 73% of OpenBLAS performances
- BLAS still faster: uses blocking, packing, micro-kernels

# Performance Comparison ($n = 10^3$, avg. over 100 runs)

3 BLAS Level 3




- High operational intensity ( 500 FLOP/Byte)
- Performance approaches compute-bound regime

# Parallel DGEMM with OpenMP



We now want to parallelize our DGEMM operation using OpenMP.

- A good starting point is to **start from** our **optimal sequential** implementation (loop order $(j, l, i)$)

# Parallel DGEMM with OpenMP



We now want to parallelize our DGEMM operation using OpenMP.

- A good starting point is to **start from** our **optimal sequential** implementation (loop order $(j, l, i)$)
- We just write the triple loop as before, and add OpenMP directives to parallelize the outer loops:

```
!$omp parallel default(none) shared(C,beta,m,n) private(i,j)
!$omp do schedule(static)
do j = 1, n
    !$omp simd
    do i = 1, m
        C(i,j) = beta * C(i,j)
    end do
end do
!$omp end do
```

⟨ Continues on the next slide ⟩

# Parallel DGEMM with OpenMP



We now want to parallelize our DGEMM operation using OpenMP.

- **!** We just write the triple loop as before, and add OpenMP directives to parallelize the outer loops:

```
!$omp do collapse(2) schedule(static) default(none) &
!$omp   shared(A,B,C,alpha,m,n,k) private(i,j,l,blj)
do j = 1, n
    do l = 1, k
        blj = alpha * B(l,j)
        !$omp simd
        do i = 1, m
            C(i,j) = C(i,j) + A(i,l) * blj
        end do
    end do
end do
!$omp end do
!$omp end parallel
```

# OpenMP SIMD Pragma



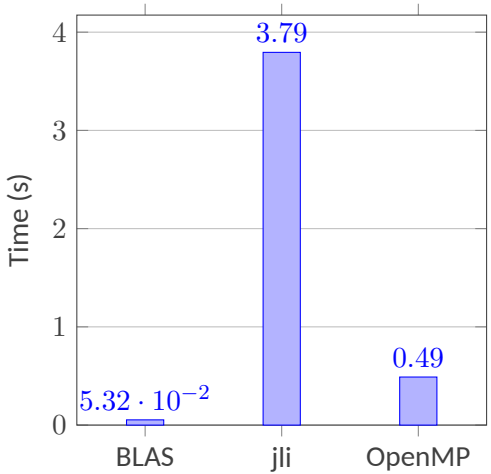
- The *!$omp simd* directive is used to instruct the compiler to vectorize the loop that follows it.
- This pragma allows the compiler to generate SIMD (Single Instruction, Multiple Data) instructions, which can process multiple data points in parallel.
- It is particularly useful in loops where iterations are independent, allowing for significant performance improvements on modern processors.
- Example usage:

```
!$omp simd
do i = 1, m
    C(i,j) = C(i,j) + A(i,l) * blj
end do
```

- In this example, the loop iterations can be executed simultaneously, leveraging the capabilities of the CPU's vector units.

# What did we gain?

## 3 BLAS Level 3




We used OpenMP to parallelize our optimal sequential DGEMM implementation

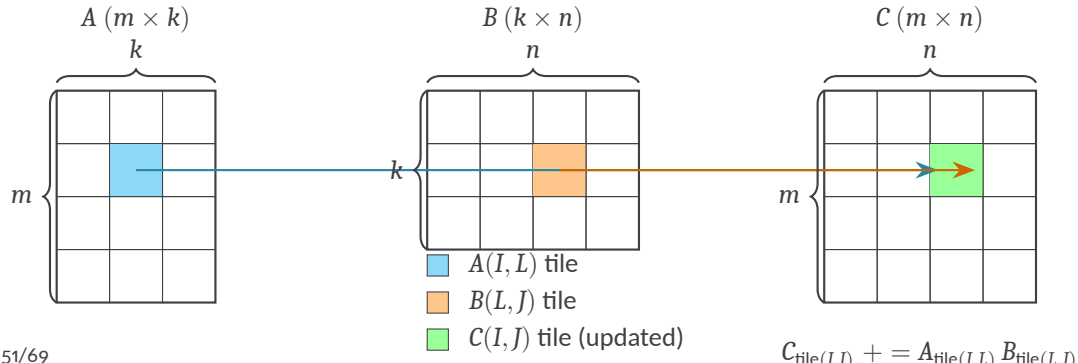
- Tested on matrices of size $n = 2560$, averaged over 20 runs, using 32 threads.
- ✓ Achieved a speedup of $\sim$7.75$\times$ over the sequential version.
- ✗ Still slower than vendor BLAS implementation.

# Tiled DGEMM

3 BLAS Level 3

To **improve cache utilization**, we can implement a tiled version of DGEMM.

- Divide the matrices into smaller sub-matrices (tiles) that fit into the cache.
- This reduces cache misses and improves data locality.



# Tiled DGEMM Implementation

3 BLAS Level 3

```
subroutine dgemm_tiled(m, n, k, alpha, A, lda, B, ldb, beta, C, ldc, tile_m,
↪  tile_n, tile_k)
   use iso_fortran_env, only: real64
   implicit none
   integer, intent(in) :: m, n, k, lda, ldb, ldc
   integer, intent(in), optional :: tile_m, tile_n, tile_k
   real(real64), intent(in) :: alpha, beta
   real(real64), intent(in) :: A(lda, *)
   real(real64), intent(in) :: B(ldb, *)
   real(real64), intent(inout) :: C(ldc, *)
   ! Local variables
   integer :: i, j, l, ii, jj, ll
   integer :: ts_m, ts_n, ts_k
   integer :: i_end, j_end, l_end
   real(real64) :: temp
```

# Tiled DGEMM Implementation



```
! Set tile sizes (default 64)
ts_m = 64
ts_n = 64
ts_k = 64
if (present(tile_m)) ts_m = tile_m
if (present(tile_n)) ts_n = tile_n
if (present(tile_k)) ts_k = tile_k

! Scale C by beta
do j = 1, n
   do i = 1, m
      C(i,j) = beta * C(i,j)
   end do
end do
```

# Tiled DGEMM Implementation

3 BLAS Level 3

```
! Tiled matrix multiplication with non-square tiles
do jj = 1, n, ts_n
   j_end = min(jj + ts_n - 1, n)
   do ll = 1, k, ts_k
      l_end = min(ll + ts_k - 1, k)
      do ii = 1, m, ts_m
         i_end = min(ii + ts_m - 1, m)

         ! Multiply tile
         do j = jj, j_end
            do l = ll, l_end
               temp = alpha * B(l,j)
               do i = ii, i_end
                  C(i,j) = C(i,j) + A(i,l) * temp
```

# Tiled DGEMM Implementation

3 BLAS Level 3

```
                end do
            end do
        end do

    end do
  end do
end do

end subroutine dgemm_tiled
```

# Tiled DGEMM: Implementation Notes



**Key design choices:**

- **Tile size selection:** Default $64 \times 64 \times 64$ tiles
  - Balances L1/L2 cache capacity vs. parallelism granularity
  - Overridable via optional arguments for tuning
- **Edge handling:** `min()` ensures correct partial tiles at boundaries
- **Loop nest structure:**
  - Outer 3 loops (`jj, ll, ii`): tile iteration
  - Inner 3 loops (`j, l, i`): computation within tile
  - Maintains optimal `(j,l,i)` ordering for cache-friendly access
- **Beta scaling:** Applied once before tiling to avoid redundant operations
- **Temporal reuse:** Each tile of $C$ accumulates contributions from multiple $A/B$ tile pairs, improving cache hit rate

**Parallelization opportunity:** Outer tile loops are independent.

# Exercise: tuning tiled DGEMM



An exercise for you to try at home!

1. Write a driver program to test the performance of the tiled DGEMM implementation.
2. Experiment with different tile sizes (e.g., 32, 64, 128) to see how they affect performance.
3. Measure execution time and compute performance (GFLOP/s) for various matrix sizes (e.g., 512, 1024, 2048).
4. Compare the performance of your tiled DGEMM with the non-tiled version and with a vendor BLAS implementation.
5. Analyze the results and determine the optimal tile size for your specific hardware.

# Tiled DGEMM with OpenMP



We can further enhance our tiled DGEMM implementation by adding OpenMP directives to parallelize the outer tile loops.

- This allows multiple tiles to be computed simultaneously, leveraging multi-core processors.
- We add OpenMP pragmas to the outer loops iterating over tiles.

  To hope for good performance, make sure to choose
  - tile sizes that provide enough work per thread to amortize threading overhead,
  - tile sizes that divide the matrix dimensions exactly to avoid load imbalance.
  - **If not**, we would need to implement dynamic scheduling or handle edge cases carefully to avoid loss of performance.

# Tiled DGEMM with OpenMP: implementation



```
subroutine dgemm_tiled_openmp(m, n, k, alpha, A, lda, B, ldb, beta, C, ldc, &
    tile_m, tile_n, tile_k)
    use iso_fortran_env, only: real64
    implicit none

    integer, intent(in) :: m, n, k, lda, ldb, ldc
    integer, intent(in), optional :: tile_m, tile_n, tile_k
    real(real64), intent(in) :: alpha, beta
    real(real64), intent(in) :: A(lda, *)
    real(real64), intent(in) :: B(ldb, *)
    real(real64), intent(inout) :: C(ldc, *)

    integer :: ts_m, ts_n, ts_k
    integer :: ii, jj, ll
    integer :: i, j, l
```

# Tiled DGEMM with OpenMP: implementation

3 BLAS Level 3

```
integer :: i_end, j_end, l_end
integer :: ib, jb
real(real64) :: tmp

! ---- MAXIMUM tile sizes (adjust safely for your CPU cache) ----
integer, parameter :: MAX_TS_M = 128
integer, parameter :: MAX_TS_N = 128

! Local tile buffer, fixed size (thread-private due to OpenMP)
real(real64) :: Cbuf(MAX_TS_M, MAX_TS_N)

! Default tile sizes
ts_m = 64
ts_n = 64
ts_k = 64
```

# Tiled DGEMM with OpenMP: implementation

3 BLAS Level 3

```
if (present(tile_m)) ts_m = min(tile_m, MAX_TS_M)
if (present(tile_n)) ts_n = min(tile_n, MAX_TS_N)
if (present(tile_k)) ts_k = tile_k

!$omp parallel default(none) &
!$omp shared(m,n,k,ts_m,ts_n,ts_k,A,B,C,alpha,beta,lda,ldb,ldc) &
!$omp private(ii,jj,ll,i,j,l,i_end,j_end,l_end,ib,jb,Cbuf,tmp)
!$omp do collapse(2) schedule(static)
do jj = 1, n, ts_n
   do ii = 1, m, ts_m
      ! Work tile bounds
      i_end = min(ii + ts_m - 1, m)
      j_end = min(jj + ts_n - 1, n)

      ib = i_end - ii + 1   ! actual tile height
```

# Tiled DGEMM with OpenMP: implementation

3 BLAS Level 3

```
jb = j_end - jj + 1    ! actual tile width
! -------------------------------
! Load and scale C tile: Cbuf = beta * C
! -------------------------------
do j = 1, jb
   do i = 1, ib
      Cbuf(i, j) = beta * C(ii + i - 1, jj + j - 1)
   end do
end do
! -------------------------------
! Accumulate over all K tiles
! -------------------------------
do ll = 1, k, ts_k
   l_end = min(ll + ts_k - 1, k)
```

# Tiled DGEMM with OpenMP: implementation



```
   do l = ll, l_end
      do j = 1, jb
         ! scalar needed for whole column
         tmp = alpha * B(l, jj + j - 1)

         !$omp simd
         do i = 1, ib
            Cbuf(i, j) = Cbuf(i, j) + A(ii + i - 1, l) * tmp
         end do
      end do
   end do
end do
! -------------------------------
! Write tile back to C
! -------------------------------
```

# Tiled DGEMM with OpenMP: implementation



```
            do j = 1, jb
               do i = 1, ib
                  C(ii + i - 1, jj + j - 1) = Cbuf(i, j)
               end do
            end do

         end do
      end do
      !$omp end do
      !$omp end parallel

   end subroutine dgemm_tiled_openmp
```

# Tiled DGEMM with OpenMP: Implementation Notes

3 BLAS Level 3

**Key improvements over sequential tiled version:**

- **Thread-private tile buffer:** Each thread allocates `Cbuf(MAX_TS_M, MAX_TS_N)` on its stack
  - Eliminates write conflicts to shared `C` during accumulation
  - Local buffer has better cache affinity than scattered `C` updates
- **Collapsed parallelization:** *`!$omp do collapse(2)`* over `(jj, ii)` tile indices
  - Increases parallel grain count: `(n/ts_n) * (m/ts_m)` independent tasks
  - Better load balance when `n` or `m` is small relative to thread count
- **Three-stage tile computation:**
  1. **Load & scale:** `Cbuf = beta * C(tile)`
  2. **Accumulate:** Loop over `ll` (K-tiles), perform `Cbuf += alpha * A(tile) * B(tile)`
  3. **Write-back:** `C(tile) = Cbuf`

  Minimizes memory traffic to global `C`: two passes instead of `O(k/ts_k)` read-modify-writes
- **Innermost SIMD:** *`!$omp simd`* on row loop within tile maximizes ILP

# Tiled DGEMM with OpenMP: Trade-offs

3 BLAS Level 3

**Memory considerations:**

- **Stack pressure:** Each thread needs `8 * MAX_TS_M * MAX_TS_N` bytes
  - Example: `MAX_TS_M = MAX_TS_N = 128` $\Rightarrow$ 128 KB/thread
  - 32 threads $\Rightarrow$ 4 MB total (acceptable on modern systems)
  - May need `ulimit -s unlimited` or adjust stack size limits
- **Cache optimization:** `Cbuf` stays hot in L1/L2 during K-loop accumulation
  - Temporal reuse: each `Cbuf` element updated `k/ts_k` times without eviction
  - Reduces `C` memory traffic by factor of `k/ts_k`

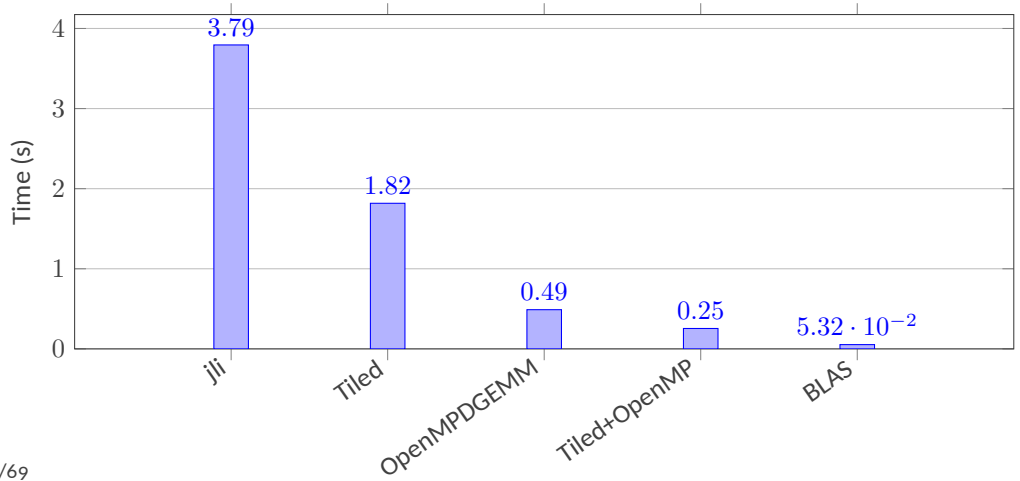
**Performance tuning:**

- Choose `ts_m`, `ts_n` to balance:
  - Tile buffer fits in L2 cache (`ts_m * ts_n * 8` bytes $\lesssim$ L2 size)
  - Enough tiles for good thread utilization: `(m/ts_m)*(n/ts_n) >= num_threads * 4`
- `ts_k` primarily affects `A`/`B` reuse, less critical than `ts_m`/`ts_n`

# Performance Comparison: Tiled vs. Tiled+OpenMP

## 3 BLAS Level 3

Test configuration: `m = n = k =` 2560, 32 threads, tile sizes $64 \times 64 \times 64$



# Final remarks and conclusions

3 BLAS Level 3

**Observations:**

- ✔ OpenMP tiling achieves $\sim$7.1$\times$ speedup over sequential tiled version
- ✔ Reaches $\sim$21% of vendor BLAS performance (reasonable for educational implementation)
- ! Remaining gap due to:

  **Register blocking:** further subdividing tiles to fit in CPU registers

  **Micro-kernels:** hand-optimized inner loops using assembly or intrinsics

  **Prefetching:** software prefetch instructions to hide memory latency

  **Packing:** reorganizing data in contiguous buffers to improve cache access patterns

# Summary of Lecture 6

4 Conclusions

- We completed our study of the DGEMV operation.
- We explored the implementation of DGEMM from basic triple-loop to optimized tiled and parallel versions.
- We analyzed performance using the Roofline model, highlighting the importance of operational intensity.
- We discussed key optimization techniques such as loop ordering, tiling, and OpenMP parallelization.
- We provided a foundation for further exploration into high-performance computing and numerical linear algebra.

**Next up:** start pushing outside the frontier of a single CPU: distributed memory parallelism with MPI!

# High Performance Linear Algebra

## Lecture 7: Distributed Memory Machines and MPI

Ph.D. program in High Performance Scientific Computing

**Fabio Durastante** **Pasqua D'Ambra** **Salvatore Filippone**

January 12, 2026 — 14.00:16.00

Dipartimento
di Matematica
Università di Pisa

# Before the time-skip



- We have seen in the first half of the course:
  - The basic concepts of parallel computing
  - The main architectures for parallel computing
  - The main programming models for parallel computing
- We have introduced the BLAS libraries for linear algebra computations
  - We have seen the Level 1, Level 2 and Level 3 BLAS operations
  - We have discussed the performance of BLAS operations on shared memory architectures
  - Explored the OpenMP programming model
  - Employed the roofline model to analyze the performance of BLAS operations

# Before the time-skip



- We have seen in the first half of the course:
  - The basic concepts of parallel computing
  - The main architectures for parallel computing
  - The main programming models for parallel computing
- We have introduced the BLAS libraries for linear algebra computations
  - We have seen the Level 1, Level 2 and Level 3 BLAS operations
  - We have discussed the performance of BLAS operations on shared memory architectures
  - Explored the OpenMP programming model
  - Employed the roofline model to analyze the performance of BLAS operations

And now *to boldly go out of shared memory architectures*...

# Table of Contents

2 Distributed memory machines


- ▶ Distributed memory machines
- ▶ How do we program such a machine?
  - An MPI hello world program
    - Finding MPI via CMake
    - The fallacies of distributed computing
  - Working on a cluster/shared machine with MPI
- ▶ The Toeplitz cluster at DMPISA
    - Partitions on Toeplitz
    - Software

# Distributed memory machines

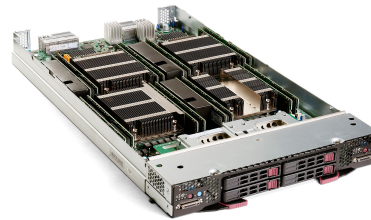


- In distributed memory machines, each processor has its own private memory
- Processors communicate by passing messages through a network
- Examples of distributed memory machines:
  - Clusters of workstations connected by a high-speed network,
  - Massively parallel supercomputers (e.g., the machines of the TOP500 list)
- Programming models for distributed memory machines:
  - **M**essage **P**assing **I**nterface (MPI)
  - **P**artitioned **G**lobal **A**ddress **S**pace (PGAS) languages (e.g., Coarray Fortran)

# Nodes

## 2 Distributed memory machines

- A distributed memory machine is composed of multiple nodes
- Each node contains one or more processors (CPUs) and its own private memory
- Nodes are connected by a high-speed network that allows them to communicate with each other
- Each node can run one or more processes that execute the parallel program
- Each node can have one or more accelerators (e.g., GPUs) to offload computations



# General information on networks

## 2 Distributed memory machines

- In distributed memory machines, communication between processors occurs through a network
- Network performance is characterized by two key parameters:
  - Latency ($\alpha$): time to send a message of zero length
  - Bandwidth ($\beta$): inverse of time to send one byte of data
- The time to send a message of size $n$ bytes is modeled as:

$$T_{\text{comm}}(n) = \alpha + \frac{n}{\beta}$$

- **Latency** is typically measured in microseconds (µs)
- **Bandwidth** is typically measured in gigabits per second (Gbit $s^{-1}$)

# What determines these parameters?



- $\alpha$ Depends almost entirely on the operating system stack. To minimize latency: avoid TCP/IP protocols
- $\beta$ Depends on both the operating system stack *and* the physical communication device hardware

**Key insight:** Low latency requires careful software optimization, while high bandwidth depends on specialized hardware (e.g., InfiniBand).

# Some examples from the market



InfiniBand **H**igh **D**ynamic **R**ange (HDR) 2018:

- Latency: $<0.6\,\mu s$
- Bandwidth: $200\,\text{Gbit s}^{-1}$

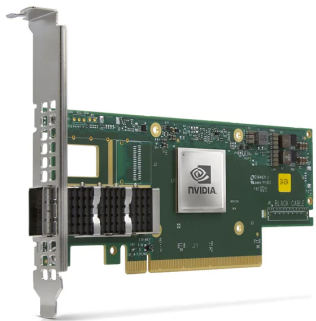
The *InfiniBand* technology is widely used in high-performance computing clusters, and it is the network technology used in many TOP500 supercomputers.

# Some examples from the market



InfiniBand **H**igh **D**ynamic **R**ange (HDR) 2018:

- Latency: $< 0.6\,\mu s$
- Bandwidth: $200\,\text{Gbit}\,\text{s}^{-1}$



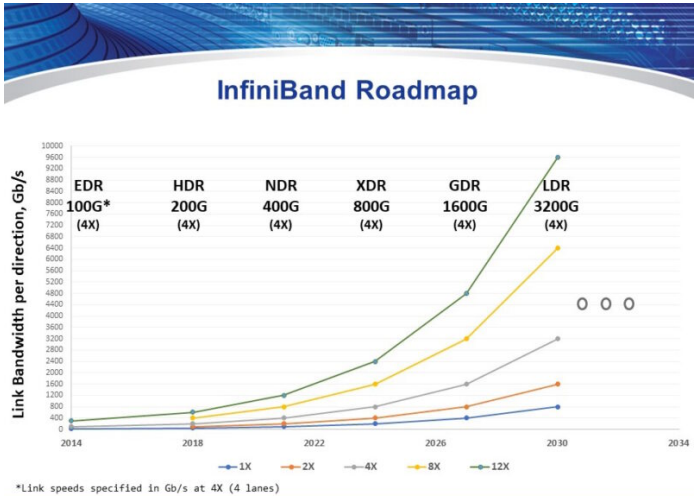

There are different vendors for Infiniband network adapters, e.g., Mellanox (now part of NVIDIA):

- NVIDIA/Mellanox Compatible AOC 20m InfiniBand HDR Active Optical Cable: 910 USD,
- NVIDIA/Mellanox MCX653105A-HDAT-SP ConnectX® -6 InfiniBand Adapter Card, HDR/200G: 1069 USD,
- NVIDIA MQM8700-HS2F Quantum HDR InfiniBand Switch, 40 x HDR QSFP56 Ports: 17538 USD.

# Some examples from the market

2 Distributed memory machines

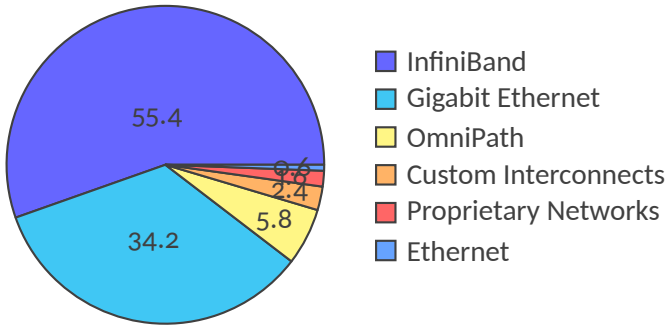
Newer InfiniBand standard exists, but are not yet widely used in HPC clusters



# The TOP500 supercomputer situation

## 2 Distributed memory machines

From the November 2025 TOP500 list[1], the distribution of interconnects used in the top 500 supercomputers is as follows:




[1] https://www.top500.org/lists/top500/2025/11/

# Should we care about network topology?



In large distributed memory machines, the network topology can have a significant impact on the performance of parallel applications.

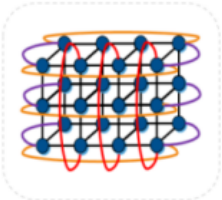
- Different topologies have different characteristics in terms of latency, bandwidth, and scalability.
- Common network topologies include:
  - Fat-tree,
  - Torus,
  - Hypercube,
  - Dragonfly.
- The choice of network topology can affect the performance of collective communication operations.

# Should we care about network topology?



In large distributed memory machines, the network topology can have a significant impact on the performance of parallel applications.

- Different topologies have different characteristics in terms of latency, bandwidth, and scalability.
- Common network topologies include:
  - Fat-tree,
  - Torus,
  - Hypercube,
  - Dragonfly.
- The choice of network topology can affect the performance of collective communication operations.

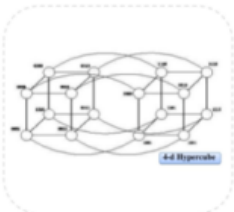


# Should we care about network topology?

## 2 Distributed memory machines

In large distributed memory machines, the network topology can have a significant impact on the performance of parallel applications.

- Different topologies have different characteristics in terms of latency, bandwidth, and scalability.
- Common network topologies include:
  - Fat-tree,
  - Torus,
  - Hypercube,
  - Dragonfly.
- The choice of network topology can affect the performance of collective communication operations.



# Should we care about network topology?



In large distributed memory machines, the network topology can have a significant impact on the performance of parallel applications.

- Different topologies have different characteristics in terms of latency, bandwidth, and scalability.
- Common network topologies include:
  - Fat-tree,
  - Torus,
  - Hypercube,
  - Dragonfly.
- The choice of network topology can affect the performance of collective communication operations.

# Should we care about network topology?

## 2 Distributed memory machines

In large distributed memory machines, the network topology can have a significant impact on the performance of parallel applications.

- Different topologies have different characteristics in terms of latency, bandwidth, and scalability.
- Common network topologies
- The choice of network topology can affect the performance of collective communication operations.


Fat Tree


Dragonfly


Torus


Hypercube

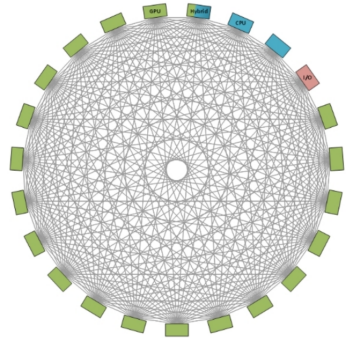
On the **implementation side**, we usually do not have to care, and think of the network as a black box with given latency and bandwidth.

# An example: the Leonardo supercomputer *Dragonfly+*



- At top level, there are 23 cells fully connected in a dragonfly topology,
- Locally, intra-cell routers are organized in a **bipartite graph**
  - in which a *first tier* is directly connected to servers (leaf routers)
  - and a *second tier* (spine routers) is equally provisioned with down-links.

See the full description in: Turisini, Cestarti, Amati. “LEONARDO A Pan-European Pre-Exascale Supercomputer for HPC and AI applications”, Vol. 9 No. 1 (2024): Journal of large-scale research facilities.




*Dragonfly* topology of the internal network. Green is used for Booster cells, blue for Data-Centric cells, pink for the I/O.

# Leonardo supercomputer: Network specifications



- Network Technology: 200 Gbps InfiniBand HDR (Mellanox/NVIDIA)
  - Switch latency: 90 nanoseconds port-to-port
  - Message rate: 390 million messages/second per port
  - Total switches: **823 QM8700 units**
- Node-level adapter: ConnectX-6 (CX6) card
  - 200 million messages per second capacity
  - $600$ ns latency per Network Interface Card (NIC)
  - PCIe Gen4 on 32 lanes
- Maximum inter-node latency: $3$ µs
  - Dominated by NIC delays: $1.2$ µs
  - Fiber segments: $1$ m (NIC to leaf), $5$ m (leaf to spine), $20$ m (spine to spine)
- External connectivity: 4 gateway routers with Ethernet-InfiniBand translators
  - $1.6$ Tbit $s^{-1}$ per unit, $6.4$ Tbit $s^{-1}$ aggregated

# Table of Contents

3 How do we program such a machine?


- Distributed memory machines
- How do we program such a machine?
  - An MPI hello world program
    - Finding MPI via CMake
    - The fallacies of distributed computing
  - Working on a cluster/shared machine with MPI
- The Toeplitz cluster at DMPISA
  - Partitions on Toeplitz
  - Software

# Message Passing Interface (MPI)

3 How do we program such a machine?

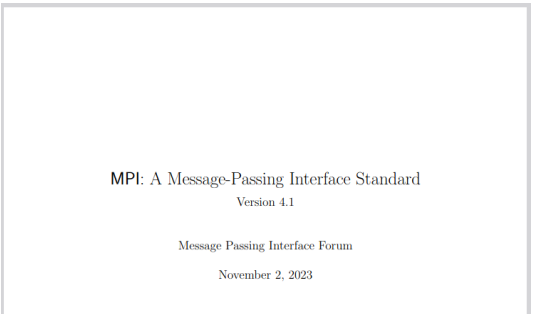
- The **M**essage **P**assing **I**nterface (MPI) is the de facto standard for programming distributed memory machines
- MPI provides a set of functions for:
  - Point-to-point communication (send/receive messages between two processes)
  - Collective communication (broadcast, scatter, gather, reduce, etc.)
  - Process management (creating and terminating processes)
- MPI is implemented as a library that can be used with different programming languages (C, C++, Fortran)
- There are several implementations of MPI
  - MPICH `https://www.mpich.org/`
  - OpenMPI `https://www.open-mpi.org/`
  - MVAPICH `https://mvapich.cse.ohio-state.edu/`
- The current stable version is 4.1, and work is underway to define version 5.0.

# What is exactly MPI?

## 3 How do we program such a machine?

> *"MPI (Message-Passing Interface) is a message-passing library interface specification."*

All parts of this definition are significant.
See: https://www.mpi-forum.org/docs/


MPI: A Message-Passing Interface Standard
Version 4.1

Message Passing Interface Forum

November 2, 2023

# MPI: Key aspects

3 How do we program such a machine?

- **Message-passing model**: Data moves from the address space of one process to another through cooperative operations
- **Specification, not implementation**: Multiple implementations exist (MPICH, OpenMPI, MVAPICH)
- **Library interface**: Operations expressed as functions, subroutines, or methods in C and Fortran
- **Extensions**: Collective operations, remote-memory access, dynamic process creation, parallel I/O

# MPI basic concepts

## 3 How do we program such a machine?

An MPI program is composed of multiple processes that run concurrently on different processors

- Each process has a unique identifier called rank
- The total number of processes is called the size of the communicator
- The main communicator is called `MPI_COMM_WORLD`, which includes all processes
- Processes can communicate by sending and receiving messages using MPI functions

Why message passing?

- Each process has its **own private address space**
- Other processes **cannot access data directly**
- Processes must **cooperate** to exchange data through messages

# An MPI hello world program in Fortran



Let us write a simple MPI program that prints “Hello, World!” from each process, we write a file called `mpi_hello.f90` with the following content:

```
program mpi_hello
    use mpi
    use iso_fortran_env, only: output_unit
    implicit none
    integer :: ierr, rank, size

    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
    write(output_unit, *) 'Hello from process', rank, 'of', size
    call MPI_Finalize(ierr)
end program mpi_hello
```

# Let us look at it line by line



```
use mpi
```

- This line includes the MPI module, which contains the definitions of MPI functions and constants

```
call MPI_Init(ierr)
```

- This line initializes the MPI environment, must be called before any other MPI function

```
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
```

- This line gets the rank of the current process in the communicator

```
call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
```

- This line gets the total number of processes in the communicator

# Let us look at it line by line

3 How do we program such a machine?

```
write(output_unit, *) 'Hello from process', rank, 'of', size
```

- This line prints a message from each process, including its rank and the total size

```
call MPI_Finalize(ierr)
```

- This line finalizes the MPI environment, must be called at the end of the program

- The program declares three integer variables:
  - `rank`: to store the rank of the current process
  - `size`: to store the total number of processes
  - `ierr`: to store the error code returned by MPI functions

  The variable `ierr` is used to capture error codes from MPI functions, and should be used for error handling in a production code.

# Compiling and running the MPI program



To compile the MPI program, we need to have installed an MPI implementation (e.g., MPICH, OpenMPI, MVAPICH).

On an Ubuntu system, we can install OpenMPI with the following command:

```
sudo apt-get install libopenmpi-dev openmpi-bin openmpi-common
```

If you are using Spack to manage your software, you can install OpenMPI with:

```
spack install openmpi
spack load openmpi
```

In a system **with multiple MPI implementations**, make sure to load the correct one using `module load` or `spack load`.

# Compiling and running the MPI program



All these implementations provide a wrapper compiler that simplifies the compilation process by automatically including the necessary **MPI headers** and **linking** against the **MPI libraries**.

For example, using OpenMPI, we can compile the program with the following command:

```
mpifort -o mpi_hello mpi_hello.f90
```

If you are using MPICH or MVAPICH, the command is the same.

# Compiling and running the MPI program



If you want to investigate what the wrapper compiler is doing behind the scenes:

```
mpifort --show:me
```

This will display the actual compilation command, e.g., on my machine it shows:

```
/usr/bin/gfortran -I/opt/spack/opt/spack/linux-skylake/openmpi-4.1.8/include
↪  -I/opt/spack/opt/spack/linux-skylake/openmpi-4.1.8/lib
↪  -L/opt/spack/opt/spack/linux-skylake/openmpi-4.1.8/lib
↪  -L/opt/spack/opt/spack/linux-skylake/hwloc-2.12.2/lib
↪  -L/opt/spack/opt/spack/linux-skylake/libevent-2.1.12/lib -Wl,-rpath
↪  -Wl,/opt/spack/opt/spack/linux-skylake/openmpi-4.1.8/lib -Wl,-rpath
↪  -Wl,/opt/spack/opt/spack/linux-skylake/hwloc-2.12.2/lib -Wl,-rpath
↪  -Wl,/opt/spack/opt/spack/linux-skylake/libevent-2.1.12/lib -lmpi_usempif08
↪  -lmpi_usempi_ignore_tkr -lmpi_mpifh -lmpi
```

showing that it is using `gfortran` as the underlying compiler, and including the necessary MPI headers and libraries from Spack.

# Compiling and running the MPI program



To run the MPI program, we use the `mpirun` or `mpiexec` command, specifying the number of processes with the `-n` option:

```
mpirun -n 4 ./mpi_hello
```

This command runs the `mpi_hello` program with 4 processes, and we should see output similar to:

```
Hello from process           1 of           4
Hello from process           2 of           4
Hello from process           0 of           4
Hello from process           3 of           4
```

Note that **the order of the output may vary**, as the processes run concurrently.

# Finding MPI via CMake



To find and use MPI in a CMake project, we can use the `FindMPI` module provided by CMake, i.e., we can add the following lines to our `CMakeLists.txt` file:

```
find_package(MPI REQUIRED COMPONENTS Fortran)
```

This command searches for an installed MPI implementation and sets the necessary variables to use MPI in our project.

To compile an MPI program, we need to link against the MPI libraries and include the MPI headers. We can do this by adding the following lines to our `CMakeLists.txt` file:

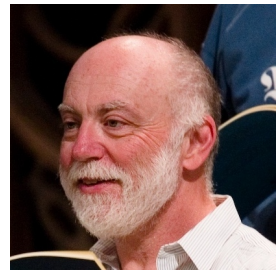
```
add_executable(mpi_hello mpi_hello.f90)
target_link_libraries(mpi_hello MPI::MPI_Fortran)
```

# The fallacies of distributed computing

3 How do we program such a machine?

There are some common misconceptions about distributed computing that can lead to poor performance and scalability

- These misconceptions are known as the **fallacies of distributed computing**:
  2. The network is reliable
  8. Latency is zero
  1. Bandwidth is infinite
  4. The network is secure
  3. Topology doesn't change
  5. There is one administrator
  6. Transport cost is zero
  7. The network is homogeneous
- It is important to be aware of these fallacies when designing and implementing distributed applications



L. Peter Deutsch

# Working on a cluster/shared machine with MPI

## 3 How do we program such a machine?

To work on a cluster or shared machine with MPI, we typically need to follow these steps:

- Connect to the cluster using SSH
- Load the MPI module using `module load` or `spack load`
- ⚠ Compile the MPI program using the MPI wrapper compiler (e.g., `mpifort`, `mpicc`)
- Submit the MPI job to the job scheduler (e.g., SLURM, PBS, LSF) using a job script
- Monitor the job status and retrieve the output files

The compile step ⚠ may have to be done on compute nodes, depending on the cluster configuration.

# The SLURM job scheduler

## 3 How do we program such a machine?

SLURM (Simple Linux Utility for Resource Management) is a popular **job scheduler** used in many HPC clusters.

- SLURM manages the allocation of resources (e.g., nodes, CPUs, memory) for jobs submitted by users
- Users submit jobs to SLURM using a job script that specifies the resources required and the commands to execute
- SLURM schedules jobs based on resource availability and job priorities
- Users can monitor the status of their jobs using SLURM commands (e.g., `squeue`, `sacct`)
- Once a job is completed, users can retrieve the output files generated by their jobs

# SLURM glossary



**node:** A single physical or virtual machine in the cluster

**task:** A single instance of a program running on a node

**job:** A collection of tasks that are submitted to SLURM for execution

**partition:** A group of nodes with similar characteristics (e.g., hardware, software)

**allocation:** A reservation of resources (nodes, CPUs, memory) for a job

**script:** A file that contains the commands to execute a job, along with SLURM directives

**interactive session:** A temporary allocation of resources for interactive use (e.g., debugging, testing, compilation)

We usually have a number of **tasks per node**, depending on the number of available CPUs/cores, and a number of **CPUs per task**, depending on the number of threads we want to use per task.

# Running an interactive session with SLURM



To run an interactive session with SLURM, we can use the `salloc` command, specifying the resources we need:

```
salloc -N 1 -n 4 --cpus-per-task=2 --time=01:00:00 --partition=cl1
```

This command requests an interactive session with:

- 1 nodes (`-N 1`),
- 4 tasks (`-n 4`),
- 2 CPUs per task (`--cpus-per-task=2`),
- a time limit of 1 hour (`--time=01:00:00`),
- on the `cl1` partition (`--partition=cl1`).

# Running an interactive session with SLURM



To run an interactive session with SLURM, we can use the `salloc` command, specifying the resources we need:

```
salloc -N 1 -n 4 --cpus-per-task=2 --time=01:00:00 --partition=cl1
```

Which will print on screen something like:

```
salloc: Pending job allocation 20864
salloc: job 20864 queued and waiting for resources
salloc: job 20864 has been allocated resources
salloc: Granted job allocation 20864
```

After a while, when the resources are allocated, we will get a shell prompt on the compute node.

# Running an interactive session with SLURM



Another option to run an interactive session is to use the `srun` command with the `--pty` option:

```
srun --pty -N 1 -n 4 --cpus-per-task=2 --time=01:00:00 --partition=cl1 bash
```

This command has the same effect as the previous one, but it directly starts a bash shell on the compute node.

If the cluster supports it, you can also use `ssh` to connect directly to a compute node for interactive work (after allocating resources with `salloc`), but this is less common.

# Preparing a SLURM job script for MPI



An example of a SLURM job script `launch.sh` for running an MPI program:

```
#!/bin/bash
#SBATCH --job-name=mpi_hello
#SBATCH --output=mpi_hello.out
#SBATCH --error=mpi_hello.err
#SBATCH --nodes=2
#SBATCH --ntasks=4
#SBATCH --ntasks-per-node=2
#SBATCH --cpus-per-task=1
#SBATCH --time=01:00:00
#SBATCH --partition=cl2
```

- The script starts with a shebang line (*#!/bin/bash*) to specify the shell to use
- The *#SBATCH* directives specify the job:
  - job name (`--job-name`),
  - output file (`--output`),
  - error file (`--error`),
  - number of nodes (`--nodes`),
  - number of tasks (`--ntasks`),
  - tasks per node (`--ntasks-per-node`),
  - CPUs per task (`--cpus-per-task`),
  - time limit (`--time`),
  - partition (`--partition`).

# Output files options in SLURM job scripts



The `--output` and `--error` options in SLURM job scripts specify the files where the **standard output** and **standard error** streams of the job will be redirected.

By default, if these options are not specified, SLURM will create output files named `slurm-<jobid>.out` in the directory where the job was submitted.

You can customize the names of these files using the `--output` and `--error` options, together with some *special placeholders*:

- `%j`: Job ID
- `%N`: Node name
- `%n`: Task ID

# Preparing a SLURM job script for MPI + OpenMP



If we want to use OpenMP in addition to MPI, we need to set the number of threads per process using the `OMP_NUM_THREADS` environment variable in the job script:

```
export OMP_NUM_THREADS=4
```

This line should be added before the command that runs the MPI program.

⚠ It is crucial to ensure that `OMP_NUM_THREADS` does not exceed the total number of available CPU cores on the allocated nodes to avoid oversubscription, i.e., the number of threads should be less than or equal to the number passsed to `--cpus-per-task`.

# The execution command in SLURM job scripts



To run the MPI program in the SLURM job script, we use the `srun` command:

```
srun ./mpi_hello
```

This command launches the MPI program `mpi_hello` using the resources allocated by SLURM.

⚠ It is important to use `srun` instead of `mpirun` or `mpiexec` in SLURM job scripts, as `srun` is integrated with SLURM and ensures proper resource allocation and management; in many clusters `mpirun` and `mpiexec` are disabled or not recommended.

# Submitting the SLURM job script and checking job status



To submit the SLURM job script, we use the `sbatch` command:

```
sbatch launch.sh
```

This command submits the job script `launch.sh` to SLURM for execution.

To check the status of the submitted job, we can use the `squeue` command:

```
squeue -u your_username
```

This command lists all the jobs submitted by the user `your_username`, showing their job IDs, statuses, and other information.
If the job is running or completed, we can check the output and error files specified in the job script.

# Table of Contents

4 The Toeplitz cluster at DMPISA


- Distributed memory machines
- How do we program such a machine?
  - An MPI hello world program
    - Finding MPI via CMake
    - The fallacies of distributed computing
  - Working on a cluster/shared machine with MPI
- The Toeplitz cluster at DMPISA
  - Partitions on Toeplitz
  - Software

# The Toeplitz cluster at DMPISA



- The Toeplitz cluster is a distributed memory machine available at DMPISA for high-performance computing tasks
- It consists of multiple nodes, each equipped with powerful processors and a significant amount of memory
- The nodes are connected by a $10\,\text{Gbit}\,\text{s}^{-1}/25\,\text{Gbit}\,\text{s}^{-1}$ network providing communication between nodes
- The cluster is managed using the SLURM job scheduler, which allows users to submit and manage their jobs effectively
- Users can access the cluster remotely via SSH and utilize MPI for parallel programming

# Specifications of the Toeplitz cluster at DMPISA



- The Toeplitz cluster consists of 9 nodes:
  - 4 AMD EPYC 7763 nodes: 2 threads per core, 64 cores per socket, 2 sockets, $2$ TB of memory ( $1.96$ TB usable).
  - 4 Intel Xeon E5-2650 v4 at $2.20$ GHz nodes: 2 threads per core, 12 cores per socket, 2 sockets, $256$ GB of memory ( $250$ GB usable).
  - 1 Intel Xeon E5-2643 v4 at $3.40$ GHz node: 2 threads per core, 6 cores per socket, 2 sockets, $128$ GB of memory ( $125$ GB usable).
- Network connectivity:
  - The first 4 AMD nodes are connected via fiber at $25$ Gbit $s^{-1}$.
  - The remaining nodes use Intel Ethernet Controller X540-AT2 10-Gigabit NICs over copper, with a $10$ Gbit $s^{-1}$ switch.

# Accessing the Toeplitz cluster



Once you receive your account credentials, you can log in to the cluster with the command:

```
ssh username@toeplitz.cs.dm.unipi.it
```

At the first connection, you will be asked to accept the machine's fingerprint.

**!** If you intend to use services with a graphical interface on the remote machine, you need to request SSH to forward the X11 server by adding the `-X` option to the previous command.

# Setting up SSH key authentication



In general, it is useful to connect via SSH key. A key can be generated on your system by following the instructions given by:

```
ssh-keygen
```

**Important:** Set a passphrase for the generated key.

You can use the ssh-key we generated for GitHub if you want to.

Once the procedure is complete, you must copy the key to the remote machine with:

```
ssh-copy-id username@toeplitz.cs.dm.unipi.it
```

Every subsequent login from your machine will not require a password; the first login from your machine in each session will require the passphrase.

# Partitions on Toeplitz



Toeplitz contains three different partitions:

| Partition | Description | Time Limit | Nodes | Node List |
|---|---|---|---|---|
| `gpu` | 2 threads/core, 128 threads/socket<br>2 sockets, 4 NVIDIA A40 (48GB RAM) | infinite | 4 | `gpu0[1-4]` |
| `cl1` | 2 threads/core, 6 cores/socket<br>2 sockets | infinite | 1 | `lnx1` |
| `cl2` | 2 threads/core, 12 cores/socket<br>2 sockets | infinite | 4 | `lnx[2-5]` |
| `all` | All nodes in the cluster | infinite | 9 | `lnx[1-5]`,<br>`gpu0[1-4]` |

# Software management on Toeplitz



The software management on the cluster is performed using **Spack**, a package manager for supercomputers.

- Simplifies installation and management of scientific software
- Not tied to a specific programming language
- Allows creating software stacks in Python or R, linking to libraries in C, C++, or Fortran
- Easily switch compilers or program for specific microarchitectures

# Environment Modules on Toeplitz



Environment Modules is a tool that simplifies shell initialization and allows users to modify their environment during the session.
To view available modules:

```
module avail
```

Modules are named with the following pattern:
`programname/version-compiler-version`
Example output:

```
anaconda3/2021.05-gcc-12.2.0
cmake/3.23.3-gcc-12.2.0
gcc/12.2.0
intel-oneapi-compilers/2022.1.0
openmpi/4.1.4-gcc-12.2.0
openblas/0.3.20-gcc-12.2.0
 valgrind/3.19.0-oneapi-2022.1.0
```

⚠ All loaded modules must refer to the **same compiler** for a consistent environment.

# Loading and managing modules



Load modules:

```
module load programname1/version-compiler-version
↪  programname2/version-compiler-version
```

Remove modules:

```
module unload programname1/version-compiler-version
```

Revert to original state:

```
module purge
```

View active modules:

```
module list
```

# Anaconda on Toeplitz



For installing and managing Python environments we use **Anaconda**.

- Anaconda is a Python environment designed to work with multiple isolated **environments**
- Each environment can contain different versions of software and modules
- This approach minimizes conflicts and undesired interactions between different projects

**Best practices when starting a new project:**

- Create a new environment (do not use the base environment)
- Install required software in the new environment
- Use the `$SCRATCH` directory for environments (more storage than home)

# Anaconda on Toeplitz



**Quick start guide:**

```
module load anaconda3
conda create -p $SCRATCH/my-env-project
conda activate $SCRATCH/my-env-project
```

For subsequent uses, simply reload the module and activate the environment. Install packages with:

```
conda install pytorch torchvision torchaudio pytorch-cuda=12.1 -c pytorch
↪  -c nvidia
```

# Conclusions and next steps

5 Conclusions

We have:

- ✔ Introduced distributed memory machines and MPI programming
- ✔ Explained how to compile and run MPI programs on a cluster
- ✔ Presented the Toeplitz cluster at DMPISA and its software management

Next steps:

- Explore communication routines in MPI

# High Performance Linear Algebra

Lecture 8: Basic functions of MPI

Ph.D. program in High Performance Scientific Computing

**Fabio Durastante** **Pasqua D'Ambra** **Salvatore Filippone**

January 15, 2026 — 16.00:18.00

Dipartimento
di Matematica
Università di Pisa

# Last time on High Performance Linear Algebra



- Durign the last lecture we have introduced the basics of parallel programming using the **Message Passing Interface (MPI)** paradigm.
- We have discussed the main features of MPI and we have seen how to setup a simple MPI environment.
- We have also discussed the usage of SLURM to handle queues on HPC systems.

Today we will continue our discussion on MPI by introducing some of the most used MPI functions.

# Table of Contents

2 Message Passing Interface (MPI)


- ▶ Message Passing Interface (MPI)
  - Point-to-point communication
  - Deadlock
  - Non-Blocking Communication
  - Buffered Communication
  - Collective Communications
    - Vectorized versions of gather and scatter
  - Reduction operations

# Point-to-Point Communication



- **Send** and **Receive** operations between pairs of processes
- Essential building block for distributed algorithms
- Two main types: Blocking and Non-blocking

# Blocking Send and Receive



```
program point_to_point
    use mpi
    implicit none
    integer :: rank, size, ierr, status(MPI_STATUS_SIZE)
    integer :: send_data, recv_data

    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)

    if (rank == 0) then
        send_data = 42
```

# Blocking Send and Receive



```
        call MPI_Send(send_data, 1, MPI_INTEGER, 1, 0, &
                      MPI_COMM_WORLD, ierr)
    else if (rank == 1) then
        call MPI_Recv(recv_data, 1, MPI_INTEGER, 0, 0, &
                      MPI_COMM_WORLD, status, ierr)
        print *, 'Received:', recv_data
    end if

    call MPI_Finalize(ierr)
end program point_to_point
```

# Blocking Send and Receive



Let us analyze the code line by line:

- `MPI_Init`: Initialize the MPI environment
- `MPI_Comm_rank`: Get the rank of the calling process in the communicator
- `MPI_Comm_size`: Get the total number of processes in the communicator
- `MPI_Send`: Locally Blocking send operation
- `MPI_Recv`: Blocking receive operation
- `MPI_Finalize`: Clean up the MPI environment

# Blocking Send and Receive



Let us analyze the code line by line:

- `MPI_Init`: Initialize the MPI environment
- `MPI_Comm_rank`: Get the rank of the calling process in the communicator
- `MPI_Comm_size`: Get the total number of processes in the communicator
- `MPI_Send`: Locally Blocking send operation
- `MPI_Recv`: Blocking receive operation
- `MPI_Finalize`: Clean up the MPI environment

A call is said to be **blocking** if the function does not return control to the calling process until the operation is complete.

- `MPI_Send` returns only after the data has been copied out of the *send buffer*
- `MPI_Recv` returns only after the data has been received and placed in the receive buffer
- This can lead to inefficiencies if processes are waiting for each other! However, blocking calls are often simpler to implement and reason about.

# Arguments of the MPI Send/Receive



The argument of the Send call are:

- `send_data`: starting address of the send buffer
- `1`: number of elements to send
- `MPI_INTEGER`: datatype of each element
- `1`: rank of the destination process
- `0`: message tag (used to identify messages)
- `MPI_COMM_WORLD`: communicator
- `ierr`: error code

The prototype of the `MPI_Send` function is:

```
call MPI_Send(send_data, counter, datatype, dest, tag, comm, ierr)
```

# Arguments of the MPI Send/Receive



The argument of the Receive call are:

- `recv_data`: starting address of the receive buffer
- `1`: number of elements to receive
- `MPI_INTEGER`: datatype of each element
- `0`: rank of the source process
- `0`: message tag (used to identify messages)
- `MPI_COMM_WORLD`: communicator
- `status`: status object (contains information about the received message)
- `ierr`: error code

The prototype of the `MPI_Recv` function is:

```
call MPI_Recv(recv_data, counter, datatype, source, tag, comm, status, ierr)
```

# Arguments of the MPI Send/Receive



The argument of the Receive call are:

- `recv_data`: starting address of the receive buffer
- `1`: number of elements to receive
- `MPI_INTEGER`: datatype of each element
- `0`: rank of the source process
- `0`: message tag (used to identify messages)
- `MPI_COMM_WORLD`: communicator
- `status`: status object (contains information about the received message)
- `ierr`: error code

The `status` object can be used to retrieve additional information about the received message, such as the actual number of elements received or the source of the message.

# MPI Datatypes



MPI provides a variety of predefined datatypes to represent different kinds of data. The following table lists them:

| Fortran Type | MPI Datatype |
|---|---|
| `INTEGER` | `MPI_INTEGER` |
| `REAL` | `MPI_REAL` |
| `DOUBLE PRECISION` | `MPI_DOUBLE_PRECISION` |
| `COMPLEX` | `MPI_COMPLEX` |
| `DOUBLE COMPLEX` | `MPI_DOUBLE_COMPLEX` |
| `LOGICAL` | `MPI_LOGICAL` |
| `CHARACTER` | `MPI_CHAR` |

# Questions



- What happens if we run the code with 1 process only? What if we run it with more than 2 processes?
- What happens if the sender and receiver have mismatched tags or datatypes?
- How can we handle errors in MPI calls?

# Questions



- What happens if we run the code with 1 process only? What if we run it with more than 2 processes?
- What happens if the sender and receiver have mismatched tags or datatypes?
- How can we handle errors in MPI calls?

The program will hang if there is only 1 process, as the receiver will wait indefinitely for a message that will never arrive. If run with more than 2 processes, only ranks 0 and 1 will participate in the communication; other ranks will do nothing.

# Questions



- What happens if we run the code with 1 process only? What if we run it with more than 2 processes?
- What happens if the sender and receiver have mismatched tags or datatypes?
- How can we handle errors in MPI calls?

- The program will hang if there is only 1 process, as the receiver will wait indefinitely for a message that will never arrive. If run with more than 2 processes, only ranks 0 and 1 will participate in the communication; other ranks will do nothing.
- Mismatched tags or datatypes will lead to errors or unexpected behavior. The receiver may not receive the intended message, leading to data corruption or program crashes.

# Questions



- What happens if we run the code with 1 process only? What if we run it with more than 2 processes?
- What happens if the sender and receiver have mismatched tags or datatypes?
- How can we handle errors in MPI calls?

- The program will hang if there is only 1 process, as the receiver will wait indefinitely for a message that will never arrive. If run with more than 2 processes, only ranks 0 and 1 will participate in the communication; other ranks will do nothing.
- Mismatched tags or datatypes will lead to errors or unexpected behavior. The receiver may not receive the intended message, leading to data corruption or program crashes.
- MPI functions return an error code that can be checked after each call. Additionally, MPI provides error handling routines to manage errors more gracefully.

# MPI Error Handling



- Every MPI function returns an error code in the `ierr` variable
- `ierr == MPI_SUCCESS` indicates successful execution
- Error codes can be converted to human-readable messages using `MPI_Error_string`

```
integer :: ierr
character(len=MPI_MAX_ERROR_STRING) :: error_string
integer :: error_len

call MPI_Send(data, 1, MPI_INTEGER, dest, tag, comm, ierr)
if (ierr /= MPI_SUCCESS) then
    call MPI_Error_string(ierr, error_string, error_len, ierr)
    print *, 'Error: ', error_string(1:error_len)
    call MPI_Finalize(ierr)
    stop
end if
```

# MPI Error Handling



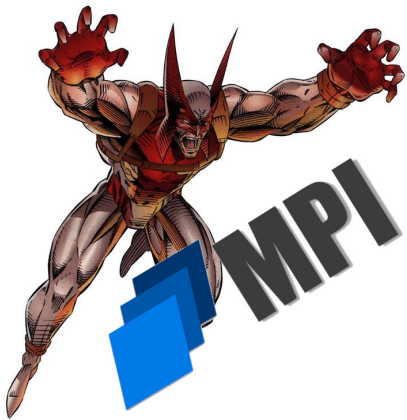
- Every MPI function returns an error code in the `ierr` variable
- `ierr == MPI_SUCCESS` indicates successful execution
- Error codes can be converted to human-readable messages using `MPI_Error_string`

- Always check error codes after critical MPI calls
- Use `MPI_Error_string` to get descriptive error messages
- Call `MPI_Finalize` before terminating on error

# Deadlock in MPI



- A **deadlock** occurs when processes are blocked indefinitely, waiting for events that will never occur
- Common cause: circular waiting patterns in blocking send/receive operations
- **Example:** Two processes each waiting to receive before sending



# Deadlock Example: Circular Wait



```
program deadlock_example
    use mpi
    implicit none
    integer :: rank, size, ierr, status(MPI_STATUS_SIZE)
    integer :: data

    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)

    if (rank == 0) then
        ! Rank 0: First receive, then send
```

# Deadlock Example: Circular Wait



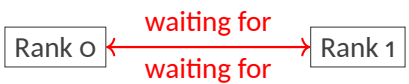
```
        call MPI_Recv(data, 1, MPI_INTEGER, 1, 0, &
                      MPI_COMM_WORLD, status, ierr)
        call MPI_Send(data, 1, MPI_INTEGER, 1, 0, &
                      MPI_COMM_WORLD, ierr)
    else if (rank == 1) then
        ! Rank 1: First receive, then send (same pattern!)
        call MPI_Recv(data, 1, MPI_INTEGER, 0, 0, &
                      MPI_COMM_WORLD, status, ierr)
        call MPI_Send(data, 1, MPI_INTEGER, 0, 0, &
                      MPI_COMM_WORLD, ierr)
    end if
```

# Deadlock Example: Circular Wait



```
    call MPI_Finalize(ierr)
end program deadlock_example
```

- Rank 0 calls `MPI_Recv` first and waits for data from rank 1
- Rank 1 calls `MPI_Recv` first and waits for data from rank 0
- Both processes are now blocked, each waiting for the other to send
- Neither process can proceed $\Rightarrow$ **deadlock**!



# Solution 1: Order Communication



```
if (rank == 0) then
    ! Rank 0: Send first, then receive
    call MPI_Send(data, 1, MPI_INTEGER, 1, 0, MPI_COMM_WORLD, ierr)
    call MPI_Recv(data, 1, MPI_INTEGER, 1, 0, MPI_COMM_WORLD, status, ierr)
else if (rank == 1) then
    ! Rank 1: Receive first, then send (opposite order)
    call MPI_Recv(data, 1, MPI_INTEGER, 0, 0, MPI_COMM_WORLD, status, ierr)
    call MPI_Send(data, 1, MPI_INTEGER, 0, 0, MPI_COMM_WORLD, ierr)
end if
```

- Break the circular dependency by using different orderings
- Rank 0 sends first, so rank 1's receive completes
- Rank 1 can then send, so rank 0's receive completes

# Non-Blocking Communication



```
integer :: request
! Start non-blocking send
call MPI_Isend(send_data, 1, MPI_INTEGER, 1, 0, &
               MPI_COMM_WORLD, request, ierr)
! Do computation while message is in transit
! ...
! Wait for completion
call MPI_Wait(request, status, ierr)
```

- `MPI_Isend`: initiate non-blocking send operation
- Returns immediately without waiting for the send to complete
- Enables overlap of communication and computation

# Non-Blocking Communication



```
integer :: request
! Start non-blocking receive
call MPI_Irecv(recv_data, 1, MPI_INTEGER, 0, 0, &
               MPI_COMM_WORLD, request, ierr)
! Do computation while waiting for message
! ...
! Wait for completion
call MPI_Wait(request, status, ierr)
```

- `MPI_Irecv`: initiate non-blocking receive operation
- Returns immediately without waiting for the message to arrive
- `MPI_Wait`: blocks until the operation completes and data is available

# The request argument and the wait commands



- The `request` argument is an integer that uniquely identifies the operation
- It is used to track the status of the operation and is required for completion routines like `MPI_Wait`
- `MPI_Wait` blocks the calling process until the specified non-blocking operation completes
- It takes the request handle and a status object as arguments
- The status object can provide information about the completed operation, such as the source, tag, and error code

If we have multiple non-blocking operations, we can use `MPI_Waitall` to wait for all of them to complete:

```
call MPI_Waitall(count, request_array, status_array, ierr)
```

# Solution 2: Use Non-Blocking Operations



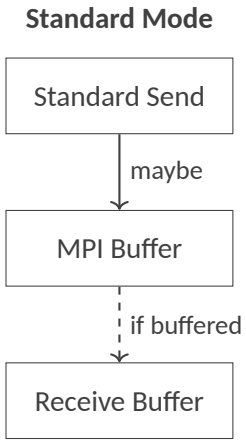
```
integer :: req_send, req_recv
integer :: status_array(MPI_STATUS_SIZE, 2)

! Both ranks initiate sends and receives simultaneously
call MPI_Isend(data, 1, MPI_INTEGER, 1-rank, 0, MPI_COMM_WORLD, &
    req_send, ierr)
call MPI_Irecv(data, 1, MPI_INTEGER, 1-rank, 0, MPI_COMM_WORLD, &
    req_recv, ierr)
! Wait for both operations to complete
call MPI_Waitall(2, (/req_send, req_recv/), &
                 status_array, ierr)
```

- Non-blocking operations return immediately
- Both sends and receives can progress independently
- MPI runtime handles message buffering and ordering

# Buffered Communication



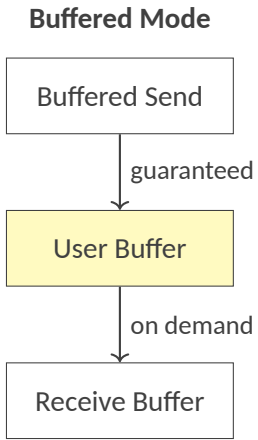
- **Standard mode** (default): MPI decides whether to buffer or not
- **Buffered mode**: User provides explicit buffer for send operations
- Useful when you want guaranteed buffering without relying on MPI's internal buffers
- Allows more predictable behavior in certain scenarios



# Buffered Communication



- **Standard mode** (default): MPI decides whether to buffer or not
- **Buffered mode**: User provides explicit buffer for send operations
- Useful when you want guaranteed buffering without relying on MPI's internal buffers
- Allows more predictable behavior in certain scenarios



# MPI Buffered Send



```
integer :: buffer_size, provided_size
integer, allocatable :: buffer(:)

! Determine required buffer size
call MPI_Pack_size(1, MPI_INTEGER, MPI_COMM_WORLD, buffer_size, ierr)
buffer_size = buffer_size + MPI_BSEND_OVERHEAD
allocate(buffer(buffer_size))

! Attach the buffer to MPI
call MPI_Buffer_attach(buffer, buffer_size, ierr)

! Now use MPI_Bsend instead of MPI_Send
call MPI_Bsend(data, 1, MPI_INTEGER, dest, tag, MPI_COMM_WORLD, ierr)
```

# MPI Buffered Send



```
! Detach buffer when done
call MPI_Buffer_detach(buffer, buffer_size, ierr)
deallocate(buffer)
```

- `MPI_Bsend`: buffered send operation
- `MPI_Buffer_attach`: attach user-provided buffer
- `MPI_Buffer_detach`: detach buffer (must wait for all sends to complete)

# Solution 3: Use Buffered Send



```
integer :: buffer_size
integer, allocatable :: buffer(:)

call MPI_Pack_size(1, MPI_INTEGER, MPI_COMM_WORLD, buffer_size, ierr)
buffer_size = buffer_size + MPI_BSEND_OVERHEAD
allocate(buffer(buffer_size))

call MPI_Buffer_attach(buffer, buffer_size, ierr)

! Both ranks can now safely use Bsend
call MPI_Bsend(data, 1, MPI_INTEGER, 1-rank, 0, MPI_COMM_WORLD, ierr)
call MPI_Recv(data, 1, MPI_INTEGER, 1-rank, 0, MPI_COMM_WORLD, &
              status, ierr)
```

# Solution 3: Use Buffered Send



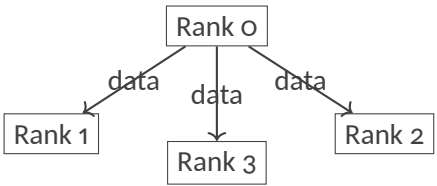
```
call MPI_Buffer_detach(buffer, buffer_size, ierr)
deallocate(buffer)
```

- `MPI_Bsend` copies data to user buffer immediately and returns
- Eliminates deadlock by guaranteeing buffering before receive is called
- Requires explicit buffer management overhead

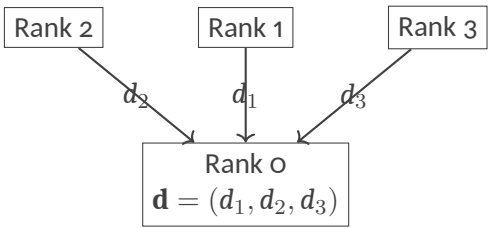
# Collective Communications



- Collective communication are operations that **involve all processes in a communicator**.
- Examples include broadcasting, gathering, scattering, and reducing data.
- These operations are essential for synchronizing data among processes.




**Broadcast**

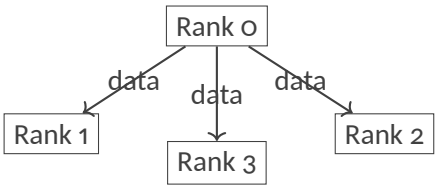



**Gather**

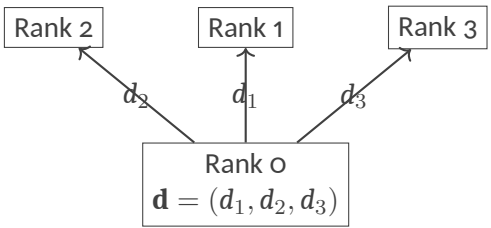
# Collective Communications



- Collective communication are operations that **involve all processes in a communicator**.
- Examples include broadcasting, gathering, scattering, and reducing data.
- These operations are essential for synchronizing data among processes.




**Broadcast**




**Scatter**

# Broadcast Operation



- The **broadcast** operation sends data from one process (the root) to all other processes in the communicator.
- Useful for distributing initial data or parameters.

```
program mpi_broadcast
    use mpi
    implicit none
    integer :: rank, size, ierr
    integer :: data

    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
```

# Broadcast Operation



```
    if (rank == 0) then
        data = 100  ! Root process initializes data
    end if

    ! Broadcast data from root process (rank 0) to all processes
    call MPI_Bcast(data, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)

    print *, 'Process', rank, 'received data:', data

    call MPI_Finalize(ierr)
end program mpi_broadcast
```

# Broadcast Operation



The arguments of the `MPI_Bcast` function are:

- `buffer`: starting address of the buffer to be broadcasted
- `1`: number of elements to broadcast
- `MPI_INTEGER`: datatype of each element
- `0`: rank of the root process
- `MPI_COMM_WORLD`: communicator
- `ierr`: error code

The prototype of the `MPI_Bcast` function is:

```
call MPI_Bcast(buffer, counter, datatype, root, comm, ierr)
```

The value of `buffer` is significant only at the root process during the call; all other processes will receive the broadcasted value into their own `buffer` variable.

# Gather Operation



- The **gather** operation collects data from all processes and sends it to a root process.
- Useful for aggregating results from multiple processes.

```
program mpi_gather
    use mpi
    implicit none
    integer :: rank, size, ierr
    integer :: send_data, recv_data(4)

    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)

    send_data = rank + 1  ! Each process sends its rank + 1
```

# Gather Operation



```
    ! Gather data at root process (rank 0)
    call MPI_Gather(send_data, 1, MPI_INTEGER, recv_data, 1, MPI_INTEGER,
    ↪  0, MPI_COMM_WORLD, ierr)

    if (rank == 0) then
        print *, 'Root process received data:', recv_data
    end if

    call MPI_Finalize(ierr)
end program mpi_gather
```

# Gather Operation



The arguments of the `MPI_Gather` function are:

- `send_buffer`: starting address of the send buffer
- `1`: number of elements sent by each process
- `MPI_INTEGER`: datatype of each element
- `recv_buffer`: starting address of the receive buffer (only significant at root)
- `1`: number of elements received from each process
- `MPI_INTEGER`: datatype of each element
- `0`: rank of the root process
- `MPI_COMM_WORLD`: communicator
- `ierr`: error code

The prototype of the `MPI_Gather` function is:

```
call MPI_Gather(send_buffer, send_count, send_datatype, &
                recv_buffer, recv_count, recv_datatype, &
                root, comm, ierr)
```

# Scatter Operation



- The **scatter** operation distributes data from a root process to all other processes.
- Useful for distributing chunks of data for parallel processing.

```
program mpi_scatter
    use mpi
    implicit none
    integer :: rank, size, ierr
    integer :: send_data(4), recv_data
    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
    if (rank == 0) then
        send_data = (/10, 20, 30, 40/)  ! Root process initializes data
    end if
```

# Scatter Operation



```
    ! Scatter data from root process (rank 0) to all processes
    call MPI_Scatter(send_data, 1, MPI_INTEGER, recv_data, 1, MPI_INTEGER,
    ↪  0, MPI_COMM_WORLD, ierr)
    print *, 'Process', rank, 'received data:', recv_data
    call MPI_Finalize(ierr)
end program mpi_scatter
```

# Scatter Operation



The arguments of the `MPI_Scatter` function are:

- `send_buffer`: starting address of the send buffer (only significant at root)
- `1`: number of elements sent to each process
- `MPI_INTEGER`: datatype of each element
- `recv_buffer`: starting address of the receive buffer
- `1`: number of elements received by each process
- `MPI_INTEGER`: datatype of each element
- `0`: rank of the root process
- `MPI_COMM_WORLD`: communicator
- `ierr`: error code

The prototype of the `MPI_Scatter` function is:

```
call MPI_Scatter(send_buffer, send_count, send_datatype, &
                 recv_buffer, recv_count, recv_datatype, &
                 root, comm, ierr)
```

# MPI_Gatherv Operation



- The **gatherv** operation collects variable amounts of data from each process to a root process.
- Useful when processes have different amounts of data to contribute.

```
program mpi_gatherv
    use mpi
    implicit none
    integer :: rank, size, ierr
    integer :: send_data(10), recv_data(25)
    integer :: send_count, recv_counts(4), displs(4)

    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
    ! Each process sends different amount of data
```

# MPI_Gatherv Operation



```
    send_count = rank + 1
    send_data(1:send_count) = rank
    ! Root process specifies how much to receive from each process
    recv_counts = (/1, 2, 3, 4/)
    displs = (/0, 1, 3, 6/)  ! Displacements in receive buffer
    call MPI_Gatherv(send_data, send_count, MPI_INTEGER, &
                     recv_data, recv_counts, displs, MPI_INTEGER, &
                     0, MPI_COMM_WORLD, ierr)
    if (rank == 0) then
        print *, 'Root received data:', recv_data(1:10)
    end if

    call MPI_Finalize(ierr)
end program mpi_gatherv
```

# MPI_Scatterv Operation



- The **scatterv** operation distributes variable amounts of data from a root process to all processes.
- Useful for load balancing when processes need different data sizes.

```
program mpi_scatterv
    use mpi
    implicit none
    integer :: rank, size, ierr
    integer :: send_data(10), recv_data(10)
    integer :: send_counts(4), displs(4), recv_count

    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
    if (rank == 0) then
```

# MPI_Scatterv Operation



```
        send_data = (/1, 2, 3, 4, 5, 6, 7, 8, 9, 10/)
    end if
    ! Root specifies how much to send to each process
    send_counts = (/1, 2, 3, 4/)
    displs = (/0, 1, 3, 6/)  ! Displacements in send buffer
    recv_count = rank + 1
    call MPI_Scatterv(send_data, send_counts, displs, MPI_INTEGER, &
                      recv_data, recv_count, MPI_INTEGER, &
                      0, MPI_COMM_WORLD, ierr)
    print *, 'Process', rank, 'received:', recv_data(1:recv_count)
    call MPI_Finalize(ierr)
end program mpi_scatterv
```

# MPI_Gatherv and MPI_Scatterv Parameters



**MPI_Gatherv**

- `send_buffer`: data to send
- `send_count`: amount sent by this process
- `recv_counts`: array of receive counts per process
- `displs`: array of displacements in receive buffer

**MPI_Scatterv**

- `send_counts`: array of send counts per process
- `displs`: array of displacements in send buffer
- `recv_buffer`: buffer to receive data
- `recv_count`: amount received by this process

# Reduce Operation



- The **reduce** operation combines data from all processes and sends the result to a root process.
- Commonly used for summing values or finding maximum/minimum.

```
program mpi_reduce
    use mpi
    implicit none
    integer :: rank, size, ierr
    integer :: send_data, recv_data

    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)

    send_data = rank + 1  ! Each process sends its rank + 1
```

# Reduce Operation Arguments



The arguments of the `MPI_Reduce` function are:

- `send_buffer`: starting address of the send buffer
- `1`: number of elements to reduce
- `MPI_INTEGER`: datatype of each element
- `MPI_SUM`: reduction operation (MPI_SUM, MPI_MAX, MPI_MIN, etc.)
- `0`: rank of the root process
- `MPI_COMM_WORLD`: communicator
- `ierr`: error code

The prototype of the `MPI_Reduce` function is:

```
call MPI_Reduce(send_buffer, recv_buffer, counter, datatype, op, root,
↪  comm, ierr)
```

The reduce buffer is only significant at the root process.

# MPI Reduction Operations



MPI provides several built-in reduction operations:

| Operation | Description | Operation | Description |
|---|---|---|---|
| `MPI_SUM` | Sum of values | `MPI_MAX` | Maximum value |
| `MPI_PROD` | Product of values | `MPI_MIN` | Minimum value |
| `MPI_MAXLOC` | Maximum value and its location (rank) | `MPI_MINLOC` | Minimum value and its location (rank) |
| `MPI_LAND` | Logical AND | `MPI_BAND` | Bitwise AND |
| `MPI_LOR` | Logical OR | `MPI_BOR` | Bitwise OR |
| `MPI_LXOR` | Logical XOR | `MPI_BXOR` | Bitwise XOR |

# MPI Collective — Reduce



You can also define your own custom reduction operations. This is done using the

```
call MPI_Op_create(user_function, commute, op, ierr)
```

where the user-defined function has the interface:

```
subroutine user_function(invec, inoutvec, len, datatype)
    implicit none
    integer :: len
    integer :: datatype
    ! invec and inoutvec are assumed-size arrays
    ! Operation: inoutvec = invec op inoutvec
end subroutine user_function
```

The operation `op` is *assumed* to be associative; if `commute == .false.` the order of the operands must be forced in ascending process rank order, see the naive implementation example in the MPI standard document for details.

# MPI Collective — Reduce



What is the output of a collective communication?

## Collective features

- If the underlying operation is *not* associative, the results *cannot* be the same with different number of processes;
- If the collective is implemented *without* enforcing ordering, even *two successive runs on the same machine* will give different outputs.

## Warnings

- Never test a floating point result for exact match;
- Never expect a specific value from different machine configurations;
- Always use the result of a collective to govern global application behaviour;
- Always test results for appropriate bounds.

# Allreduce Operation



Combines data from all processes and distributes the result to **all** processes:

```
program mpi_allreduce
    use mpi
    implicit none
    integer :: rank, size, ierr
    integer :: send_data, recv_data

    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)

    send_data = rank + 1
```

# Allreduce Operation



```
    ! All processes receive the result
    call MPI_Allreduce(send_data, recv_data, 1, MPI_INTEGER, &
                       MPI_SUM, MPI_COMM_WORLD, ierr)

    print *, 'Process', rank, 'received:', recv_data

    call MPI_Finalize(ierr)
end program mpi_allreduce
```

- Useful when all processes need the reduced result
- No root process specification needed
- Commonly used in iterative algorithms (e.g., convergence checks, orthogonalization coefficients)
- May be a bottleneck at scale due to synchronization requirements

# Conclusions and next steps

3 Conclusions

We have covered:

- ✔ Point-to-point communication (blocking, non-blocking, buffered)
- ✔ Collective communication (broadcast, gather, scatter, reduce)

Next steps:

- Investigate the cost of communication in parallel applications
- Measuring time, and putting barriers
- Explore few advanced MPI features (derived datatypes, communicators)
- Reuse and adapt code examples for our linear algebra tasks

# High Performance Linear Algebra

Lecture 9: Basic functions of MPI — Part 2

Ph.D. program in High Performance Scientific Computing

**Fabio Durastante** **Pasqua D'Ambra** **Salvatore Filippone**

January 19, 2026 — 14.00:16.00

Dipartimento
di Matematica
Università di Pisa

# Last time on High Performance Linear Algebra



- Durign the last lecture we have introduced the basics of parallel programming using the **Message Passing Interface (MPI)** paradigm.
- We have discussed the main features of MPI and we have seen how to setup a simple MPI environment.
- We have also discussed the usage of SLURM to handle queues on HPC systems.

Today we will continue our discussion on MPI by introducing some of the most used MPI functions.

# Table of Contents

2 Message Passing Interface (MPI)


- ▶ Message Passing Interface (MPI)
  - Timing and barriers
  - The Cost of Communication
    - Ping-Pong Test for Point-to-Point Communication
    - The cost of collective communications
- ▶ Restarting with BLAS: Level 1 routines
  - How do we distribute vectors?
  - Creation and destruction

# MPI Timers



- `MPI_Wtime()` returns a double-precision wall-clock time in seconds
- `MPI_Wtick()` returns the resolution of `MPI_Wtime()`
- Suitable for measuring elapsed time of code regions (not CPU time)
- Call `MPI_Init` before and `MPI_Finalize` after using timers

## CPU time vs Wall-clock time

CPU time measures the time a CPU spends executing a program, while wall-clock time measures the real-world elapsed time from start to finish, including waiting times and delays. To **measure performance** in parallel computing, wall-clock time is often more relevant as it reflects the actual time users experience.

# Example: Timing a Loop



```
program timing_example
  use mpi
  implicit none
  integer :: ierr, rank, nprocs, i
  real(kind=8) :: t0, t1, elapsed

  call MPI_Init(ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
  call MPI_Comm_size(MPI_COMM_WORLD, nprocs, ierr)

  call MPI_Barrier(MPI_COMM_WORLD, ierr)  ! synchronize start
  t0 = MPI_Wtime()
```

# Example: Timing a Loop



```
  do i = 1, 10**7
     call random_seed()   ! dummy work
  end do

  call MPI_Barrier(MPI_COMM_WORLD, ierr)   ! synchronize end
  t1 = MPI_Wtime()

  elapsed = t1 - t0
  print '(A,I3,A,F8.4)', 'Rank ', rank, ' elapsed: ', elapsed

  call MPI_Finalize(ierr)
end program timing_example
```

# Barriers and Timing Best Practices



- **Barriers** (`MPI_Barrier`) force all ranks to synchronize
- Use barriers to align start/end of timed regions; avoid overuse
- Prefer `MPI_Reduce` (e.g., `MPI_MAX`) to collect max elapsed time across ranks
- Run multiple iterations and average to smooth variability

# Example: Timing with Reduction



```
program timing_reduce
  use mpi
  implicit none
  integer :: ierr, rank, root
  real(kind=8) :: t0, t1, tlocal, tmax

  call MPI_Init(ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
  root = 0
  call MPI_Barrier(MPI_COMM_WORLD, ierr)
  t0 = MPI_Wtime()
  call MPI_Bcast(tlocal, 1, MPI_DOUBLE_PRECISION, root, MPI_COMM_WORLD,
  ↪  ierr)
```

# Example: Timing with Reduction



```
  call MPI_Barrier(MPI_COMM_WORLD, ierr)
  t1 = MPI_Wtime()

  tlocal = t1 - t0
  call MPI_Reduce(tlocal, tmax, 1, MPI_DOUBLE_PRECISION, MPI_MAX, root,
  ↪  MPI_COMM_WORLD, ierr)

  if (rank == root) then
     print *, 'Max elapsed across ranks: ', tmax
  end if

  call MPI_Finalize(ierr)
end program timing_reduce
```

# The Cost of Communication



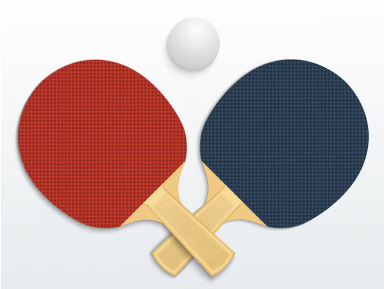
- Communication overhead increases with the number of processes
- Two main components: **latency** and **bandwidth**
  - **Latency**: time to initiate a message (startup cost)
  - **Bandwidth**: data transfer rate once communication starts
- Total communication time: $T_{\text{comm}} = \alpha + \frac{N}{\beta}$
  - $\alpha$: latency (message startup cost)
  - $\beta$: reciprocal of the bandwidth
  - $N$: message size in bytes

# Scaling Communication Costs: Point-to-Point Example



The fist measure we can do is to evaluate the scaling of point-to-point communication costs with respect to the number of MPI ranks.

An idea is to implement a simple benchmark that implements a **ping-pong test** between two MPI ranks, where rank 0 sends a message to rank $n-1$, which immediately sends it back.



The time taken for this round-trip communication is measured and reported.

# Scaling Communication Costs: Point-to-Point Example



```
program test_mpi_pingpong
   use mpi
   implicit none
   integer :: ierr, rank, nprocs
   integer :: n, i, niter
   real(kind=8), allocatable :: sendbuf(:), recvbuf(:)
   real(kind=8) :: t_start, t_end, t_local, t_avg, t_max
   call MPI_Init(ierr)
   call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
   call MPI_Comm_size(MPI_COMM_WORLD, nprocs, ierr)
   ! ------------------------------
   ! Fixed problem size (strong scaling)
   ! ------------------------------
```

# Scaling Communication Costs: Point-to-Point Example



```
n = 1000000       ! number of double-precision elements
niter = 100       ! number of repetitions
allocate(sendbuf(n), recvbuf(n))
sendbuf = 1.0d0
recvbuf = 0.0d0
```

We perform a warm-up phase to avoid measuring initial overheads,

```
! Warm-up (not timed)
if (rank == 0) then
   call MPI_Send(sendbuf, n, MPI_DOUBLE_PRECISION, nprocs-1, 0,
   ↪  MPI_COMM_WORLD, ierr)
   call MPI_Recv(recvbuf, n, MPI_DOUBLE_PRECISION, nprocs-1, 0,
   ↪  MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)
else if (rank == nprocs-1) then
```



```
    call MPI_Recv(recvbuf, n, MPI_DOUBLE_PRECISION, 0, 0, MPI_COMM_WORLD,
    ↪  MPI_STATUS_IGNORE, ierr)
    call MPI_Send(sendbuf, n, MPI_DOUBLE_PRECISION, 0, 0, MPI_COMM_WORLD,
    ↪  ierr)
  end if
  call MPI_Barrier(MPI_COMM_WORLD, ierr)
```

then we perform the ping-pong test for a number of iterations, measuring the time taken for each round-trip communication.

# Scaling Communication Costs: Point-to-Point Example



```
t_local = 0.0d0
do i = 1, niter
   call MPI_Barrier(MPI_COMM_WORLD, ierr)
   t_start = MPI_Wtime()
   if (rank == 0) then
      call MPI_Send(sendbuf, n, MPI_DOUBLE_PRECISION, nprocs-1, 0,
      ↪  MPI_COMM_WORLD, ierr)
      call MPI_Recv(recvbuf, n, MPI_DOUBLE_PRECISION, nprocs-1, 0,
      ↪  MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)
   else if (rank == nprocs-1) then
      call MPI_Recv(recvbuf, n, MPI_DOUBLE_PRECISION, 0, 0,
      ↪  MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)
      call MPI_Send(sendbuf, n, MPI_DOUBLE_PRECISION, 0, 0,
      ↪  MPI_COMM_WORLD, ierr)
```

# Scaling Communication Costs: Point-to-Point Example



```
      end if
      t_end = MPI_Wtime()
      t_local = t_local + (t_end - t_start)
   end do
   t_avg = t_local / niter
```

Finally, we compute the average time and bandwidth, reporting the results from rank 0.

```
   ! Take the maximum time across all ranks
   call MPI_Reduce(t_avg, t_max, 1, MPI_DOUBLE_PRECISION, MPI_MAX, 0,
   ↪  MPI_COMM_WORLD, ierr)
   if (rank == 0) then
      print *, "MPI Ping-Pong strong scaling test"
      print *, "Message size (MB): ", n * 8.0d0 / 1.0d6
      print *, "MPI ranks        : ", nprocs
```

# Scaling Communication Costs: Point-to-Point Example



```
      print *, "Avg Ping-Pong time (s): ", t_max
      print *, "Avg Bandwidth (GB/s): ", (n * 8.0d0 * 2.0d0) / (t_max *
      ↪  1.0d9)
      print *, "Avg Bandwidth (GBit/s): ", (n * 8.0d0 * 8.0d0 * 2.0d0) /
      ↪  (t_max * 1.0d9)
   end if
   deallocate(sendbuf, recvbuf)
   call MPI_Finalize(ierr)
end program test_mpi_pingpong
```

# Scaling Communication Costs: Point-to-Point Example



We recall that the bandwidth is computed as the total data transferred (send + receive) divided by the time taken, i.e.,

$$\text{Bandwidth} \approx \frac{N \times 2}{T_{\text{comm}}}$$

where $N$ is the message size in bytes and $T_{\text{comm}}$ is the average time taken for the ping-pong communication.
We have an array of size $n$ double-precision elements, so the message size in bytes is $N = n \times 8$ bytes, and the total data transferred in the ping-pong test is $N \times 2$ bytes and we multiply by 8 to convert to bits.

# Scaling Communication Costs: Point-to-Point Example



To run this benchmark on a HPC system with SLURM, we use the Amelia cluster at IAC-CNR.

This is a machine whose nodes are equipped with Intel Xeon Gold 6338 processors (32 cores per socket, 2 sockets per node), connected via an InfiniBand HDR200 network with a theoretical **peak bandwidth** of $200\,\mathrm{Gbit\,s^{-1}}$.

We write a SLURM script to run the benchmark with different numbers of MPI ranks,

```
#!/bin/bash
#SBATCH --job-name=pingpong_strong_64ppn
#SBATCH --nodes=7
#SBATCH --ntasks-per-node=64
#SBATCH --time=00:20:00
#SBATCH --partition=prod-gn
#SBATCH --mem=900Gb
#SBATCH --output=pingpong_%j.out
#SBATCH --error=pingpong_%j.err
```

# Scaling Communication Costs: Point-to-Point Example



We load the necessary modules and compile the Fortran code using `mpifort`,

```
module load intel/gcc-12.2.1/openmpi-4.1.6
```

And then use bash loops to run the benchmark with different numbers of MPI ranks,

```
for NODES in $(seq 1 7); do
    NTASKS=$((NODES * 64))
    echo "Running Ping-Pong with:"
    echo "  Nodes : $NODES"
    echo "  Tasks : $NTASKS (64 per node)"
    mpirun --bind-to core --map-by ppr:64:node --mca btl ^openib --mca pml
    ↪  ucx -x UCX_NET_DEVICES='mlx5_0:1' -np $NTASKS ./pingpong
done
```

The script can be submitted to the SLURM queue using the command:
`sbatch runner-pingpong.sh`.

# Scaling Communication Costs: Point-to-Point Example



We have passed several options to `mpirun` to optimize the communication performance:

- `--bind-to core`: binds each MPI process to a specific CPU core to reduce context switching and improve cache utilization.
- `--map-by ppr:64:node`: maps 64 MPI processes per node, ensuring an even distribution of processes across the available nodes.
- `--mca btl ^openib`: disables the OpenIB BTL (Byte Transfer Layer) to avoid potential issues with certain InfiniBand configurations.
- `--mca pml ucx`: selects the UCX (Unified Communication X) PML (Point-to-Point Messaging Layer) for improved performance on high-speed networks.
- `-x UCX_NET_DEVICES='mlx5_0:1'`: specifies the network devices to be used by UCX for communication, optimizing data transfer over the InfiniBand network.

❓ The last three options are relevant only to the specific network configuration of the Amelia cluster, thus they might not be necessary on other systems.

# Scaling Communication Costs: Point-to-Point Example



We have passed several options to `mpirun` to optimize the communication performance:

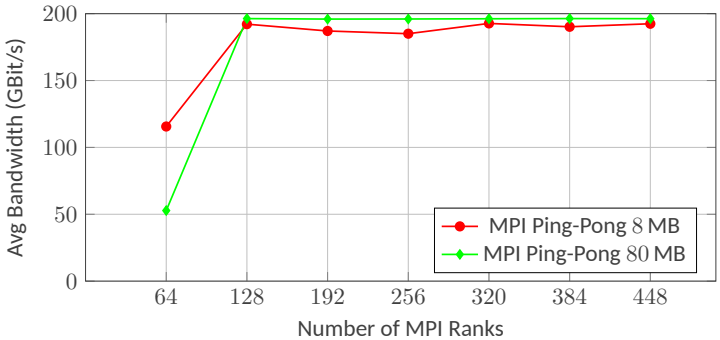
- `--bind-to core`: binds each MPI process to a specific CPU core to reduce context switching and improve cache utilization.
- `--map-by ppr:64:node`: maps 64 MPI processes per node, ensuring an even distribution of processes across the available nodes.
- `--mca btl ^openib`: disables the OpenIB BTL (Byte Transfer Layer) to avoid potential issues with certain InfiniBand configurations.
- `--mca pml ucx`: selects the UCX (Unified Communication X) PML (Point-to-Point Messaging Layer) for improved performance on high-speed networks.
- `-x UCX_NET_DEVICES='mlx5_0:1'`: specifies the network devices to be used by UCX for communication, optimizing data transfer over the InfiniBand network.

⚠ The first two options are important for what we are doing, because they ensure that we really use the network as expected.

# Scaling Communication Costs: Point-to-Point Example



We increase the number of MPI ranks from 64 to 448 (7 nodes with 64 ranks each).




The average ping-pong bandwidth approaches the theoretical peak bandwidth of the InfiniBand HDR200 network ($200\,\text{Gbit}\,\text{s}^{-1}$).

# Exercise



There are a number of possible exercises you can do to further explore the ping-pong benchmark

1. Vary the message size and perform linear regression to estimate the latency and bandwidth parameters ($\alpha$ and $\beta$) of the communication model.
2. Implement a similar benchmark using non-blocking communication (e.g., `MPI_Isend` and `MPI_Irecv`).
3. Benchmark what happens when you use MPI ranks on different nodes versus on the same node.
4. Explore the impact of different MPI implementations (e.g., OpenMPI vs MPICH vs Intel MPI) on communication performance.

# The problem of collective communications



Let's review the idea of the Broadcast operation. Algorithmically, we can implement it as:

```
if (my_rank == 0) then
    do i = 1, p-1
        call MPI_Send(a, 1, MPI_REAL, i, tag, MPI_COMM_WORLD, ierr)
    end do
else
    call MPI_Recv(a, 1, MPI_REAL, 0, tag, MPI_COMM_WORLD,
    ↪  MPI_STATUS_IGNORE, ierr)
end if
```

- This is a **collective communication** since *all* processes participate in its implementation;
- Just from a software engineering perspective, it makes sense to encapsulate it in a function: the `MPI_Bcast`.

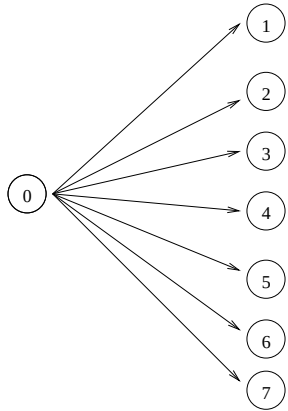
# How much does it cost?



- With fast networks, cost for 1 float is dominated by latency $\alpha$;
- Cost of this algorithm is therefore

$$T(p) \approx \alpha \cdot (p-1)$$

or, linear in $p$.



# Can we do any better?



Let's make some assumptions about the network:

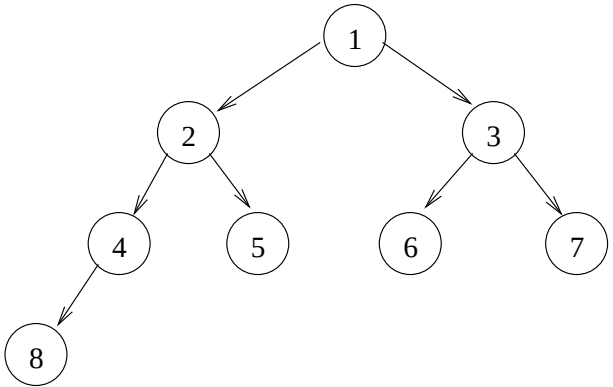
- The cost of communication between any two network nodes is uniform, and is given by $\alpha + N/\beta$;
- In particular, it is *possible* to send a message between *any* two nodes[1]
- The network is capable of sustaining multiple messages (*noisy topology*) at the same time, provided pairs of nodes involved in the messages do not overlap.

The latter assumption is especially important: we can improve communication if we can have multiple messages "flying" through the network at the same time.

[1] Historically there existed networks where only neighbouring nodes could exchange messages

# Tree broadcast — simple minded





# Tree broadcast — simple minded



Assume processes are numbered from 1:

- Each process $i$ (except 1) receives from $\lfloor (p-1)/2 \rfloor$;
- Each process such that $2i <= p$ sends first to process $2i$, then to process $2i+1$.

Cost:

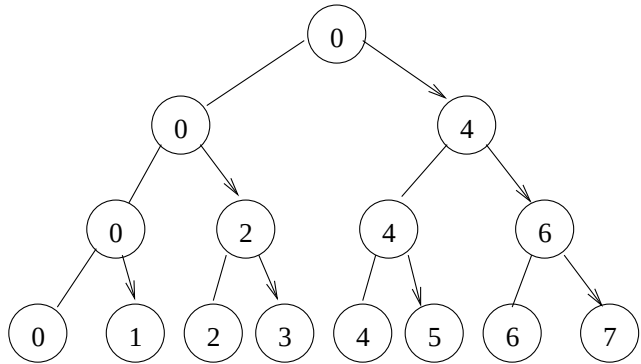
$$T(p) \approx 2\log(p),$$

or logarithmic in $p$. To be precise, with $p > 1$ then

$$T(p) = \begin{cases} 0 & \text{for } p = 1 \\ 2 \cdot (k-1) + 1 & \text{for } p = 2^k, k > 0 \\ 2 \cdot \lfloor \log_2(p) \rfloor & \text{otherwise} \end{cases}$$

# Tree broadcast — Recursive Doubling





# Tree broadcast — Recursive Doubling



- Consider that there are $p$ processes with root 0;
- Set $K$ the minimum power of 2 such that $K \geq p$;
- Process 0 sends to process $K/2$;
- Divide the processes in two groups: from 0 to $K/2 - 1$, and from $K/2$ to $\min(p-1, K-1)$;
- Apply recursively to:
  - Processes 0 to $K/2 - 1$ with root 0;
  - Processes $K/2$ to $\min(p-1, K-1)$ with root $K/2$.

Cost:

$$T(p) = \lceil \log(p) \rceil,$$

or logarithmic in $p$.

# Collective communications



Considerations for collective communications:

- Their functionality can be defined in terms of simple loops;
- There exist much better implementations;
- The **optimal implementation** for a given collective depends on:
  - The operation;
  - The network interface;
  - The network topology;
  - The amount of data.

A good MPI implementation will switch internally among different algorithms where appropriate (another advantage of encapsulating the collective)

# Collective communications



Considerations for collective communications:

- Their functionality can be defined in terms of simple loops;
- There exist much better implementations;
- The **optimal implementation** for a given collective depends on:
  - The operation;
  - The network interface;
  - The network topology;
  - The amount of data.

A good MPI implementation will switch internally among different algorithms where appropriate (another advantage of encapsulating the collective)

🔨 Let us try and take some measurements of the broadcast operation to see how its cost scales with the number of MPI ranks.

# Scaling Communication Costs: Broadcast Example



```
program test_mpi_bcast_latency
    use mpi
    implicit none

    integer :: ierr, rank, nprocs
    integer :: root
    integer :: n, i, niter
    real(kind=8), allocatable :: buffer(:)
    real(kind=8) :: t_start, t_end, t_local, t_avg, t_max

    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, nprocs, ierr)
```

# Scaling Communication Costs: Broadcast Example



```
root = 0
! Small message size for latency measurement
n = 1               ! single element for pure latency
niter = 10000       ! more iterations for better statistics
allocate(buffer(n))
if (rank == root) then
     buffer = 1.0d0
else
     buffer = 0.0d0
end if
! Warm-up (not timed)
call MPI_Bcast(buffer, n, MPI_DOUBLE_PRECISION, root, MPI_COMM_WORLD,
↪  ierr)
```

# Scaling Communication Costs: Broadcast Example



```
call MPI_Barrier(MPI_COMM_WORLD, ierr)
t_local = 0.0d0
call MPI_Barrier(MPI_COMM_WORLD, ierr)
t_start = MPI_Wtime()
do i = 1, niter
     call MPI_Bcast(buffer, n, MPI_DOUBLE_PRECISION, root,
     ↪  MPI_COMM_WORLD, ierr)
end do
t_end = MPI_Wtime()
     t_local = t_local + (t_end - t_start)
t_avg = (t_end - t_start) / niter
! Take the maximum latency across all ranks
call MPI_Reduce(t_avg, t_max, 1, MPI_DOUBLE_PRECISION, MPI_MAX, root,
↪  MPI_COMM_WORLD, ierr)
```

# Scaling Communication Costs: Broadcast Example

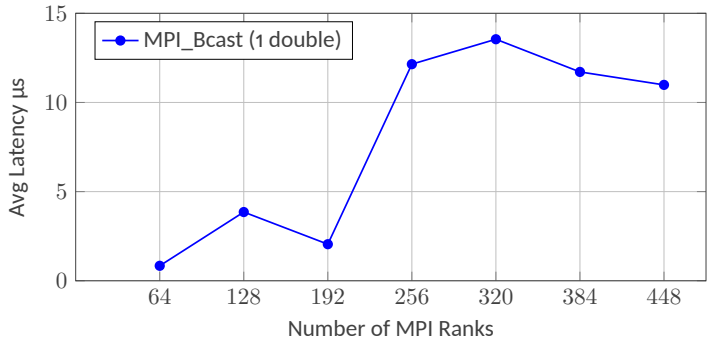


```
    if (rank == root) then
         print *, "MPI_Bcast latency test"
         print *, "MPI ranks            : ", nprocs
         print *, "Avg Bcast latency (us): ", t_max * 1.0d6
    end if
    deallocate(buffer)
    call MPI_Finalize(ierr)
end program test_mpi_bcast_latency
```

# Scaling Communication Costs: Broadcast Example



We run the benchmark on the Amelia cluster at IAC-CNR, (20 nodes with 64 ranks each).




The observed `MPI_Bcast` latency shows a non-monotonic scaling trend as the number of MPI ranks increases.

# Interpreting Broadcast Scaling Behavior



This behavior is expected and reflects the internal algorithm selection.

- **1 node (64 ranks):** Extremely low latency due to shared-memory communication.
- **2–3 nodes (128–192 ranks):** Transition to inter-node communication with efficient tree-based collectives.
- **4–5 nodes (256–320 ranks):** Sudden increase in latency caused by algorithm switching and network contention.
- **6–7 nodes (384–448 ranks):** Stabilization as MPI switches to scalable hierarchical broadcast algorithms.

Overall, broadcast latency is dominated by collective startup costs rather than message size, and is strongly influenced by communicator size, topology awareness, and algorithm thresholds.

# Table of Contents

3 Restarting with BLAS: Level 1 routines


- ▶ Message Passing Interface (MPI)
  - Timing and barriers
  - The Cost of Communication
    - Ping-Pong Test for Point-to-Point Communication
    - The cost of collective communications
- ▶ Restarting with BLAS: Level 1 routines
  - How do we distribute vectors?
  - Creation and destruction

# Restarting with BLAS: Level 1 routines



The first operation we are going to considere are the level-1 BLAS routines, which we recall are **vector-vector operations**.

- Vector scaling: $\mathbf{y} \leftarrow \alpha\mathbf{y}$ (`DSCAL`)
- Vector addition: $\mathbf{y} \leftarrow \mathbf{y} + \alpha\mathbf{x}$ (`DAXPY`)
- Vector dot product: $\alpha \leftarrow \mathbf{x}^T\mathbf{y}$ (`DDOT`)
- Vector norm: $\alpha \leftarrow \|\mathbf{x}\|_2$ (`DNRM2`)

In our model, **each process holds a local portion of the vectors**, this means that for a vector of size $N$ distributed over $P$ processes, each process holds a local vector of size $N/P$.

# How do we distribute vectors?



The **basic idea** can be represented with an easy picture:

| Proc 0 | Proc 1 | Proc 2 | Proc 3 | Proc 4 | Proc 5 | Proc 6 | Proc 7 |
|---|---|---|---|---|---|---|---|

Full Vector of Size $N$ divided across $P = 8$ Processes

This permits us to:

- Perform local computations on each process independently;
- Know what data is stored where with a closed-form formula:

$$\text{Process } p \text{ holds elements } \left[ p \cdot \frac{N}{P}, (p+1) \cdot \frac{N}{P} - 1 \right]$$

# Implementing vector distribution



To implement this distribution in code, we need to:

- Initialize MPI and get the rank and size of the communicator;
- Determine the global vector size $N$ and compute the local size $N/P$;
- Allocate local arrays for each process to hold its portion of the vector;

The inizialization code looks always the same:

```
call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, nprocs, ierr)
```

To **compute the local size** and allocate the local vector we can use a simple-minded block distribution.

# Implementing vector distribution: block distribution



Assuming $N$ is divisible by $P$, the local size is simply:

$$N_{\text{local}} = \frac{N}{P}$$

In the general case, we can compute the local size as:

$$N_{\text{local}} = \left\lfloor \frac{N + P - 1 - p}{P} \right\rfloor$$

where $p$ is the rank of the process. This formula ensures that the last process gets any remaining elements if $N$ is not perfectly divisible.

We can implement this as a function:

```
function compute_local_size(N, P, p) result(n_local)
  integer, intent(in) :: N, P, p
  integer :: n_local
  n_local = (N + P - 1 - p) / P
end function compute_local_size
```

# Implementing Level-1 BLAS: a type for distributed vectors



The first thing we need to do is to create a distributed vector datytpe, this can be done using the **object-oriented** functionalities of Fortran:

```
type :: mpi_ddistributed_vector
    integer :: n_local(1)  ! number of local elements
    integer :: n_global    ! total number of global elements
    integer :: comm        ! MPI communicator
    real(real64), allocatable :: data(:)  ! local data array
end type mpi_ddistributed_vector
```

The **type** encapsulates all necessary information about the distributed vector, including *local size*, *global size*, *communicator*, and *local data array*.

# Implementing Level-1 BLAS: type-bound procedures



In Fortran a **type** can have **type-bound procedures**, which are functions or subroutines associated with the type. Thesere are declared within the `contains` section of the type definition.

```
type :: mpi_ddistributed_vector
  <Type members>
contains
  <Type bound procedure are declared here>
end type mpi_ddistributed_vector
```

For our distributed vector type, we need to start by implementing:

- A **constructor** to initialize the distributed vector;
- A **destructor** to free resources;

# Implementing Level-1 BLAS: Constructor



The **constructor** initializes the distributed vector by computing the local size, allocating the local data array, and setting the global s ize and communicator.

- We add the following type-bound procedure to the type:

```
procedure, pass(this) :: dinit
```

- The implementation of the constructor is then inserted in che `contains` part

```
subroutine dinit(this, comm, n_global)
 use mpi
 use iso_fortran_env, only: error_unit
 implicit none
 class(mpi_ddistributed_vector), intent(inout) :: this
 integer, intent(in) :: comm
 integer, intent(in) :: n_global
end subroutine dinit
```

# Implementing Level-1 BLAS: Constructor



The **type constructor** has a default variable `this` that refers to the instance of the type being initialized.

```
integer :: ierr, rank, nprocs
this%comm = comm
this%n_global = n_global
call MPI_Comm_rank(this%comm, rank, ierr)
call MPI_Comm_size(this%comm, nprocs,
↪  ierr)
this%n_local =
↪  compute_local_size(this%n_global,
↪  nprocs, rank)
allocate(this%data(this%n_local),
↪  stat=ierr)
```

- We assign the communicator and global size to the type members;
- We get the rank and size of the communicator;
- We compute the local size using the previously defined function;
- We allocate the local data array based on the computed local size.

We should also include **error handling** for the allocation.

# Implementing Level-1 BLAS: Constructor (error handling)



The **error handling** for the allocation can be done by checking the status of the allocation.

```
if (ierr /= 0) then
    write(error_unit, *) "Error allocating local data array on rank ", rank
    call MPI_Abort(this%comm, ierr, ierr)
end if
```

The `MPI_Abort` function is called to **terminate the MPI program** if the allocation fails, ensuring that all processes are informed of the error, and that the *program fails and exits gracefully*.

# Implementing Level-1 BLAS: Destructor



The **destructor** is responsible for freeing the resources allocated to the distributed vector.

- We add use the `final` keyword in the type bound procedures to define the destructor:

```
final :: dfinalize
```

- The implementation of the destructor is then inserted in che `contains` part

```
subroutine dfinalize(this)
  type(mpi_ddistributed_vector), intent(inout) :: this
  if (allocated(this%data)) then
      deallocate(this%data)
  end if
end subroutine dfinalize
```

# The creation and destruction of distributed vectors



With the constructor and destructor defined, we can now create and destroy distributed vector instances easily.

```
integer :: n_global, rank, ierr
type(mpi_ddistributed_vector) :: x
call MPI_Init(ierr)
n_global = 1000000  ! Total number of elements
call x%dinit(MPI_COMM_WORLD, n_global)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
if (rank == 0) then
  write(output_unit, *) "MPI ranks           : ", nprocs
  write(output_unit, *) "Global vector size  : ", n_global
end if
call MPI_Barrier(MPI_COMM_WORLD, ierr)
write(output_unit, *) "Rank ", rank, ": Local vector size : ", x%n_local
call MPI_Finalize(ierr)
```

# Conclusions and next steps



Today we have:

- ✔ Reviewed the concept of communication costs in MPI;
- ✔ Implemented a point-to-point ping-pong benchmark to measure communication latency and bandwidth;
- ✔ Explored the scaling behavior of collective communications using a broadcast benchmark;
- ✔ Introduced the concept of distributed vectors and how to implement them in Fortran using object-oriented programming.

Next steps:

- Implement Level-1 BLAS operations (vector scaling, addition, dot product, norm) for distributed vectors;
- Explore Level-2 and Level-3 BLAS operations

# High Performance Linear Algebra

Lecture 10: Distributed BLAS

Ph.D. program in High Performance Scientific Computing

**Fabio Durastante** **Pasqua D'Ambra** **Salvatore Filippone**

January 22, 2026 — 16.00:18.00

Dipartimento
di Matematica
Università di Pisa

# Last time on High Performance Linear Algebra



Up to now we have seen:

- How to implement basic MPI programs in Fortran,
- Environement setup for MPI development,
- Point-to-point and collective communication,
- How to distribute vectors across MPI ranks,

We are now ready to implement distributed linear algebra routines, we will start from the Level-1 BLAS.

# Table of Contents

2 Distributed Level-1 BLAS


▶ Distributed Level-1 BLAS

Level 1: the scaling operation

Level 1: the axpy operation

Level 1: the scalar product

Level 1: the norm operation

Performance considerations for Level-1 BLAS

Scaling of the dot product and norm operations

# Last time we had implemented a distributed vector type



We have constructed the following **distributed vector type**:

```
type :: mpi_ddistributed_vector
  integer :: n_local      ! number of local elements
  integer :: n_global     ! total number of global elements
  integer :: comm         ! MPI communicator
  real(real64), allocatable :: data(:)  ! local data array
contains
  procedure, pass(this) :: dinit
  final :: dfinalize
end type mpi_ddistributed_vector
```

We have now to **enrich this type** with the Level-1 BLAS operations.

# Level 1: the scaling operation



The first Level-1 BLAS operation we are going to implement is the **vector scaling**:

$$\mathbf{y} \leftarrow \alpha\,\mathbf{y}, \quad \alpha \in \mathbb{R}, \quad \mathbf{y} \in \mathbb{R}^N.$$

This operation is **embarrassingly parallel**, since each element of the vector can be scaled independently from the others.

We can therefore implement this operation as a **method** of the distributed vector type, as before we add the method signature to the type definition:

```
type :: mpi_ddistributed_vector
  ...
contains
  ...
  procedure, pass(this) :: dscal_dist
end type mpi_ddistributed_vector
```

# Level 1: the scaling operation



Then we can implement the method using the BLAS `dscal` routine on the local data:

```
subroutine dscal_dist(this, alpha)
  implicit none
  class(mpi_ddistributed_vector), intent(inout) :: this
  real(real64), intent(in) :: alpha
  call dscal(this%n_local, alpha, this%data, 1)
end subroutine dscal_dist
```

We remind that the `dscal` routine has the **following signature**:

```
dscal(n, alpha, x, incx)
```

where `n` is the number of elements to scale, `alpha` is the scaling factor,

# Compiling and linking the distributed BLAS library



To compile and link the distributed BLAS library we need to **link against both the MPI library and the BLAS library**. The compilation command is:

```
mpifort -c -I<path_to_blas_include> mpi_ddistributed_vector.f90 -o
↪  mpi_ddistributed_vector.o
```

and the linking command is:

```
mpifort mpi_ddistributed_vector.o -L<path_to_blas_lib>
↪  -l<blas_library_name> -o libmpi_ddistributed_blas.a
```

As usual, it is convenient to create a `Makefile` or even better a `CMakeLists.txt` to automate the compilation and linking process. For the latter we need to look for both MPI and BLAS using the `find_package()` command.

# Compiling and linking the distributed BLAS library



This is an example of a `CMakeLists.txt` file to compile and link the distributed BLAS library:

```
cmake_minimum_required(VERSION 3.10)
project(mpi_ddistributed_blas LANGUAGES Fortran)
find_package(MPI REQUIRED COMPONENTS Fortran)
find_package(BLAS REQUIRED)

add_library(mpi_ddistributed_blas mpi_ddistributed_vector.f90)
target_link_libraries(mpi_ddistributed_blas MPI::MPI_Fortran
↪  BLAS::BLAS)
```

# Level 1: the axpy operation



The second Level-1 BLAS operation we are going to implement is the **axpy operation**:

$$\mathbf{y} \leftarrow \alpha \mathbf{x} + \mathbf{y}, \quad \alpha \in \mathbb{R}, \quad \mathbf{x}, \mathbf{y} \in \mathbb{R}^N.$$

This operation is again **embarrassingly parallel**, since each element of the vectors can be updated independently from the others.
The implementation pathway is the same as before, we add the method signature to the type definition:

```
type :: mpi_ddistributed_vector
  ...
contains
  ...
  procedure, pass(this) :: daxpy_dist
end type mpi_ddistributed_vector
```

# Level 1: the axpy operation



Then we can implement the method using the BLAS `daxpy` routine on the local data, but before we have **two checks** to perform:

- Check that the vector `x` is distributed over the same communicator as `this`,
- Check that the local sizes of the two vectors are the same.

The communicator check can be performed using the `MPI_Comm_compare` routine:

```
MPI_Comm_compare(comm1, comm2, result, ierr)
```

and then check that `result == MPI_IDENT`.

The local size check is simply an `if` statement on the `n_local` attribute of the two vectors.

# Level 1: the axpy operation



```
subroutine daxpy_dist(this, alpha, x)
  implicit none
  class(mpi_ddistributed_vector), intent(inout) :: this
  real(real64), intent(in) :: alpha
  class(mpi_ddistributed_vector), intent(in) :: x
  integer :: ierr, are_comm_compatible
  if (this%n_local /= x%n_local) then
      write(error_unit, *) "Error: Local sizes do not match in daxpy_dist"
      call MPI_Abort(this%comm, 1, ierr)
  end if
  call MPI_Comm_compare(this%comm, x%comm, are_comm_compatible, ierr)
  if (are_comm_compatible /= MPI_IDENT) then
      write(error_unit, *) "Error: Communicators do not match in daxpy_dist"
      call MPI_Abort(this%comm, 1, ierr)
  end if
  call daxpy(this%n_local, alpha, x%data, 1, this%data, 1)
end subroutine daxpy_dist
```

# Level 1: the scalar product



The third Level-1 BLAS operation we are going to implement is the **scalar product**:

$$\alpha = \mathbf{x}^\top \mathbf{y} = \sum_{i=1}^{N} x_i y_i, \quad \mathbf{x}, \mathbf{y} \in \mathbb{R}^N.$$

This operation is **not embarrassingly parallel**, since we need to **reduce** the contributions from all the processes to compute the final result.

We can therefore implement this operation as a **method** of the distributed vector type, as before we add the method signature to the type definition:

```
type :: mpi_ddistributed_vector
  ...
contains
  ...
  procedure, pass(this) :: ddot_dist
end type mpi_ddistributed_vector
```

# Level 1: the scalar product



Then we can implement the method using the BLAS `ddot` routine on the local data, but before we have **two checks** to perform:

- Check that the vector `x` is distributed over the same communicator as `this`,
- Check that the local sizes of the two vectors are the same.

And we have to perform a **global reduction** of the local contributions to compute the final result, but we have a decision to make:

- ? Do we want to return the result to all processes?
- ? Do we want to return the result only to the root process?

# Level 1: the scalar product



The prototype to **delegate** the choice of the reduction strategy to the user is:

```
subroutine ddot_dist(this, x, result, rank)
  implicit none
  class(mpi_ddistributed_vector), intent(in) :: this
  class(mpi_ddistributed_vector), intent(in) :: x
  integer, intent(in), optional :: rank
  real(real64), intent(out) :: result
end subroutine ddot_dist
```

If the user provides the `rank` argument, then the result is returned only to the specified rank, otherwise it is returned to all ranks.
We can then implement the method using the BLAS `ddot` routine on the local data and then performing the reduction using `MPI_Reduce` or `MPI_Allreduce`.

# Level 1: the scalar product



The two checks are the same as before, and we don't rewrite them here, then we compute the local dot product:

```
local_dot = ddot(this%n_local, this%data, 1, x%data, 1)
```

and then we perform the reduction:

```
if (present(rank)) then
    call MPI_Reduce(local_dot, result, 1, MPI_REAL8, MPI_SUM, rank,
    ↪  this%comm, ierr)
else
    call MPI_Allreduce(local_dot, result, 1, MPI_REAL8, MPI_SUM,
    ↪  this%comm, ierr)
end if
```

# Level 1: the norm operation



The fourth Level-1 BLAS operation we are going to implement is the **vector norm**:

$$\|\mathbf{x}\|_2 = \sqrt{\mathbf{x}^\top \mathbf{x}} = \left( \sum_{i=1}^{N} x_i^2 \right)^{1/2}, \quad \mathbf{x} \in \mathbb{R}^N.$$

This operation is similar to the scalar product, since we need to **reduce** the contributions from all the processes to compute the final result.

We can therefore implement this operation as a **method** of the distributed vector type, as before we add the method signature to the type definition:

```
type :: mpi_ddistributed_vector
  ...
contains
  ...
  procedure, pass(this) :: dnrm2_dist
end type mpi_ddistributed_vector
```

# Level 1: the norm operation



Then we can implement the method using the BLAS `dnrm2` routine on the local data, and then performing a global reduction of the local contributions to compute the final result, using the same strategy as before.

The prototype of the method is:

```
subroutine dnrm2_dist(this, result, rank)
  implicit none
  class(mpi_ddistributed_vector), intent(in) :: this
  integer, intent(in), optional :: rank
  real(real64), intent(out) :: result
end subroutine dnrm2_dist
```

# Level 1: the norm operation



The local norm computation is performed using the `dnrm2` routine:

```
local_norm = dnrm2(this%n_local, this%data, 1)**2
```

and then we **perform the reduction**:

```
if (present(rank)) then
    call MPI_Reduce(local_norm, result, 1, MPI_REAL8, MPI_SUM, rank,
    ↪  this%comm, ierr)
else
    call MPI_Allreduce(local_norm, result, 1, MPI_REAL8, MPI_SUM,
    ↪  this%comm, ierr)
end if
```

Finally we take the square root of the result to obtain the final norm, if we are on the root process (or on all processes if no root was specified).

# Level 1: the norm operation



The final step is to take the square root of the result:

```
if (present(rank)) then
  if (myrank == rank) then
    result = sqrt(global_sum)
  end if
else
  result = sqrt(global_sum)
end if
```

With this we have completed the implementation of the Level-1 BLAS operations for our distributed vector type.

# Performance considerations for Level-1 BLAS



The Level-1 BLAS operations we have implemented are all **memory-bound**, since they require a lot of data movement compared to the number of floating-point operations performed.

In a distributed memory setting, the performance of these operations is further limited by the **communication overhead** introduced by the MPI routines used for data distribution and reduction.

Let us try to analyze the performance of the

- axpy,
- dot product,
- norm.

What models can we use?

# Distributed axpy performance model



The distributed axpy operation requires:

- Reading the local parts of vectors $\mathbf{x}$ and $\mathbf{y}$ from memory,
- Writing the updated local part of vector $\mathbf{y}$ to memory.

Therefore, the total data movement is:

$$\text{Data Movement} = 2 \cdot n_{\text{local}} \cdot \text{sizeof(real64)}.$$

The number of floating-point operations is:

$$\text{FLOPs} = 2 \cdot n_{\text{local}}.$$

Thus, the operational intensity is:

$$I = \frac{\text{FLOPs}}{\text{Data Movement}} = \frac{2 \cdot n_{\text{local}}}{2 \cdot n_{\text{local}} \cdot \text{sizeof(real64)}} = \frac{1}{\text{sizeof(real64)}}.$$

This has not changed from the shared memory case.

# The communication cost in distributed axpy



In addition to the memory access costs, the distributed axpy operation incurs a communication cost due to the distribution of vectors across MPI processes.

However, since the axpy operation is **embarrassingly parallel**, there is no need for inter-process communication during the computation itself.

Therefore, the **communication cost is negligible** for the axpy operation, and the *performance is primarily determined by the memory bandwidth of the local memory*.

Let us run some experiments to confirm this.

# axpy performance bechmark



We can write a single benchmark program to test the performance of the distributed axpy operation for both **strong** and **weak scaling**.

We read the type of experiment from the command line arguments:

```
if (rank == 0) then
  call get_command_argument(1, scaling_type)
  call get_command_argument(2, arg_buffer)
  read(arg_buffer, *) n_size_input

  if (trim(scaling_type) /= 'strong' .and. trim(scaling_type) /= 'weak') then
      write(error_unit, '(A)') "Error: scaling_type must be 'strong' or 'weak'"
      call MPI_Abort(MPI_COMM_WORLD, 1, ierr)
  end if
end if
```

# axpy performance bechmark



Then we set the global vector size based on the scaling type:

```
! Broadcast arguments to all ranks
call MPI_Bcast(scaling_type, 20, MPI_CHARACTER, 0, MPI_COMM_WORLD, ierr)
call MPI_Bcast(n_size_input, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)

! Compute global size based on scaling type
if (trim(scaling_type) == 'strong') then
  n_global = n_size_input
else  ! weak scaling
  n_global = n_size_input * nprocs
end if
```

Recall: **strong scaling** means fixed problem size, **weak scaling** means fixed problem size per process.

# axpy performance bechmark

## 2 Distributed Level-1 BLAS

We are now ready to create the distributed vectors and run the benchmark:

```
! Initialize vectors and data
call x%dinit(MPI_COMM_WORLD, n_global)
call y%dinit(MPI_COMM_WORLD, n_global)
x%data = 1.0_real64
y%data = 2.0_real64
call MPI_Barrier(MPI_COMM_WORLD, ierr)
call y%daxpy_dist(alpha, x) ! Warmup run
call MPI_Barrier(MPI_COMM_WORLD, ierr)
! Timing runs
start_time = MPI_Wtime()
do i = 1, num_trials
  call y%daxpy_dist(alpha, x)
end do
end_time = MPI_Wtime()
elapsed_time = (end_time - start_time) / num_trials
```

# axpy performance bechmark



Finally we gather the results:

```
! Gather timing statistics
call MPI_Reduce(elapsed_time, max_time, 1, MPI_REAL8, MPI_MAX, 0, MPI_COMM_WORLD, ierr)
call MPI_Reduce(elapsed_time, min_time, 1, MPI_REAL8, MPI_MIN, 0, MPI_COMM_WORLD, ierr)
call MPI_Reduce(elapsed_time, avg_time, 1, MPI_REAL8, MPI_SUM, 0, MPI_COMM_WORLD, ierr)
```

Which we can then print from the root process:

```
avg_time = avg_time / nprocs
write(output_unit, *)
write(output_unit, '(A, I12)') "Global vector size:     ", n_global
write(output_unit, '(A, I12)') "Local vector size:      ", x%n_local
write(output_unit, '(A, F12.6, A)') "Average time:           ", avg_time * 1000, " ms"
write(output_unit, '(A, F12.6, A)') "Min time:               ", min_time * 1000, " ms"
write(output_unit, '(A, F12.6, A)') "Max time:               ", max_time * 1000, " ms"
write(output_unit, '(A, F12.2, A)') "Throughput:             ", &
    (n_global * 3.0_real64 * 8.0_real64) / (avg_time * 1.0e9), " GB/s"
write(output_unit, '(A, F12.2, A)') "Performance:            ", &
    (n_global * 2.0_real64) / (avg_time * 1.0e9), " GFLOP/s"
```

# axpy performance experiment setup



We run the benchmark on the AMELIA cluster at IAC-CNR, which is the same we used to measure network performance.

```
#!/bin/bash
#SBATCH --job-name=axpy_scaling_64ppn
#SBATCH --nodes=20
#SBATCH --ntasks-per-node=64
#SBATCH --time=00:30:00
#SBATCH --partition=prod-gn
#SBATCH --mem=900Gb
#SBATCH --output=axpy_%j.out
#SBATCH --error=axpy_%j.err

# Load Intel oneAPI modules
module load intel/oneapi/intel_MKL-2023.2.0 intel/oneapi/intel_MPI-2023.2.0
```

# axpy performance experiment setup

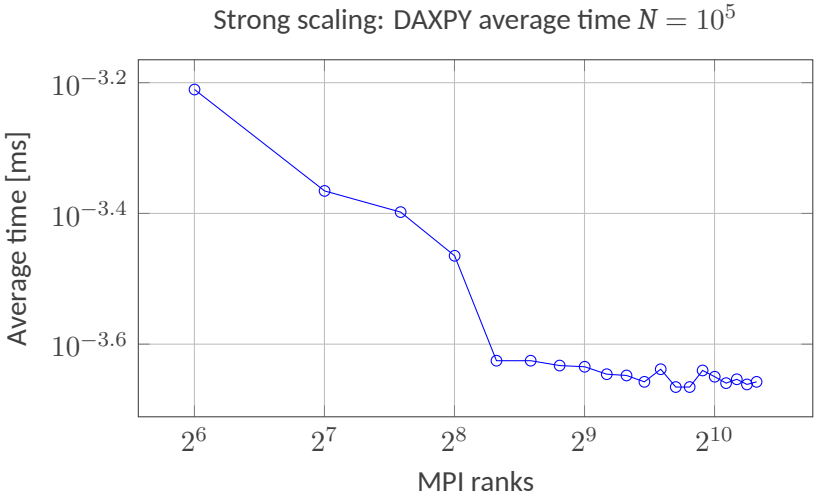


For the **strong scaling** we launch the benchmark:

```
N_GLOBAL_STRONG=100000   # fixed total vector size
echo " STRONG SCALING TEST"
echo " Global vector size = ${N_GLOBAL_STRONG}"
for NODES in $(seq 1 20); do
    NTASKS=$((NODES * 64))
    echo
    echo "Strong scaling run:"
    echo "  Nodes     : $NODES"
    echo "  MPI ranks: $NTASKS"
    echo "  N_global : $N_GLOBAL_STRONG"
    mpirun -np $NTASKS ./mpi_axpy_scaling strong $N_GLOBAL_STRONG
    echo "----------------------------------------"
done
```

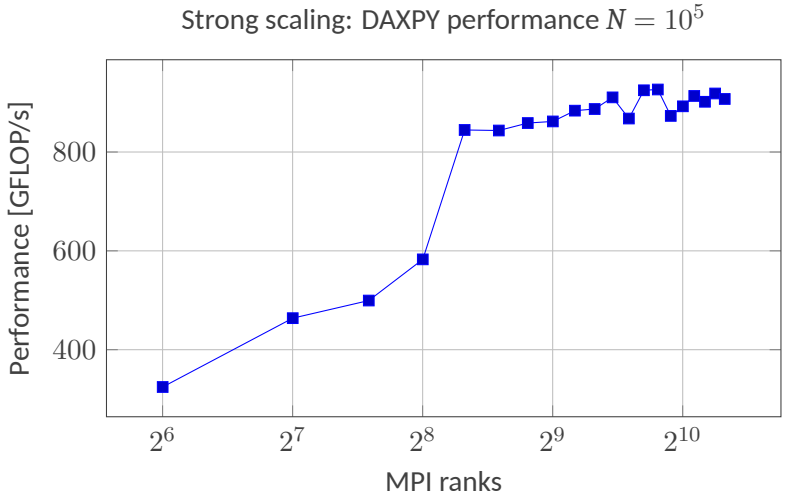
# axpy performance experiment: strong scaling





# axpy performance experiment: strong scaling





# Strong scaling analysis



The strong scaling results show that the execution time **decreases** as we increase the number of MPI ranks, which is the expected behavior. However, the **rate of decrease** slows down significantly after 256 ranks.

Key observations:

- From 64 to 256 ranks: execution time decreases from 0.616 ms to 0.343 ms ($\approx 45\%$ reduction),
- From 256 to 1280 ranks: execution time decreases from 0.343 ms to 0.220 ms ($\approx 36\%$ reduction).

This indicates that we are approaching the **communication and memory bandwidth limits** of the system.

# Strong scaling efficiency



The performance per rank (in GFLOP/s) shows a **significant improvement** from 64 to 320 ranks, increasing from $\approx 324$ GFLOP/s to $\approx 845$ GFLOP/s.

Beyond 320 ranks, the performance **stabilizes** around 900 GFLOP/s with minor fluctuations.

This behavior is typical for memory-bound operations:

- Initial improvement due to better cache utilization and memory bandwidth saturation,
- Plateau effect due to the inherent memory bandwidth limitation of individual compute nodes.

# Strong scaling saturation



The saturation in performance suggests that we have **reached the memory bandwidth limit** of the underlying hardware at approximately 900 GFLOP/s.

This is consistent with the memory-bound nature of the axpy operation, where:

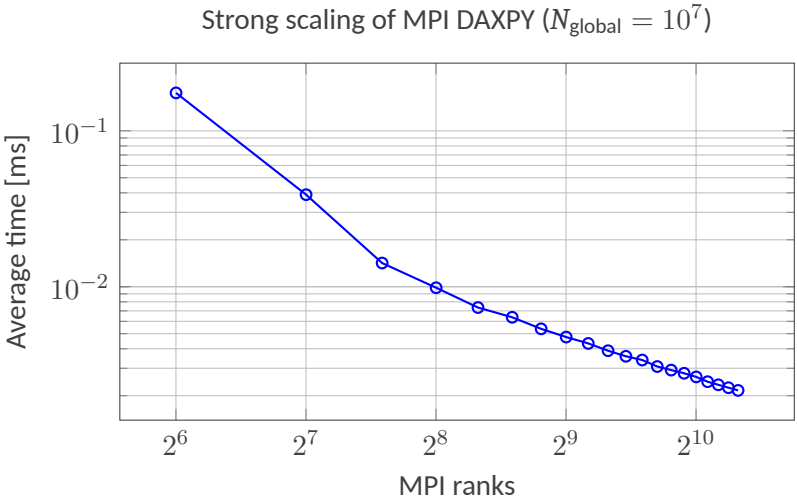
- The operational intensity is very low ($I = 1/8$ for 64-bit floats),
- The computation is limited by memory bandwidth, not floating-point performance.

Further improvements would require either:

- Increasing the problem size (to improve cache reuse), and let's do it!

# axpy performance experiment: strong scaling

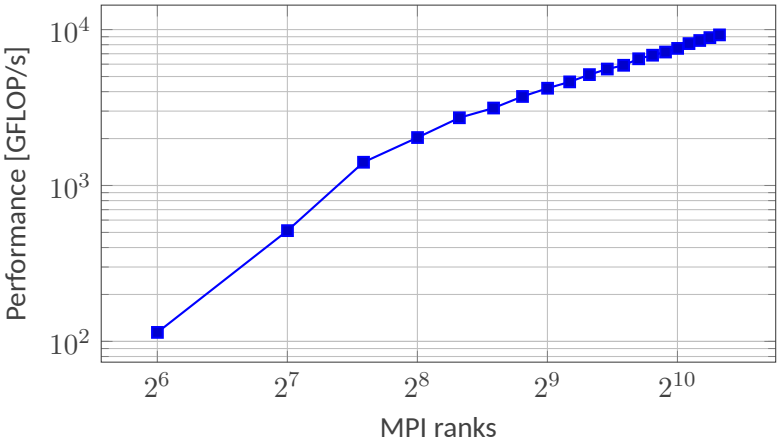




# axpy performance experiment: strong scaling



Strong scaling performance of MPI DAXPY ($N_{\text{global}} = 10^7$)



# axpy performance experiment: weak scaling setup

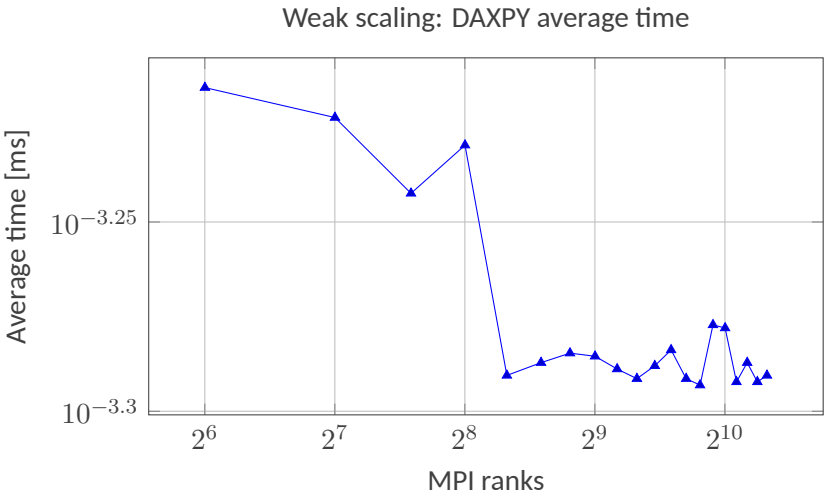


For the **weak scaling** we launch the benchmark:

```
N_LOCAL_WEAK=1562   # local size per MPI rank
echo " WEAK SCALING TEST"
echo " Local vector size per rank = ${N_LOCAL_WEAK}"
for NODES in $(seq 1 20); do
    NTASKS=$((NODES * 64))
    N_GLOBAL_WEAK=$((NTASKS * N_LOCAL_WEAK))
    echo
    echo "Weak scaling run:"
    echo "  Nodes    : $NODES"
    echo "  MPI ranks: $NTASKS"
    echo "  N_global : $N_GLOBAL_WEAK (${N_LOCAL_WEAK} per rank)"
    mpirun -np $NTASKS ./mpi_axpy_scaling weak $N_LOCAL_WEAK
    echo "----------------------------------------"
done
```

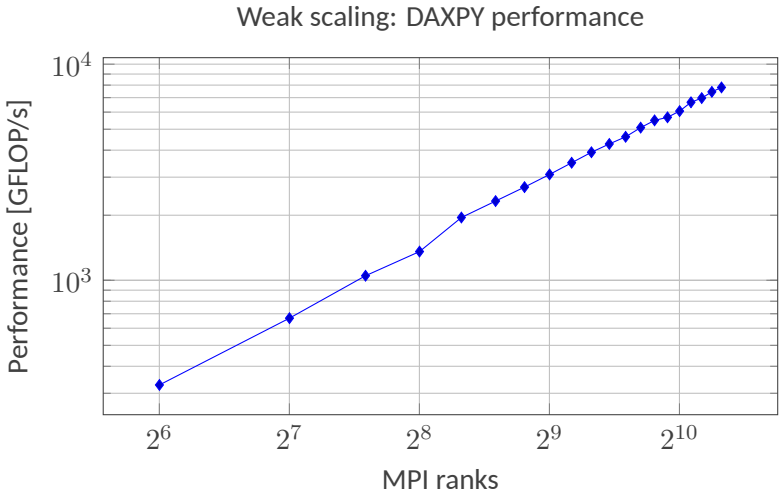
# axpy performance experiment: weak scaling





# axpy performance experiment: weak scaling





# Weak Scaling Results Analysis



The weak scaling results for the distributed axpy operation demonstrate that as we increase the number of MPI ranks while maintaining a constant local vector size per rank, the execution time remains relatively stable at approximately 0.5 ms.

- From 64 to 1280 ranks: execution time fluctuates between 0.51 and 0.61 ms, showing excellent stability,
- The performance in GFLOP/s scales nearly linearly with the number of ranks, indicating efficient distributed memory utilization,
- This demonstrates that the **embarrassingly parallel** nature of the axpy operation is preserved in the distributed implementation.

This behavior is ideal for weak scaling scenarios, where the problem size grows proportionally with the number of processes. The stable execution time confirms that there is no significant communication overhead in the distributed axpy operation.

# Weak Scaling Performance Insights



The weak scaling results for the distributed axpy operation highlight several key insights:

- The constant local computation time across all ranks confirms that the distributed axpy implementation has negligible communication overhead,
- The linear increase in total GFLOP/s (from $\approx 328$ to $\approx 7807$ GFLOP/s) demonstrates perfect computational scaling,
- The local memory bandwidth utilization remains constant, indicating that each rank operates independently without interference from inter-process communication.

Overall, the weak scaling results validate that the distributed axpy operation is well-suited for large-scale parallel computations, with minimal communication overhead and excellent scalability properties.

# Scaling of the dot product and norm operations



We can now investigate the scaling behavior of the **dot product** and **norm** operations, which involve global reductions and are therefore expected to exhibit different scaling characteristics compared to the axpy operation.

The dot product and norm operations require communication between MPI processes to aggregate local results, which can introduce significant overhead, especially as the number of processes increases.

We will analyze both **strong** and **weak scaling** for these operations to understand their performance limits.

# Writing the benchmark

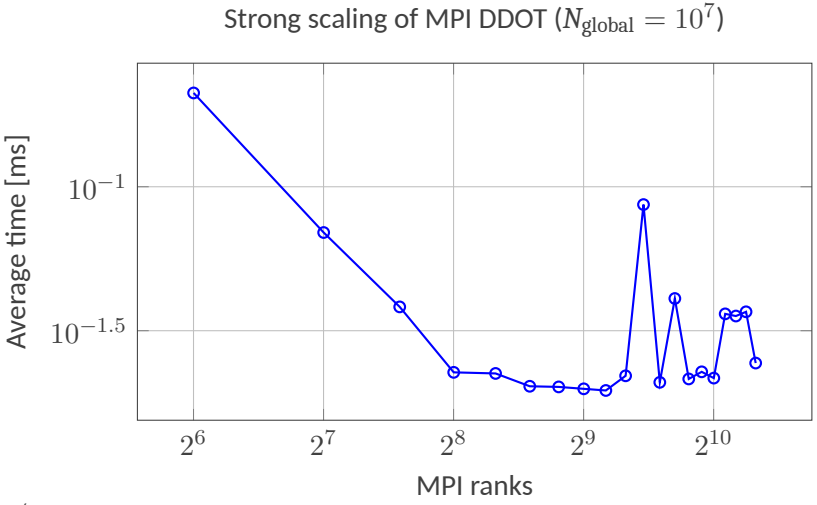
## 2 Distributed Level-1 BLAS

The initial part of the benchmark program is similar to the axpy benchmark, decide the scaling type and set the global vector size accordingly. The main difference is in the timing section, where we replace the axpy call with either the dot product or norm call:

```
call x%dinit(MPI_COMM_WORLD, n_global)
call y%dinit(MPI_COMM_WORLD, n_global)
x%data = 1.0_real64
y%data = 2.0_real64
call MPI_Barrier(MPI_COMM_WORLD, ierr)
call x%ddot_dist(y, dot_value)
call MPI_Barrier(MPI_COMM_WORLD, ierr)
start_time = MPI_Wtime()
do i = 1, num_trials
  call x%ddot_dist(y, dot_value)
end do
end_time = MPI_Wtime()
elapsed_time = (end_time - start_time) / num_trials
```

# ddot performance experiment: strong scaling

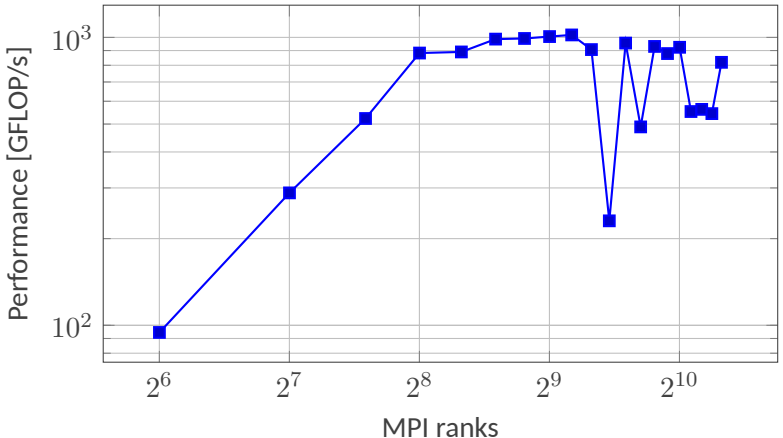




# ddot performance experiment: strong scaling



Strong scaling performance of MPI DDOT ($N_{\text{global}} = 10^7$)



# Strong Scaling Analysis: MPI DDOT ($N_{\text{global}} = 10^7$)



The strong scaling results for the distributed dot product reveal interesting behavior patterns:

- **Initial strong scaling (64–576 ranks):** Execution time decreases from 0.212 ms to 0.020 ms, showing nearly ideal scaling behavior.
- **Peak performance (512–576 ranks):** Approximately 1000 GFLOP/s, indicating efficient local computation and reduction.
- **Performance degradation (704+ ranks):** Significant variability and performance drops appear, suggesting communication overhead dominates.

The critical observation is that performance becomes unstable beyond 576 ranks, with execution time fluctuating between 0.020 ms and 0.087 ms.

# Understanding the Scaling Breakdown



**Why does performance degrade at high rank counts?**

- Local vector size: $n_{\text{local}} = \frac{10^7}{N_{\text{ranks}}}$
- At 1280 ranks: $n_{\text{local}} \approx 7800$ elements
- Too small for effective cache utilization

**Communication cost dominates:**

- MPI reduction becomes a bottleneck
- Synchronization overhead exceeds computation time
- Network latency not fully hidden by computation

**Key insight:** Unlike embarrassingly parallel axpy, dot product requires global synchronization via MPI_Allreduce, which becomes increasingly expensive as rank count grows.

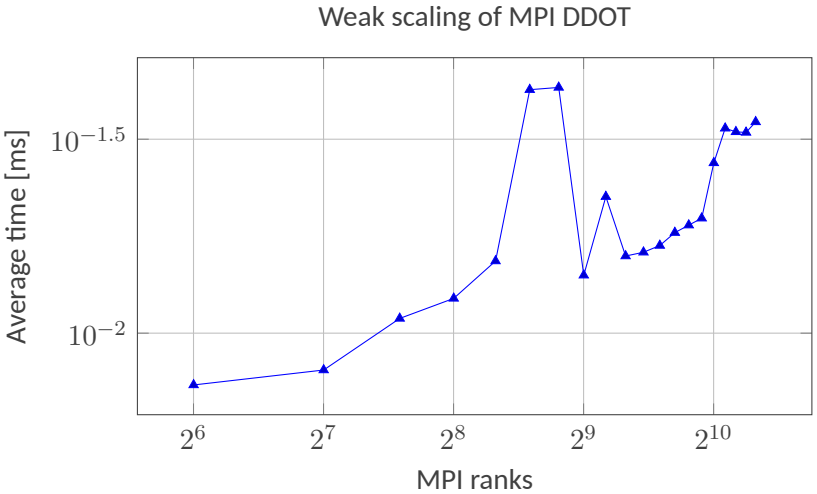
# Recommendations for Practical Use



- **Optimal configuration:** 256–576 ranks with problem size $N \geq 10^6$ per rank
  - Maintains local vector size large enough for efficient computation
  - Communication overhead remains manageable
- **Weak scaling preferred:** For dot product and norm operations, weak scaling (constant work per rank) provides better performance predictability
- **Avoid extreme decomposition:** Do not distribute to more ranks than necessary; overhead grows quadratically with rank count for global reductions

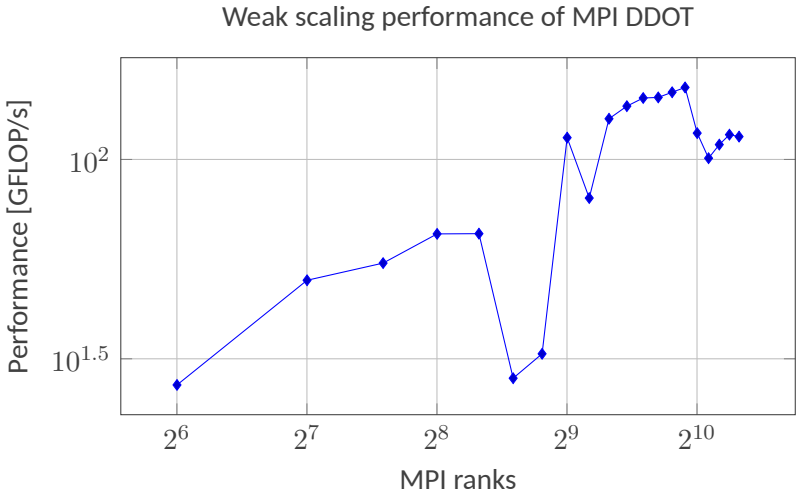
# ddot performance experiment: weak scaling





# ddot performance experiment: weak scaling





# Weak Scaling of Distributed DDOT with MPI_Allreduce



**Setup:** Each MPI rank maintains a fixed local vector of 1562 elements. The global dot product requires `MPI_Allreduce` to aggregate local contributions across all ranks.

**Execution time behavior:**

- Scales well up to ≈320 ranks with gradual time increase
- Significant spikes at 384–448 and 1024+ ranks indicate communication bottlenecks
- Variability suggests network congestion and synchronization overhead

**Performance scaling:**

- Peak performance ≈150 GFLOP/s around 960 ranks
- Drops align with execution time spikes, confirming communication dominates
- Unlike axpy, performance is latency-bound, not compute-bound

# Weak Scaling of Distributed DDOT with MPI_Allreduce



**Key insight:** At high rank counts, each rank's local vector is too small to hide the cost of the global all-reduce operation. The computation time becomes negligible compared to synchronization overhead.

## Communication avoiding algorithms

To mitigate these issues, research has moved towards **communication-avoiding algorithms** that try to reduce the number of global synchronizations required, for example by restructuring computations to perform more local work before communicating, or by reformulating algorithms to reduce the frequency of reductions.

# Conclusion and next steps



We have

- ✔ Implemented Level-1 BLAS operations for distributed vectors using MPI,
- ✔ Analyzed performance characteristics for axpy, dot product, and norm,
- ✔ Identified scaling limits due to communication overhead in reduction operations

Next steps:

- Explore Level-2 and Level-3 BLAS operations in distributed settings.

# High Performance Linear Algebra

Lecture 11: Distributed BLAS level 2

Ph.D. program in High Performance Scientific Computing

**Fabio Durastante** **Pasqua D'Ambra** **Salvatore Filippone**

January 26, 2026 — 14.00:16.00

Dipartimento
di Matematica
Università di Pisa

# Last time on High Performance Linear Algebra



We have

- Described the Message Passing Interface (MPI) programming model,
- Implemented Level-1 BLAS operations for distributed vectors using MPI,
- Analyzed performance characteristics for axpy, dot product, and norm,
- Identified scaling limits due to communication overhead in reduction operations

Next steps:

- Discuss data distribution strategies for distributed matrices,
- Implement Level-2 BLAS operations for distributed matrices and vectors,
- Explore the Level-2 and Level-3 BLAS operations in distributed settings.

# Table of Contents

2 Distributed matrices

▶ Distributed matrices
Data layout strategies
2D-Block Cyclic Implementation
A distributed matrix defined on a descriptor
The distributed GEMV operation

# Distributed matrices



When dealing with large matrices that cannot fit into the memory of a single node, we need to **distribute the matrix across multiple nodes** of our distributed-memory environment.

How can we do that?

- we have already partially addressed this question when discussing the construction of matrix-vector products and matrix-matrix products in a shared-memory context;
- we have discussed this for the case of distributed vectors last time.

# Key concerns in data layout



When choosing a data layout for dense matrix computations on distributed-memory systems, we must consider:

**Load Balance**

- Split work evenly among processors
- Avoid idle processors during computation

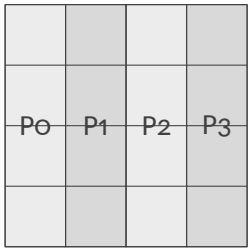
**Computational Efficiency**

- Utilize Level 3 BLAS on local data
- Leverage memory hierarchy

These requirements often conflict and require careful trade-offs!

# 1D Block Column Distribution






**Characteristics:**

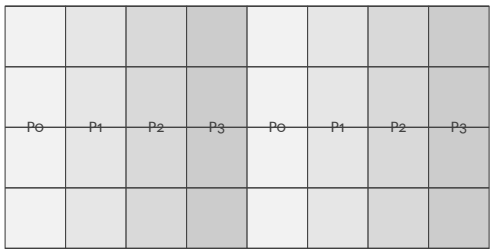
- Each process stores one block of contiguous columns
- Column $k$ goes to process $\lfloor (k-1)/n_c \rfloor \mod P$

**Problems:**

- **Poor load balance**: once columns are completed, processes become idle
- Not suitable for matrix operations like Gaussian elimination

# 1D Cyclic Column Distribution






**Characteristics:**

- Column $k$ assigned to process $(k-1) \mod P$
- Single columns are interleaved across processes

**Advantages:**

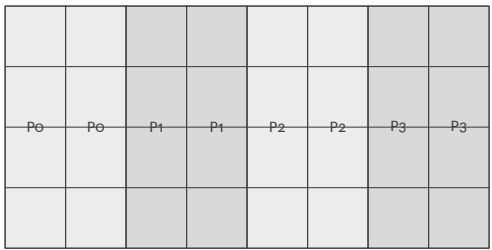
- Better load balance
- Good work distribution

**Problems:**

- Cannot use Level 2/3 BLAS efficiently (only single columns)
- Poor memory hierarchy utilization

# 1D Block-Cyclic Column Distribution






**Characteristics:**

- Choose block size $NB$
- Column block $k$ assigned to process $\lfloor (k-1)/NB \rfloor \mod P$

**Advantages:**

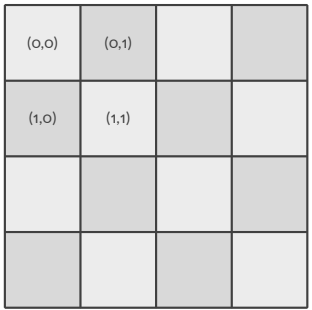
- Reasonable load balance
- Can use Level 2/3 BLAS locally

**Problems:**

- Serial bottleneck: factorization occurs on one process
- Limited parallelism in factorization step

# 2D Block-Cyclic Distribution






Processors in $P_r \times P_c$ grid

**Characteristics:**

- Arrange $P$ processes in $P_r \times P_c$ rectangular grid
- Block size parameters: $MB \times NB$
- Block $(i, j)$ goes to process $(i \bmod P_r, j \bmod P_c)$

**Advantages:**

- Excellent load balance
- Allows $P_c$-fold parallelism in columns
- Can use Level 2/3 BLAS
- Good scalability

**This is the standard layout used by ScaLAPACK! But how do we implement it?**

# Comparison of Distribution Strategies



| Strategy | Load Balance | BLAS Use | Scalability | Complexity |
|---|---|---|---|---|
| 1D Block | Poor | Good | Bad | Simple |
| 1D Cyclic | Good | Poor | Fair | Simple |
| 1D Block-Cyclic | Fair | Fair | Fair | Moderate |
| 2D Block-Cyclic | Excellent | Good | Excellent | Complex |

**Key Trade-offs:**

- Load balance vs. computational efficiency
- Simple schemes vs. scalability
- The 2D block-cyclic strategy achieves the best overall balance

# Implementing the 2D Block-Cyclic Distribution



We need to map global matrix indices to local storage on each process.

**Global to Local Index Mapping (2D block-cyclic):**

- Communicator with $P = P_r \times P_c$ processes
- Global matrix of size $M \times N$
- Process grid of size $P_r \times P_c$
- Block size $MB \times NB$
- Source process $(R_{\text{src}}, C_{\text{src}})$
- Use 0-based global indices: $i_0 = i - 1$, $j_0 = j - 1$
- Global block indices:

$$b_r = \left\lfloor \frac{i_0}{MB} \right\rfloor, \quad b_c = \left\lfloor \frac{j_0}{NB} \right\rfloor$$

# Implementing the 2D Block-Cyclic Distribution



We need to map global matrix indices to local storage on each process.

**Global to Local Index Mapping (2D block-cyclic):**

- Communicator with $P = P_r \times P_c$ processes
- Global matrix of size $M \times N$
- Process grid of size $P_r \times P_c$
- Block size $MB \times NB$
- Source process $(R_{\text{src}}, C_{\text{src}})$
- Use 0-based global indices: $i_0 = i - 1$, $j_0 = j - 1$
- Owning process:

$$p_r = (b_r - R_{\text{src}}) \bmod P_r, \quad p_c = (b_c - C_{\text{src}}) \bmod P_c$$

# Implementing the 2D Block-Cyclic Distribution



We need to map global matrix indices to local storage on each process.

**Global to Local Index Mapping (2D block-cyclic):**

- Communicator with $P = P_r \times P_c$ processes
- Global matrix of size $M \times N$
- Process grid of size $P_r \times P_c$
- Block size $MB \times NB$
- Source process $(R_{\text{src}}, C_{\text{src}})$
- Use 0-based global indices: $i_0 = i - 1$, $j_0 = j - 1$
- Local block indices:

$$\ell_r = \left\lfloor \frac{b_r - R_{\text{src}}}{P_r} \right\rfloor, \quad \ell_c = \left\lfloor \frac{b_c - C_{\text{src}}}{P_c} \right\rfloor$$

# Implementing the 2D Block-Cyclic Distribution



We need to map global matrix indices to local storage on each process.

**Global to Local Index Mapping (2D block-cyclic):**

- Communicator with $P = P_r \times P_c$ processes
- Global matrix of size $M \times N$
- Process grid of size $P_r \times P_c$
- Block size $MB \times NB$
- Source process $(R_{\text{src}}, C_{\text{src}})$
- Use 0-based global indices: $i_0 = i - 1$, $j_0 = j - 1$
- Local indices (1-based, Fortran):

$$\text{local\_row} = \ell_r \cdot MB + (i_0 \bmod MB) + 1$$

$$\text{local\_col} = \ell_c \cdot NB + (j_0 \bmod NB) + 1$$

# Building a descriptor for the 2D process grid



To implement the 2D block-cyclic distribution, we need a descriptor that contains:

- Global matrix dimensions `M`, `N`
- Block sizes `MB`, `NB`
- Process grid dimensions `P_r`, `P_c`
- Leading dimension of local arrays
- Starting indices for submatrices, they are usually called `RSRC` and `CSRC` parameters specify the starting process coordinates for the distribution, and are typically set to $0$.

This descriptor will be used in all distributed matrix operations to correctly map global indices to local storage. We can build it as a Fortran derived type.

# Fortran Descriptor Type

2 Distributed matrices

```
type :: descriptor
    integer :: comm      ! MPI communicator
    integer :: M         ! Global number of rows
    integer :: N         ! Global number of columns
    integer :: MB        ! Block size in rows
    integer :: NB        ! Block size in columns
    integer :: P_r       ! Number of process rows
    integer :: P_c       ! Number of process columns
    integer :: LLD       ! Leading dimension of local array
    integer :: RSRC      ! Row source process
    integer :: CSRC      ! Column source process
end type descriptor
```

This structure encapsulates all necessary information for managing the 2D block-cyclic distribution of matrices across a process grid.

# But how do we init the descriptor?



To initialize the descriptor, we can create an initialization procedure that sets all the fields based on the global matrix size, block sizes, and process grid dimensions.

- This procedure will be called once at the beginning of the program to set up the descriptor.
- It will compute the leading dimension of the local arrays based on the block sizes and process grid.
- The process coordinates $(p_r, p_c)$ must be provided to correctly map back to global indices, they can be retrieved from the MPI rank.

# But how do we init the descriptor?



```
subroutine init(desc, comm, M, N, MB, NB, P_r, P_c, RSRC, CSRC)
    class(descriptor), intent(out) :: desc
    integer, intent(in) :: comm, M, N, MB, NB, P_r, P_c, RSRC, CSRC
    desc%comm = comm
    desc%M = M
    desc%N = N
    desc%MB = MB
    desc%NB = NB
    desc%P_r = P_r
    desc%P_c = P_c
    desc%RSRC = RSRC
    desc%CSRC = CSRC
    desc%LLD = ((M + P_r * MB - 1) / (P_r * MB)) * MB
end subroutine init
```

# A distributed matrix type



We now need to build a **distributed matrix** type that uses the **descriptor** to manage its data.

This can be done by defining a Fortran derived type that contains:

- The local array to store the matrix data,
- The descriptor for the matrix distribution,
- Type-bound procedures for the BLAS matrix operations.

# Distributed Matrix Type Definition



We can define the distributed matrix type as follows:

```
type :: distributed_matrix
    real, allocatable :: local_data(:,:) ! Local matrix storage
    type(descriptor) :: desc              ! Descriptor for distribution
contains
  <Type bound procedures for matrix operations go here>
end type distributed_matrix
```

We don't need to add more fields, as the descriptor contains all necessary information about the global matrix, and the size of the local is in the `desc%NB` and `desc%MB` fields.

# The distributed GEMV operation



The first Level-2 BLAS operation we can implement is the distributed matrix-vector product (GEMV):

$$\mathbf{y} \leftarrow \alpha A\mathbf{x} + \beta\mathbf{y}$$

where $A$ is a distributed matrix, and $\mathbf{x}$ and $\mathbf{y}$ are distributed vectors.

**Key Steps:**

- Each process computes its local contribution to the matrix-vector product.
- A global reduction is performed to combine the local results into the final vector $\mathbf{y}$.

❓ But how do we distribute the vectors $\mathbf{x}$ and $\mathbf{y}$?

# The distributed GEMV operation



The first Level-2 BLAS operation we can implement is the distributed matrix-vector product (GEMV):

$$\mathbf{y} \leftarrow \alpha A\mathbf{x} + \beta\mathbf{y}$$

where $A$ is a distributed matrix, and $\mathbf{x}$ and $\mathbf{y}$ are distributed vectors.

**Key Steps:**

- Each process computes its local contribution to the matrix-vector product.
- A global reduction is performed to combine the local results into the final vector $\mathbf{y}$.

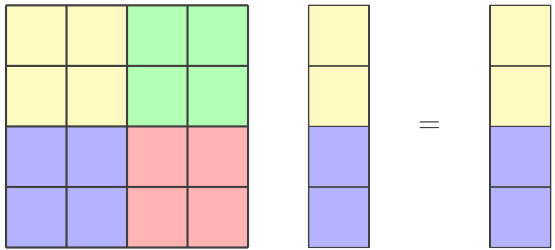
❓ But how do we distribute the vectors $\mathbf{x}$ and $\mathbf{y}$?

❗ We can use a 1D block distribution for the vectors to align with the matrix distribution.

# Distributed GEMV Implementation

## 2 Distributed matrices

We can visualize the distributed GEMV operation with a block cyclic distributed matrix, and a 1D block distributed vector as in the following diagram:

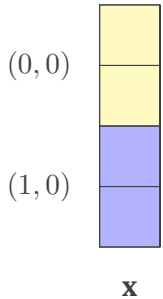



$$\mathbf{y} = A\mathbf{x}, \qquad A \in \mathbb{R}^{N \times N}, \quad \mathbf{x}, \mathbf{y} \in \mathbb{R}^{N \times 1}$$

# Vector Distribution



- Column vector: $x \in \mathbb{R}^{N \times 1}$
- Descriptor parameters:
  - `M = N`
  - `N = 1`
  - `NB = 1`
- Block-cyclic distribution in rows only
- Only **process column** $0$ stores vector data



# What communications are needed for GEMV?



We can represent the distributed GEMV operation as follows:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} A_{00}x_0 + A_{01}x_1 + A_{02}x_2 + A_{03}x_3 \\ A_{10}x_0 + A_{11}x_1 + A_{12}x_2 + A_{13}x_3 \\ A_{20}x_0 + A_{21}x_1 + A_{22}x_2 + A_{23}x_3 \\ A_{30}x_0 + A_{31}x_1 + A_{32}x_2 + A_{33}x_3 \end{bmatrix}$$

This means that:

- All processes with processo column $0$ need $x_0$
- All processes with processo column $1$ need $x_1$
- All processes with processo column $2$ need $x_2$
- All processes with processo column $3$ need $x_3$

# What communications are needed for GEMV?

## 2 Distributed matrices

We can represent the distributed GEMV operation as follows:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} A_{00}x_0 + A_{01}x_1 + A_{02}x_2 + A_{03}x_3 \\ A_{10}x_0 + A_{11}x_1 + A_{12}x_2 + A_{13}x_3 \\ A_{20}x_0 + A_{21}x_1 + A_{22}x_2 + A_{23}x_3 \\ A_{30}x_0 + A_{31}x_1 + A_{32}x_2 + A_{33}x_3 \end{bmatrix}$$

This means that:

- All processes with processo column $0$ need $x_0$
- All processes with processo column $1$ need $x_1$
- All processes with processo column $2$ need $x_2$
- All processes with processo column $3$ need $x_3$

# What communications are needed for GEMV?



We can represent the distributed GEMV operation as follows:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} A_{00}x_0 + A_{01}x_1 + A_{02}x_2 + A_{03}x_3 \\ A_{10}x_0 + A_{11}x_1 + A_{12}x_2 + A_{13}x_3 \\ A_{20}x_0 + A_{21}x_1 + A_{22}x_2 + A_{23}x_3 \\ A_{30}x_0 + A_{31}x_1 + A_{32}x_2 + A_{33}x_3 \end{bmatrix}$$

This means that:

- All processes with processo column $0$ need $x_0$
- All processes with processo column $1$ need $x_1$
- All processes with processo column $2$ need $x_2$
- All processes with processo column $3$ need $x_3$

# What communications are needed for GEMV?



We can represent the distributed GEMV operation as follows:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} A_{00}x_0 + A_{01}x_1 + A_{02}x_2 + A_{03}x_3 \\ A_{10}x_0 + A_{11}x_1 + A_{12}x_2 + A_{13}x_3 \\ A_{20}x_0 + A_{21}x_1 + A_{22}x_2 + A_{23}x_3 \\ A_{30}x_0 + A_{31}x_1 + A_{32}x_2 + A_{33}x_3 \end{bmatrix}$$

This means that:

- All processes with processo column $0$ need $x_0$
- All processes with processo column $1$ need $x_1$
- All processes with processo column $2$ need $x_2$
- All processes with processo column $3$ need $x_3$

# What communications are needed for GEMV?



We can represent the distributed GEMV operation as follows:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} A_{00}x_0 + A_{01}x_1 + A_{02}x_2 + A_{03}x_3 \\ A_{10}x_0 + A_{11}x_1 + A_{12}x_2 + A_{13}x_3 \\ A_{20}x_0 + A_{21}x_1 + A_{22}x_2 + A_{23}x_3 \\ A_{30}x_0 + A_{31}x_1 + A_{32}x_2 + A_{33}x_3 \end{bmatrix}$$

This means that:

- All processes with processo column $0$ need $x_0$
- All processes with processo column $1$ need $x_1$
- All processes with processo column $2$ need $x_2$
- All processes with processo column $3$ need $x_3$
- Each process computes its local contribution to **y** and then **reduce along the column**.

# What is the right communication function?



To implement the required communication pattern, we can use:

- We need to broadcast the vector segments $x_0, x_1, x_2, x_3$ along the process columns.

$$x_0 \to (0,0), (1,0), (2,0), (3,0)$$

$$x_1 \to (0,1), (1,1), (2,1), (3,1)$$

$$x_2 \to (0,2), (1,2), (2,2), (3,2)$$

$$x_3 \to (0,3), (1,3), (2,3), (3,3)$$

- We need **a communicator for each process column** to perform these broadcasts.

# Building communicators for GEMV



We need to create communicators for the process rows and columns:

```
dims(1) = P_r ! Define process grid dimensions
dims(2) = P_c
periods(1) = .false.
periods(2) = .false.
! Create Cartesian communicator
call MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, .true., cart_comm,
↪  ierr)
! Split into row and column communicators
call MPI_Comm_rank(cart_comm, cart_rank, ierr)
call MPI_Cart_coords(cart_comm, cart_rank, 2, coords, ierr)
call MPI_Comm_split(cart_comm, coords(1), coords(2), row_comm, ierr)
call MPI_Comm_split(cart_comm, coords(2), coords(1), col_comm, ierr)
```

These communicators allow us to perform broadcasts and reductions efficiently along the rows and columns of the process

# Details and signature of the new MPI functions



The subroutine `MPI_Cart_create` has signature:

```
call MPI_Cart_create(comm_old, ndims, dims, periods, reorder,
↪  comm_cart, ierr)
```

where:

- `comm_old`: input communicator (e.g., `MPI_COMM_WORLD`)
- `ndims`: number of dimensions (2 for a 2D grid)
- `dims`: array specifying the size of each dimension
- `periods`: array specifying whether each dimension is periodic
- `reorder`: logical flag to allow process rank reordering
- `comm_cart`: output Cartesian communicator
- `ierr`: error code

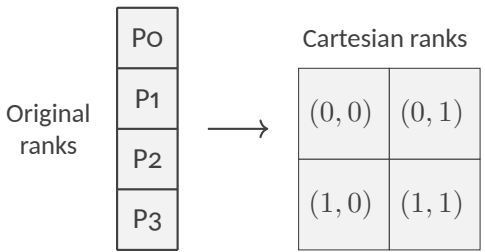
# Details and signature of the new MPI functions



The subroutine `MPI_Cart_create` has signature:

```
call MPI_Cart_create(comm_old, ndims, dims, periods, reorder,
↪  comm_cart, ierr)
```

This subroutine creates a **Cartesian communicator** based on the specified dimensions and periodicity, i.e., a 2D grid of processes.



# Details and signature of the new MPI functions



The subroutine `MPI_Comm_split` has signature:

```
call MPI_Comm_split(comm, color, key, newcomm, ierr)
```

where:

- `comm`: input communicator
- `color`: integer that determines the new communicator grouping
- `key`: integer that determines the rank ordering in the new communicator
- `newcomm`: output new communicator
- `ierr`: error code

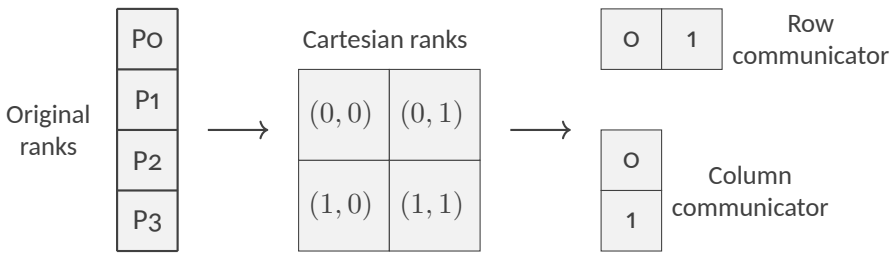
# Details and signature of the new MPI functions



The subroutine `MPI_Comm_split` has signature:

```
call MPI_Comm_split(comm, color, key, newcomm, ierr)
```

This subroutine **splits an existing communicator** into multiple new communicators based on the `color` parameter, allowing for the creation of row and column communicators from the Cartesian grid.



# An example of communicator creation and splitting



```
program communicator_example
  use mpi
  use iso_fortran_env, only: output_unit
  implicit none
  integer :: ierr, rank, size
  integer :: cart_comm, row_comm, col_comm
  integer :: dims(2), coords(2)
  logical :: periods(2)
  integer :: myx, myy
  call MPI_Init(ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
  call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
```

# An example of communicator creation and splitting



```
! Define process grid dimensions
dims(1) = 2  ! Number of process rows
dims(2) = size / 2  ! Number of process columns
periods(1) = .false.
periods(2) = .false.

! Create Cartesian communicator
call MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, .true.,
↪  cart_comm, ierr)
call MPI_Cart_coords(cart_comm, rank, 2, coords, ierr)
```

# An example of communicator creation and splitting



```
  ! Split into row and column communicators
  call MPI_Comm_split(cart_comm, coords(1), coords(2), row_comm,
  ↪  ierr)
  call MPI_Comm_split(cart_comm, coords(2), coords(1), col_comm,
  ↪  ierr)
  ! Print ranks in each communicator for verification
  call MPI_Comm_rank(row_comm, myx, ierr)
  call MPI_Comm_rank(col_comm, myy, ierr)
  write(output_unit, *) 'Global Rank:', rank, 'Row Comm Rank:', myx,
  ↪  'Col Comm Rank:', myy
  call MPI_Finalize(ierr)
end program communicator_example
```

# An example of communicator creation and splitting



We can compile and run this program with 4 processes:

```
mpifort -o communicator_example communicator_example.f90
mpirun -np 4 ./communicator_example
```

The result will be something like:

```
Global Rank:      3 Row Comm Rank:        1 Col Comm Rank:      1
Global Rank:      0 Row Comm Rank:        0 Col Comm Rank:      0
Global Rank:      1 Row Comm Rank:        1 Col Comm Rank:      0
Global Rank:      2 Row Comm Rank:        0 Col Comm Rank:      1
```

# Packaging these communicators in our descriptor



To facilitate the use of these communicators in our distributed matrix operations, we can add them as fields in our `descriptor` type:

```
type :: descriptor
  integer :: comm      ! MPI communicator
  integer :: M, N      ! Global number of rows/columns
  integer :: MB, NB    ! Block size in rows/columns
  integer :: P_r, P_c  ! Number of process rows/columns
  integer :: LLD       ! Leading dimension of local array
  integer :: RSRC, CSRC     ! Row, Column source process
  integer :: cart_comm, row_comm, col_comm ! Cached communicators
  integer :: myrow, mycol
end type descriptor
```

And modify the initialization routine to set these fields accordingly.

# Modifying the descriptor init to include communicators



We can modify the `init` subroutine to create and store the communicators in the descriptor:

```
subroutine init(desc, comm, M, N, MB, NB, P_r, P_c, RSRC, CSRC)
  implicit none
  class(descriptor), intent(out) :: desc
  integer, intent(in) :: comm, M, N, MB, NB, P_r, P_c, RSRC, CSRC
  ! Local variables
  integer :: ierr, dims(2), coords(2)
  integer :: size
  logical :: periods(2)
```

# Modifying the descriptor init to include communicators



```
! standard initialization (same as before)
desc%comm = comm
desc%M = M
desc%N = N
desc%MB = MB
desc%NB = NB
desc%P_r = P_r
desc%P_c = P_c
desc%RSRC = RSRC
desc%CSRC = CSRC
desc%LLD = ((M + P_r * MB - 1) / (P_r * MB)) * MB
```

# Modifying the descriptor init to include communicators



To create the Cartesian, row, and column communicators, we need to check if the distribution is really of a 2D nature (i.e., $P_r > 1$ or $P_c > 1$):

```
! Create Cartesian communicators: created for any 2D/1D distribution
if ( P_r > 1 .or. P_c > 1 ) then
    ! Define process grid dimensions
    dims(1) = P_r
    dims(2) = P_c
    periods(1) = .false.
    periods(2) = .false.
    call MPI_Cart_create(comm, 2, dims, periods, .true.,
    ↪  desc%cart_comm, ierr)
    if ( ierr /= MPI_SUCCESS ) then
```

# Modifying the descriptor init to include communicators



```
    print *, 'Error creating Cartesian communicator'
    call MPI_Abort(comm, ierr, stat)
    stop
  end if
  if ( desc%cart_comm == MPI_COMM_NULL ) then
    desc%row_comm = MPI_COMM_NULL
    desc%col_comm = MPI_COMM_NULL
    desc%my_row = -1
    desc%my_col = -1
    return
  end if
  call MPI_Comm_rank(desc%cart_comm, rank, ierr)
```

# Modifying the descriptor init to include communicators



```
if ( ierr /= MPI_SUCCESS ) then
  print *, 'Error getting rank in Cartesian communicator'
  call MPI_Abort(comm, ierr, stat)
  stop
end if

call MPI_Cart_coords(desc%cart_comm, rank, 2, coords, ierr)
if ( ierr /= MPI_SUCCESS ) then
  print *, 'Error getting coordinates in Cartesian communicator'
  call MPI_Abort(comm, ierr, stat)
  stop
end if
```

# Modifying the descriptor init to include communicators



```
    desc%my_row = coords(1)
    desc%my_col = coords(2)

    ! Split into row and column communicators
    call MPI_Comm_split(desc%cart_comm, coords(1), coords(2),
    ↪  desc%row_comm, ierr)
    call MPI_Comm_split(desc%cart_comm, coords(2), coords(1),
    ↪  desc%col_comm, ierr)
else
    ! For 1D distributions, use the original communicator
    desc%cart_comm = comm
    desc%row_comm = comm
```

# Modifying the descriptor init to include communicators



```
    desc%col_comm = comm
    desc%my_row = 0
    desc%my_col = 0
end if
end subroutine init
```

- This modification ensures that each distributed matrix has access to the necessary communicators for efficient parallel operations.
- The process coordinates `myrow` and `mycol` are also stored for easy reference.
- We have added error checks to ensure the communicator creation is successful.

# Global to Local and Local to Global Index Mapping



Now that we have the communicators set up, we also need to implement the **global to local** and **local to global** index mapping functions.

These functions are useful to query and write specific elements of the distributed matrix.

```
subroutine global_to_local(desc, i_global, j_global, &
  p_r, p_c, i_local, j_local)
  implicit none
  class(descriptor), intent(in) :: desc
  integer, intent(in)  :: i_global, j_global
  integer, intent(out) :: p_r, p_c
  integer, intent(out) :: i_local, j_local

  integer :: ig, jg          ! 0-based global element indices
  integer :: ib, jb          ! absolute block indices
```

# Global to Local and Local to Global Index Mapping



```
integer :: off_i, off_j     ! offset inside block (0-based)
integer :: first_b_r, first_b_c
integer :: lbr, lbc         ! local block indices

! Convert to 0-based element indices
ig = i_global - 1
jg = j_global - 1
! Absolute block indices
ib = ig / desc%MB
jb = jg / desc%NB
! Offsets inside their blocks
off_i = mod(ig, desc%MB)
off_j = mod(jg, desc%NB)
```

# Global to Local and Local to Global Index Mapping



```
  ! Owning process coordinates (0..P_r-1, 0..P_c-1)
  p_r = mod(ib - desc%RSRC, desc%P_r)
  p_c = mod(jb - desc%CSRC, desc%P_c)
  ! First absolute block index assigned to that process
  first_b_r = mod(desc%RSRC + p_r, desc%P_r)
  first_b_c = mod(desc%CSRC + p_c, desc%P_c)
  ! Local block index on owning process (non-negative)
  lbr = (ib - first_b_r) / desc%P_r
  lbc = (jb - first_b_c) / desc%P_c
  ! Local 1-based indices in Fortran storage
  i_local = lbr * desc%MB + off_i + 1
  j_local = lbc * desc%NB + off_j + 1
end subroutine global_to_local
```

# Global to Local and Local to Global Index Mapping



While the converse mapping is given by:

```
subroutine local_to_global(desc, p_r, p_c, &
  i_local, j_local, &
  i_global, j_global)
  implicit none
  class(descriptor), intent(in) :: desc
  integer, intent(in)  :: p_r, p_c
  integer, intent(in)  :: i_local, j_local
  integer, intent(out) :: i_global, j_global

  integer :: il, jl          ! 0-based local indices
  integer :: lbr, lbc        ! local block indices
  integer :: off_i, off_j    ! offset inside local block
```

# Global to Local and Local to Global Index Mapping



```
integer :: first_b_r, first_b_c
integer :: ib, jb          ! absolute block indices

! Convert to 0-based local indices
il = i_local - 1
jl = j_local - 1
! Local block indices and offsets (0-based)
lbr = il / desc%MB
lbc = jl / desc%NB
off_i = mod(il, desc%MB)
off_j = mod(jl, desc%NB)
! First absolute block index assigned to (p_r,p_c)
first_b_r = mod(desc%RSRC + p_r, desc%P_r)
first_b_c = mod(desc%CSRC + p_c, desc%P_c)
```

# Global to Local and Local to Global Index Mapping



```
  ! Reconstruct absolute block indices
  ib = first_b_r + lbr * desc%P_r
  jb = first_b_c + lbc * desc%P_c
  ! Convert back to 1-based global indices
  i_global = ib * desc%MB + off_i + 1
  j_global = jb * desc%NB + off_j + 1
end subroutine local_to_global
```

Which can be used as needed in our distributed matrix operations.

# The prototype of the distributed GEMV subroutine



We have now done all the necessary preparations to implement the distributed GEMV.

The **prototype** of the distributed GEMV subroutine can be defined as follows:

```
subroutine gemv_distributed(mat, x, alpha, y, beta)
    class(distributed_matrix), intent(in) :: mat
    class(distributed_matrix), intent(in) :: x
    class(distributed_matrix), intent(inout) :: y
    real(wp), intent(in) :: alpha, beta
end subroutine gemv_distributed
```

Where:

- `mat` is the distributed matrix $A$,
- `x` and `y` are the distributed input and output vectors, respectively,
- `alpha` and `beta` are scalars for the operation $y = \alpha Ax + \beta y$.

# Temporary storage for the broadcasted vector



We need to create a **temporary storage** for the broadcasted vector segments, and to reduce the code redundancy, we can define a *local variable* for the descriptor of the matrix:

```
type(descriptor) :: desc
integer :: mloc, nloc, ierr, stat
real(wp), allocatable :: xbuf(:), y_partial(:), y_local(:)

desc = mat%desc
mloc = size(mat%local_data, 1)
nloc = size(mat%local_data, 2)
```

When do we need to allocate the temporary buffer `xbuf`?

# Temporary storage for the broadcasted vector



We need to allocate the temporary buffer `xbuf` on all processes:

```
allocate(xbuf(nloc), stat=stat)
if (stat /= 0) then
    print *, 'Error allocating xbuf'
    call MPI_Abort(mat%desc%comm, stat, ierr)
end if
```

However, only the processes in the **source column** of the vector need to copy their local data into this buffer:

```
if (desc%my_row == desc%RSRC) then
    xbuf = x%local_data(:,1)
end if
```

And now we are ready to perform the broadcast along the process rows.

# Executing the broadcast down the process columns



The call is

```
call MPI_Bcast(xbuf, nloc, MPI_REAL8, desc%RSRC, desc%col_comm, ierr)
```

where:

- `xbuf` is the temporary buffer for the broadcasted vector segment, it contains the local portion of the vector on the source process and will be filled on all other processes,
- `nloc` is the size of the local portion of the vector,
- `MPI_REAL8` is the MPI datatype for double precision real numbers,
- `desc%RSRC` is the row source process for the vector—the process which owns the local data to be broadcasted,
- `desc%col_comm` is the column communicator.
- `ierr` is the error code.

# Executing the local GEMV



We now have the necessary data to perform the local GEMV operation:

$$\mathbf{y}_{\text{partial}} = \alpha \mathbf{A}_{\text{local}} \mathbf{x}_{\text{buf}}$$

```
allocate(y_partial(mloc), stat=stat)
if (stat /= 0) then
    print *, 'Error allocating y_partial'
    call MPI_Abort(mat%desc%comm, stat, ierr)
end if
call dgemv('N', mloc, nloc, alpha, mat%local_data, mloc, xbuf, 1,
↪  0.0_wp, y_partial, 1)
```

Now on each process we have the local contribution to the output vector $\mathbf{y}$.

# Reducing the partial results along process columns



The last two step are

1. Reduce the partial results along the process columns,
2. Scale and update the output vector $\mathbf{y}$.

The **reduction** can be performed using:

```
if (desc%my_row == desc%RSRC) then
allocate(y_local(mloc), stat=stat)
  if (stat /= 0) then
      print *, 'Error allocating y_local'
      call MPI_Abort(mat%desc%comm, stat, ierr)
  end if
end if
call MPI_Reduce(y_partial, y_local, mloc, MPI_REAL8, MPI_SUM,
↪  mat%desc%RSRC, mat%desc%col_comm, ierr)
```

# Reducing the partial results along process columns



The last two step are

1. Reduce the partial results along the process columns,
2. Scale and update the output vector $\mathbf{y}$.

The **scaling and update** of the output vector can be done using:

```
if (desc%my_row == desc%RSRC) then
    ! Scale existing y by beta
    y%local_data(:,1) = beta * y%local_data(:,1)
    ! Add the reduced result
    y%local_data(:,1) = y%local_data(:,1) + y_local
    deallocate(y_local)
end if
```

# Let’s summarize the distributed GEMV implementation



1. Create communicators for process rows and columns.
2. In the distributed GEMV subroutine:
   - 2.1 Allocate a temporary buffer for the broadcasted vector segment.
   - 2.2 Copy local vector data into the buffer on the source process.
   - 2.3 Broadcast the vector segment along the process columns.
   - 2.4 Perform the local GEMV operation to compute the partial result.
   - 2.5 Reduce the partial results along the process columns.
   - 2.6 Scale and update the output vector on the source process.

As you can see, implementing distributed GEMV requires **careful management** of data distribution and communication patterns to ensure efficiency and correctness in a parallel computing environment.

# Running a test

## 2 Distributed matrices

We can now run a test of our distributed GEMV implementation by computing:

$$\mathbf{y} = 1.0\,A\mathbf{x} + 0.0\,\mathbf{y}, \text{ with } \{(A)_{i,j} = 1.0\}_{i,j=1}^{n}, \ \mathbf{x} = \mathbf{1}.$$

```
program test_distributed_gemv
   use mpi
   use distributed_gemv
   use iso_fortran_env, only: wp => real64
   implicit none
   integer :: ierr, rank, world_size, colrank, rowrank
   ! Matrix size
   integer, parameter :: M = 900, N = 900
   ! Matrix descriptor
   type(descriptor) :: desc_matrix, desc_vector
   type(distributed_matrix) :: mat, x, y
```

# Running a test

## 2 Distributed matrices

```
! Check variables
integer :: i
logical :: correct

call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, world_size, ierr)

! Create a descriptor for a square matrix distributed over a 3x3 process grid
call desc_matrix%init(MPI_COMM_WORLD, M, N, M/3, N/3, 3, 3, 0, 0)
! Initialize distributed matrix
call mat%init_matrix(desc_matrix, 1.0_wp)

! Create a distributed vector (which is an M x 1 distributed matrix)
call desc_vector%init(MPI_COMM_WORLD, M, 1, M/3, 1, 3, 1, 0, 0)
```

# Running a test

## 2 Distributed matrices

```
call x%init_matrix(desc_vector, 1.0_wp)
call y%init_matrix(desc_vector, 0.0_wp)

! Perform distributed matrix-vector multiplication
call mat%gemv_distributed(x, 1.0_wp, y, 0.0_wp)

! Check results: we are multiplying a matrix of ones by a vector of ones
! So the result should be a vector of size M with all entries equal to N
! we need to check only the local part of y only on the ranks that own data
correct = .true.
call MPI_Comm_rank(desc_matrix%col_comm, colrank, ierr)

if (colrank == 0) then
   do i = 1, size(y%local_data, 1)
      if (y%local_data(i,1) /= real(N, wp)) then
```

# Running a test

## 2 Distributed matrices

```
            correct = .false.
            exit
         end if
      end do
      if (correct) then
         print *, 'Test passed: Distributed GEMV result is correct on rank', rank,
         ↪  'colrank', colrank
      else
         print *, 'Test failed: Distributed GEMV result is incorrect on rank', rank,
         ↪  'colrank', colrank
      end if
   end if

   call MPI_Finalize(ierr)
end program test_distributed_gemv
```

# We compile and run the test



We can **compile** and **run** the test program using:

```
mpifort -o distributed_gemv distributed_gemv.f90 -lopenblas
```

And then **run** the program with 9 processes (3x3 grid):

```
mpirun -np 9 ./distributed_gemv
```

Which should output:

```
Test passed: Distributed GEMV result is correct on rank  2 colrank  0
Test passed: Distributed GEMV result is correct on rank  0 colrank  0
Test passed: Distributed GEMV result is correct on rank  1 colrank  0
```

# Run another test



We can run **another test** with a non unifor vector $x$:

```
if (colrank == 0) then
  y%local_data(:,1) = 1.0_wp
  ! Reinitialize x to have values ((rank-1)*(N/3)+1):(rank)*(N/3)
  do i = 1, size(x%local_data, 1)
    x%local_data(i,1) = real((rowrank+1)*(N/3)+i, wp)
    ! write(*,*) 'Rank', rank, 'x local(', i, ') =', x%local_data(i,1)
  end do
end if
```

And perform the distributed GEMV again:

```
call mat%gemv_distributed(x, 1.0_wp, y, 1.0_wp)
```

The expected result is now:

$$y_i = \sum_{j=1}^{N} A_{i,j}x_j + y_i = \sum_{j=1}^{N} 1.0 \cdot x_j + 1.0 = \sum_{j=1}^{N} x_j = \frac{N(N+1)}{2} + 1.0 = 405451.0$$

# Check the result of the second test



```
correct = .true.
if (colrank == 0) then
  do i = 1, size(y%local_data, 1)
    if (y%local_data(i,1) /= real(N*(N+1), wp)/2.0+1.0) then
      correct = .false.
      exit
    end if
  end do
  if (correct) then
      print *, 'Test passed: Distributed GEMV with beta=1.0 result is
       ↪  correct on rank', rank, 'colrank', colrank
  else
      print *, 'Test failed: Distributed GEMV with beta=1.0 result is
       ↪  incorrect on rank', rank, 'colrank', colrank
  end if
end if
```

# Run and check the second test



We can now **run** the modified test program again:

```
mpirun -np 9 ./distributed_gemv
```

Which should output:

```
Test passed: Distributed GEMV with beta=1.0 result is correct on rank 1
↪  colrank 0
Test passed: Distributed GEMV with beta=1.0 result is correct on rank 2
↪  colrank 0
Test passed: Distributed GEMV with beta=1.0 result is correct on rank 0
↪  colrank 0
```

# Summary, conclusions, and outlook



We have seen:

- How to implement a descriptor for distributed matrices,
- How to create communicators for process rows and columns,
- How to implement a distributed GEMV operation using MPI communication routines

**N**ext **s**teps are:

- Explore the ScaLAPACK library for distributed linear algebra,
- Implement distributed matrix-matrix multiplication (GEMM),
- Investigate the performance of distributed operations.

# High Performance Linear Algebra

Lecture 12: ScaLAPACK and Distributed BLAS level 3

Ph.D. program in High Performance Scientific Computing

**Fabio Durastante** **Pasqua D'Ambra** **Salvatore Filippone**

Thursday 29, 2026 — 16.00:18.00

Dipartimento
di Matematica
Università di Pisa

# Last time on High Performance Linear Algebra



We have

- We have implemented the basic routines for distributed matrices and vectors,
- We have implemented Level-1 BLAS operations for distributed matrices and vectors,
- We have implemented the GEMV operation for distributed matrices and vectors.

The plan for today is to:

- Introduce ScaLAPACK,
- Implement the Level-3 BLAS operation GEMM for distributed matrices.

# Table of Contents

2 PBLAS and ScaLAPACK


▶ PBLAS and ScaLAPACK

The BLACS library

The PBLAS operations

The distributed Matrix-Matrix multiplication

# The PBLAS and ScaLAPACK libraries



What we have done so far is to implement some of the routines of the PBLAS library, which is the distributed memory version of the BLAS library.

- ScaLAPACK is designed to mirror LAPACK, relying on a **P**arallel **BLAS** (PBLAS) interface that stays close to BLAS.
- Only one substantially new PBLAS routine is added: distributed matrix transposition.

Goal: Provide a distributed-memory standard like BLAS for shared memory.

# The PBLAS and ScaLAPACK libraries



What we have done so far is to implement some of the routines of the PBLAS library, which is the distributed memory version of the BLAS library.

- ScaLAPACK is designed to mirror LAPACK, relying on a **P**arallel **BLAS** (PBLAS) interface that stays close to BLAS.
- Only one substantially new PBLAS routine is added: distributed matrix transposition.

Goal: Provide a distributed-memory standard like BLAS for shared memory.

## 2D block-cyclic layout

- PBLAS matrices use a 2D block-cyclic distribution.
- Distribution parameters are stored in an array descriptor—instead than in the modern object-oriented style we have used.

# Distributed matrix descriptors



## Descriptor fields

1. Number of rows
2. Number of columns
3. Row block size (Section 2.5)
4. Column block size (Section 2.5)
5. Process row of first row
6. Process column of first column
7. BLACS context
8. Leading dimension of the local array

# BLACS contexts



A **BLACS context** defines a communication universe.

- Each distributed matrix is associated with a BLACS context.
- Different contexts allow independent communication universes.
- All descriptors in a PBLAS call must share the same context.
- This allows modularity in programs using multiple distributed matrices.

In our *modern implementation*, we can think of the BLACS context as an object storing the MPI communicator and the process grid information.

# BLAS vs PBLAS: DGEMM vs PDGEMM



Comparing two routines for matrix-matrix multiplication

**BLAS**

```
CALL DGEMM(TRANSA, TRANSB, M, N, K,
           ALPHA, A(IA, JA), LDA,
           B(IB, JB), LDB, BETA,
           C(IC, JC), LDC)
```

**PBLAS**

```
CALL PDGEMM(TRANSA, TRANSB, M, N, K,
            ALPHA, A, IA, JA, DESC_A,
            B, IB, JB, DESC_B, BETA,
            C, IC, JC, DESC_C)
```

- DGEMM uses A(IA, JA) to specify the submatrix.
- PDGEMM requires IA, JA, and DESC_A to locate the global submatrix.
- The same applies to B and C with DESC_B and DESC_C.

# We still need an MPI communicator!



In order to create a **BLACS context**, we first need to create an **MPI communicator**.

```
call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD,
↪  nprocs, ierr)
call MPI_Comm_rank(MPI_COMM_WORLD,
↪  myrank, ierr)
```

The generic structure of a BLACS application is as follows:

1. Initialize MPI
2. Initialize BLACS
3. Create a process grid
4. Query process coordinates
5. Perform communication
6. Destroy grid and exit BLACS
7. Finalize MPI

# We still need an MPI communicator!



```
integer :: ictxt, nprow, npcol
call blacs_get(-1, 0, ictxt)
call blacs_gridinit(ictxt, 'R',
↪  nprow, npcol)
```

Where

- `ictxt` is the BLACS context identifier,
- `blacs_get` initializes the BLACS system,
- `blacs_gridinit` creates a process grid with `nprow` rows and `npcol` columns.

The generic structure of a BLACS application is as follows:

1. Initialize MPI
2. Initialize BLACS
3. Create a process grid
4. Query process coordinates
5. Perform communication
6. Destroy grid and exit BLACS
7. Finalize MPI

# We still need an MPI communicator!



We can then query the process coordinates in the grid:

```
integer :: myrow, mycol
call blacs_gridinfo(ictxt, nprow,
↪  npcol, myrow, mycol)
```

Where

- `myrow` is the row coordinate of the process,
- `mycol` is the column coordinate of the process.

The generic structure of a BLACS application is as follows:

1. Initialize MPI
2. Initialize BLACS
3. Create a process grid
4. Query process coordinates
5. Perform communication
6. Destroy grid and exit BLACS
7. Finalize MPI

# Row major vs Column major



The two common ways to map a 1D array to a 2D array are:

```
! Row major mapping
index = (i-1)*ncols + (j-1) + 1
```

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |

```
! Column major mapping
index = (j-1)*nrows + (i-1) + 1
```

| | | | |
|---|---|---|---|
| 0 | 3 | 6 | 9 |
| 1 | 4 | 7 | 10 |
| 2 | 5 | 8 | 11 |

Row major:

```
call blacs_gridinit(ictxt, 'R', nprow, npcol)
```

# Row major vs Column major



The two common ways to map a 1D array to a 2D array are:

```
! Row major mapping
index = (i-1)*ncols + (j-1) + 1
```

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |

```
! Column major mapping
index = (j-1)*nrows + (i-1) + 1
```

| | | | |
|---|---|---|---|
| 0 | 3 | 6 | 9 |
| 1 | 4 | 7 | 10 |
| 2 | 5 | 8 | 11 |

Column major:

```
call blacs_gridinit(ictxt, 'C', nprow, npcol)
```

# A complete example of initialization



```
integer :: ierr, nprocs, myrank
integer :: ctxt, nrows, ncols, myrow, mycol
integer :: info
! Initialize MPI
call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD, nprocs, ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, myrank, ierr)
! Initialize BLACS
call blacs_get(-1, 0, ctxt)
! Create a process grid
nrows = int(sqrt(real(nprocs)))
ncols = nprocs / nrows
call blacs_gridinit(ctxt, 'C', nrows, ncols)
! Query process coordinates
call blacs_gridinfo(ctxt, nrows, ncols, myrow, mycol)
```

# If we compile and execute this code



Compiling with

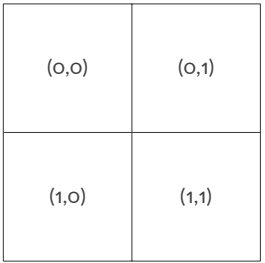
```
mpifort -o blacs_init blacs_init.f90 -lscalapack
```

And executing with

```
mpirun -np 4 ./blacs_init
```

We could obtain the following output:

```
BLACS grid: 2 x 2
Processor 3 is at (1,1)
Processor 0 is at (0,0)
Processor 1 is at (1,0)
Processor 2 is at (0,1)
```



# Where to find BLACS, PBLAS, and ScaLAPACK



- BLACS, PBLAS, and ScaLAPACK are usually provided as part of high-performance linear algebra libraries such as
  - Intel MKL,
  - AMD ACML,
  - Netlib ScaLAPACK.
- The latter can also be built from source code available from the Netlib repository:
  - http://www.netlib.org/scalapack/

They can be installed via Spack as well:

```
spack install netlib-scalapack
```

or

```
spack install intel-oneapi-mkl
```

# PBLAS operations



The **PBLAS library** provides distributed memory implementations of the Level-1, Level-2, and Level-3 BLAS operations.

- Level-1 PBLAS operations include vector-vector operations such as **P?AXPY**.
- Level-2 PBLAS operations include matrix-vector operations such as **P?GEMV**.
- Level-3 PBLAS operations include matrix-matrix operations such as **P?GEMM**.

We can now try using the P?GEMV operation, and try to measure its performance.

# The P?GEMV operation



The **P?GEMV** operation computes the matrix-vector product

$$y \leftarrow \alpha Ax + \beta y,$$

where $A$ is a distributed matrix, and $x$ and $y$ are distributed vectors.

- The operation is called via the `PDGEMV` routine.
- The routine requires the descriptors of the distributed matrix and vectors.

The routine signature is as follows:

```
CALL PDGEMV(TRANS, M, N, ALPHA, A, IA, JA, DESC_A, X, IX, 1, DESC_X,
            BETA, Y, IY, 1, DESC_Y)
```

# The P?GEMV operation: parameters



```
CALL PDGEMV(TRANS, M, N, ALPHA, A, IA, JA, DESC_A, X, IX, 1, DESC_X,
            BETA, Y, IY, 1, DESC_Y)
```

- `TRANS` specifies whether to use $A$ or $A^T$,
- `M` and `N` are the number of rows and columns of $A$,
- `ALPHA` and `BETA` are scalars,
- `A` is the local array containing the local pieces of $A$,
- `IA` and `JA` are the row and column indices of the first element of the submatrix of $A$,
- `DESC_A` is the descriptor of $A$,
- `X` is the local array containing the local pieces of $x$,
- `IX` is the index of the first element of the subvector of $x$,
- `DESC_X` is the descriptor of $x$,
- `Y` is the local array containing the local pieces of $y$,
- `IY` is the index of the first element of the subvector of $y$,
- `DESC_Y` is the descriptor of $y$.

# Testing PDGEMV



To test the `PDGEMV` routine, we can write a simple Fortran program that:

1. Build the test program and link against ScaLAPACK.
2. Run with a square number of MPI ranks.
3. Choose $m$, $n$, and block size $nb$ via command-line arguments.

Then we can call `PDGEMV` to perform the matrix-vector multiplication.

The code to compile and run the test program is something on the lines of:

```
mpifort -O3 -o test_pdegemv test_pdegemv.f90 -lscalapack
mpirun -np 4 ./test_pdegemv 4000 4000 128
```

# Reading command-line arguments



We can read command-line arguments in Fortran as follows:

```
m = 4000
n = 4000
nb = 128
nreps = 10

arg_count = command_argument_count()
if (arg_count >= 1) then
  call get_command_argument(1, arg)
  read(arg, *) m
end if
if (arg_count >= 2) then
  call get_command_argument(2, arg)
  read(arg, *) n
end if
if (arg_count >= 3) then
  call get_command_argument(3, arg)
  read(arg, *) nb
end if
if (arg_count >= 4) then
  call get_command_argument(4, arg)
  read(arg, *) nreps
end if
```

Which allows us to set $m$, $n$, $nb$, and the number of repetitions $nreps$ for the matrix-vector multiplication.

# Initializing distributed environment



We use the init code shown before to initialize MPI and BLACS.

```
call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, world_rank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, world_size, ierr)
nprocs_per_dim = int(sqrt(dble(world_size)))
if (nprocs_per_dim * nprocs_per_dim /= world_size) then
  if (world_rank == 0) then
    print *, 'Error: number of processes must be a perfect square.'
  end if
  call MPI_Finalize(ierr)
  stop 1
end if
```

and check that the number of processes is a **perfect square**.

# Initializing distributed environment



Then the BLACS initialization follows:

```
call blacs_get(-1, 0, ictxt)
nprow = nprocs_per_dim
npcol = nprocs_per_dim
call blacs_gridinit(ictxt, 'R', nprow, npcol)
call blacs_gridinfo(ictxt, nprow, npcol, myrow, mycol)

if (myrow == -1 .or. mycol == -1) then
  call blacs_exit(1)
  call MPI_Finalize(ierr)
  stop 1
end if
```

# Creating distributed matrices and vectors



We can now create the distributed matrix $A$ and vectors $\mathbf{x}$ and $\mathbf{y}$:

```
mloc = numroc(m, nb, myrow, 0, nprow)
nloc = numroc(n, nb, mycol, 0, npcol)
lldA = max(1, mloc)

xloc = numroc(n, nb, myrow, 0, nprow)
yloc = numroc(m, nb, myrow, 0, nprow)
lldX = max(1, xloc)
lldY = max(1, yloc)
```

Where we use `NUMROC` to compute the local sizes of the distributed arrays.

# The NUMROC utility



The NUMROC utility computes the number of rows or columns of a distributed matrix or vector owned by a given process.

```
integer function numroc(n, nb, iproc, isrcproc, nprocs)
```

Where

- `n` is the global number of rows or columns,
- `nb` is the block size,
- `iproc` is the coordinate of the process in the grid,
- `isrcproc` is the coordinate of the process owning the first row or column,
- `nprocs` is the number of processes in the grid dimension.

# Creating the descriptors



The descriptors for the distributed matrix and vectors for ScaLAPACK are variables of type **integer**, defined as arrays of size 9.

```
integer :: descA(9), descX(9), descY(9)
```

We can initialize them as follows:

```
call descinit(descA, m, n, nb, nb, 0, 0, ictxt, lldA, info)
call descinit(descX, n, 1, nb, 1, 0, 0, ictxt, lldX, info)
call descinit(descY, m, 1, nb, 1, 0, 0, ictxt, lldY, info)
```

We should always check the value of `info` after each call to `descinit`, e.g.,

```
if (info /= 0) then
  if (world_rank == 0) print *, 'descinit error: ', info
  call MPI_Abort(MPI_COMM_WORLD, info, ierr)
end if
```

# Allocate and fill the local arrays



Next, we allocate and fill the local arrays:

```
allocate(A(lldA, max(1, nloc)))
allocate(X(lldX, max(1, numroc(1, 1, mycol, 0, npcol))))
allocate(Y(lldY, max(1, numroc(1, 1, mycol, 0, npcol))))
```

Where we use again the `numroc` utility to compute the local sizes of the vectors.

We **fill the matrix** $A$ with:

```
if (mloc > 0 .and. nloc > 0) then
  do j = 1, nloc
    col_global = indxl2g(j, nb, mycol, 0, npcol)
    do i = 1, mloc
      row_global = indxl2g(i, nb, myrow, 0, nprow)
      A(i, j) = dble(row_global + col_global) / dble(m + n)
    end do
  end do
end if
```

# Allocate and fill the local arrays



Next, we allocate and fill the local arrays:

```
allocate(A(lldA, max(1, nloc)))
allocate(X(lldX, max(1, numroc(1, 1, mycol, 0, npcol))))
allocate(Y(lldY, max(1, numroc(1, 1, mycol, 0, npcol))))
```

Where we use again the `numroc` utility to compute the local sizes of the vectors.
We **fill the vectors $\mathbf{x}$**, and $\mathbf{y}$ with:

```
if (xloc > 0) then
  do i = 1, xloc
    row_global = indxl2g(i, nb, myrow, 0, nprow)
    X(i, 1) = 1.0d0 + dble(row_global) / dble(n)
  end do
end if
if (yloc > 0) Y(1:yloc, 1) = 0.0d0
```

# Timing and synchronization



- Use `MPI_Barrier` to synchronize before and after `PDGEMV`.
- Measure elapsed time with `MPI_Wtime`.

```
call MPI_Barrier(MPI_COMM_WORLD, ierr)
t0 = MPI_Wtime()
do i = 1, nreps
  call pdgemv('N', m, n, alpha, A, 1, 1, descA, X, 1, 1, descX, 1, beta, Y,
  ↪  1, 1, descY, 1)
end do
call MPI_Barrier(MPI_COMM_WORLD, ierr)
t1 = MPI_Wtime()
elapsed_time = (t1 - t0)
```

As usual, this the `elapsed_time` contains the total time for *nreps* repetitions of the matrix-vector multiplication on each process.

# Computing the performance

## 2 PBLAS and ScaLAPACK

We take the **worst-case time** across all processes:

```
call MPI_Reduce(elapsed_time, max_elapsed, 1, MPI_DOUBLE_PRECISION,
↪  MPI_MAX, 0, MPI_COMM_WORLD, ierr)
```

Then, on rank $0$, we can compute the measurements:

```
if (myrow == 0 .and. mycol == 0) then
  gflops = (2.0d0 * dble(m) * dble(n) * dble(nreps)) / max_elapsed / 1.0d9
  avg_time = max_elapsed / dble(nreps)
  avg_gflops = (2.0d0 * dble(m) * dble(n)) / avg_time / 1.0d9
  print *, 'PDGEMV m=', m, ' n=', n, ' nb=', nb, ' procs=', world_size, '
  ↪  reps=', nreps
  print *, 'Total time (s)=', max_elapsed, ' Total GFLOPS=', gflops
  print *, 'Avg time (s)=', avg_time, ' Avg GFLOPS=', avg_gflops
end if
```

# GFLOPS calculation for PDGEMV



We can compute the number of floating-point operations for the matrix-vector multiplication as follows:

$$\text{FLOPS} = 2 \cdot m \cdot n$$

Thus, the performance in GFLOPS can be computed as:

```
gflops = (2.0d0 * dble(m) * dble(n) * dble(nreps)) / max_elapsed / 1.0d9
```

Where `nreps` is the number of repetitions of the multiplication.

Similarly, the worst-case performance per repetition is:

```
avg_gflops = (2.0d0 * dble(m) * dble(n)) / avg_time / 1.0d9
```

# Finalizing the distributed environment



Finally, we need to finalize BLACS and MPI:

```
if (allocated(A)) deallocate(A)
if (allocated(X)) deallocate(X)
if (allocated(Y)) deallocate(Y)
call blacs_gridexit(ictxt)
call blacs_exit(0)
```

We don't need to call `MPI_Finalize` explicitly, as it is called inside the `blacs_gridexit`.

We are finally done, and we can compile and run our test program. As for the case in the other lectures, we plan on running it on the Amelia cluster at IAC-CNR. Hence, we use the Intel Oneapi compilers, and link against Intel MKL—which provides BLACS, PBLAS, and ScaLAPACK.

# Running script



The script to run the test program on Amelia is written as follows:

```
#!/usr/bin/env bash
#SBATCH --nodes=@NODES@
#SBATCH --ntasks=@TASKS@
#SBATCH --cpus-per-task=@THREADS@
#SBATCH --partition=prod-gn
#SBATCH --time=01:00:00
#SBATCH --mem=950Gb
#SBATCH --job-name=pdegemv_weak

export OMP_NUM_THREADS=1
export MKL_NUM_THREADS=1
cd /ifs/hpc/home/fdurastante/scalapacktest/launchscripts/
mpirun -np @TASKS@ ./../build/test_pdegemv @N@ @N@ @NB@ 2>&1 >
↪  ../logfiles/pdegenmv/log_t@TASKS@.log
```

# Running script explanation



The script uses SLURM directives to request resource and has a number of placeholders:

- `@NODES@`: number of nodes to use,
- `@TASKS@`: number of MPI tasks to use,
- `@THREADS@`: number of threads per task,
- `@N@`: size of the matrix and vectors,
- `@NB@`: block size.

We use an auxiliary script to replace the placeholders and submit the job to SLURM:

```
./genscript.sh launch_pdegenmv.sh
```

Where `genscript.sh` replaces the placeholders and produces the launch files.

# The `genscript.sh` auxiliary script



The auxiliary script `genscript.sh` is as follows:

```
#!/bin/sh
TEMPLATE="$1"
TASKS_PER_NODE=64
THREADS_PER_TASK=1
N_LOCAL=4000
NB=128
# Perfect-square task counts up to 20 nodes * 64 tasks/node = 1280 tasks
TASKS_LIST="64 144 256 400 576 900 1024 1156"
for TASKS in $TASKS_LIST; do
    # Compute number of nodes (ceiling division)
    NODES=$(( (TASKS + TASKS_PER_NODE - 1) / TASKS_PER_NODE ))
    PROCS_PER_DIM=$(LC_NUMERIC=C echo "scale=0; sqrt($TASKS)" | bc -l)
```

# The `genscript.sh` auxiliary script



```
    N=$((N_LOCAL * PROCS_PER_DIM))
    OUTFILE="outscript_t${TASKS}.sh"
    sed \
        -e "s/@NODES@/${NODES}/g" \
        -e "s/@TASKS@/${TASKS}/g" \
        -e "s/@THREADS@/${THREADS_PER_TASK}/g" \
        -e "s/@N@/${N}/g" \
        -e "s/@NB@/${NB}/g" \
        "$TEMPLATE" > "$OUTFILE"
    chmod +x "$OUTFILE"
    echo "Generated $OUTFILE (tasks=$TASKS, nodes=$NODES,
    ↪  threads=$THREADS_PER_TASK, N=$N, NB=$NB)"
done
```

# The `genscript.sh` auxiliary script



The script iterates over a list of perfect-square task counts

$$n_p = 64, 144, 256, 400, 576, 900, 1024, 1156,$$

computes the number of nodes required, the problem size $N$, and replaces the placeholders in the template script using `sed`.

The `sed` command replaces each placeholder with the corresponding value, its general form being:

```
sed -e "s/@PLACEHOLDER@/value/g" input_template > output_script
```

Where `@PLACEHOLDER@` is replaced with `value` in the `input_template`, and the result is saved in `output_script`.

# The `genscript.sh` auxiliary script



We run the `genscript.sh` script as follows:

```
chmod +x genscript.sh
./genscript.sh launch_pdegemv.sh
```

Which produces the following output:

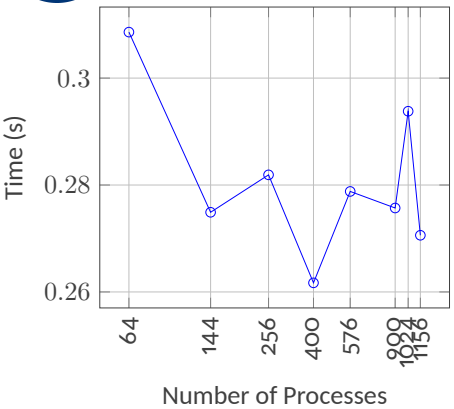
```
Generated outscript_t64.sh (tasks=64, nodes=1, threads=1, N=32000, NB=128)
Generated outscript_t144.sh (tasks=144, nodes=3, threads=1, N=48000, NB=128)
Generated outscript_t256.sh (tasks=256, nodes=4, threads=1, N=64000, NB=128)
Generated outscript_t400.sh (tasks=400, nodes=7, threads=1, N=80000, NB=128)
Generated outscript_t576.sh (tasks=576, nodes=9, threads=1, N=96000, NB=128)
Generated outscript_t900.sh (tasks=900, nodes=15, threads=1, N=120000, NB=128)
Generated outscript_t1024.sh (tasks=1024, nodes=16, threads=1, N=128000, NB=128)
Generated outscript_t1156.sh (tasks=1156, nodes=19, threads=1, N=136000, NB=128)
```

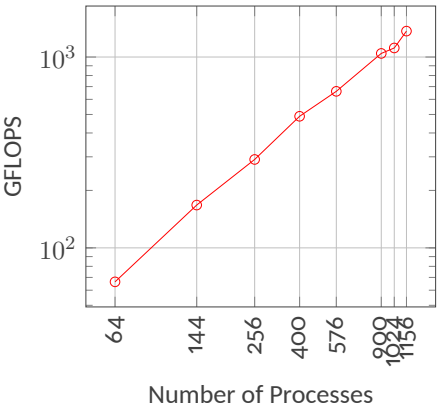
We can then submit each generated script to SLURM with:

```
sbatch outscript_t??.sh
```

# Analyzing the results

## 2 PBLAS and ScaLAPACK




We can see that the time remains roughly constant as we increase the number of processes, while the performance increases accordingly $N =$, $NB =$.

# The distributed Matrix-Matrix multiplication



- We will consider the formation of the matrix products

$$C = \alpha AB + \beta C$$
$$C = \alpha AB^T + \beta C$$
$$C = \alpha A^T B + \beta C$$
$$C = \alpha A^T B^T + \beta C$$

- These are the special cases implemented in the sequential BLAS `GEMM`.
- Assume each matrix $X$ has dimensions $m^X \times n^X$, with $X \in \{A, B, C\}$.
- Dimensions must be compatible; we take $C \in \mathbb{R}^{m \times n}$ and the inner dimension $k$.

# Back on the data distribution



- We consider 2D data decompositions;
- The 1D case is obtained by setting one grid dimension to 1.
- Given $X \in \{A, B, C\}^{m \times n}$ on an $r \times c$ process grid, we partition:

$$X = \begin{pmatrix} X_{00} & \cdots & X_{0(c-1)} \\ \vdots & & \vdots \\ X_{(r-1)0} & \cdots & X_{(r-1)(c-1)} \end{pmatrix}$$

- Submatrix $X_{ij}$ is assigned to process $P_{ij}$.
- $X_{ij}$ has size $m_i^X \times n_j^X$, with $\sum_i m_i^X = m$ and $\sum_j n_j^X = n$.
- Each algorithm variant enforces row/column compatibility in these dimensions: For this operation to be well-defined, we require $m^A = m$, $n^A = m^B = k$, and $n^B = n$.

# Forming $C = \alpha AB + \beta C$



For simplicity, we take $\alpha = 1$ and $\beta = 0$ in our description.
If $a_{ij}$, $b_{ij}$, and $c_{ij}$ denote the $(i,j)$ element of the matrices, respectively, then the elements of $C$ are given by

$$c_{ij} = \sum_{l=1}^{k} a_{il} b_{lj}.$$

Notice that:

- rows of $C$ are computed from rows of $A$, and columns of $C$
- We hence restrict our data decomposition so that rows of $A$ and $C$ are assigned to the same row of nodes and columns of $B$ and $C$ are assigned to the same column of nodes.
- Hence, $m_i^C = m_i^A$ and $n_j^C = n_j^B$.

# Basic parallel algorithm (as we have seen for the OpenMP case)



- Compute $C_{ij}$ as a sequence of rank-one updates.
- Assign block row $\tilde{A}_i$ to process row $i$.
- Assign block column $\tilde{B}^j$ to process column $j$.

$$\tilde{A}_i = \begin{pmatrix} a_i^{(0)} & a_i^{(1)} & \cdots & a_i^{(k-1)} \end{pmatrix}, \qquad \tilde{B}^j = \begin{pmatrix} b_j^{(0)T} \\ b_j^{(1)T} \\ \vdots \\ b_j^{(k-1)T} \end{pmatrix}$$

$$C_{ij} = \sum_{t=0}^{k-1} a_i^{(t)} b_j^{(t)T}$$

# Rank-one update view



Each step $t$:

- broadcasts $a_i^{(t)}$ along process row $i$.
- broadcasts $b_j^{(t)}$ along process column $j$.

Perform

- Local update on $P_{ij}$: $C_{ij} \leftarrow C_{ij} + a_i^{(t)} b_j^{(t)T}$.

$$C_{ij}^{(t+1)} = C_{ij}^{(t)} + a_i^{(t)} b_j^{(t)T}$$

1: $C_{ij} \leftarrow 0$
2: **for** $\ell = 0, \ldots, k-1$ **do**
3: broadcast $a_i^{(\ell)}$ within my row
4: broadcast $b_j^{(\ell)}$ within my column
5: $C_{ij} \leftarrow C_{ij} + a_i^{(\ell)} b_j^{(\ell)T}$
6: **end for**

Now we can analyze the cost of this algorithm, both in **terms of computation** and **communication**.

# Algorithm cost: minimum spanning tree broadcast



To analyze the cost of this basic algorithm, we make some **simplifying assumptions**:

$$m_i^C = m_i^A = m/r, \qquad n_j^C = n_j^B = n/c,$$
$$n_i^A = k/c, \qquad m_j^B = k/r.$$

Since relatively little data is involved during each broadcast we assume a **minimum-spanning-tree broadcast**.

# Algorithm cost: minimum spanning tree broadcast



To analyze the cost of this basic algorithm, we make some **simplifying assumptions**:

$$m_i^C = m_i^A = m/r, \qquad n_j^C = n_j^B = n/c,$$
$$n_i^A = k/c, \qquad m_j^B = k/r.$$

Since relatively little data is involved during each broadcast we assume a **minimum-spanning-tree broadcast**.

## Minimum-Spanning-Tree Broadcast

A minimum-spanning-tree (MST) broadcast minimizes latency by organizing the broadcast in $\log(p)$ stages, where $p$ is the number of processes.

- Stage 0: Process 0 sends to process 1
- Stage 1: Processes 0,1 send to processes 2,3
- Stage 2: Processes 0,1,2,3 send to processes 4,5,6,7

 And so on...

# Algorithm cost: minimum spanning tree broadcast



To analyze the cost of this basic algorithm, we make some **simplifying assumptions**:

$$m_i^C = m_i^A = m/r, \qquad n_j^C = n_j^B = n/c,$$
$$n_i^A = k/c, \qquad m_j^B = k/r.$$

Since relatively little data is involved during each broadcast we assume a **minimum-spanning-tree broadcast**.

## Minimum-Spanning-Tree Broadcast

**Cost model** for MST broadcast of data size $S$ to $p$ processes:

$$T_{\text{bcast}} = \lceil \log_2(p) \rceil \cdot (\alpha + S \cdot \beta)$$

Where:

- $\alpha$ and $\beta$ are the latency and the inverse bandwidth (per unit data),
- Each stage involves one message transmission for $= \lceil \log_2(p) \rceil$ total stages.

# Cost model (detailed)



The cost of the algorithm (per panel) is therefore

$$k\left[\frac{2mn}{p}\gamma + \lceil\log(c)\rceil\left(\alpha + \frac{m}{r}\beta\right) + \lceil\log(r)\rceil\left(\alpha + \frac{n}{c}\beta\right)\right].$$

The three terms inside the square brackets correspond to

- $2mn/p\,\gamma$: local rank-one update,
- $\lceil\log(c)\rceil\,(\alpha + m/r\beta)$: row broadcast of $A$,
- $\lceil\log(r)\rceil\,(\alpha + n/c\beta)$: column broadcast of $B$.

Hence the total time is

$$T(m,n,k,p) = \frac{2mnk}{p}\gamma + k\big(\lceil\log(c)\rceil + \lceil\log(r)\rceil\big)\alpha + \lceil\log(c)\rceil\frac{mk}{r}\beta + \lceil\log(r)\rceil\frac{nk}{c}\beta$$

# Scalability analysis

## 2 PBLAS and ScaLAPACK

The cost $T(m,n,k,p)$ compares to the sequential time $2mnk\,\gamma$.

To **study scalability**, set $m = n = k$, $r = c = \sqrt{p}$ (assume $p$ power of two).
From $T(m,n,k,p)$ the **estimated speedup** is

$$S(n,p) = \frac{2n^3\gamma}{\frac{2n^3}{p}\gamma + n\log(p)\alpha + \log(p)\frac{n^2}{\sqrt{p}}\beta} = \frac{p}{1 + \frac{p\log(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{\sqrt{p}\log(p)}{2n}\frac{\beta}{\gamma}}.$$

The corresponding **parallel efficiency** is

$$E(n,p) = \frac{S(n,p)}{p} = \frac{1}{1 + \frac{p\log(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{\sqrt{p}\log(p)}{2n}\frac{\beta}{\gamma}} = \frac{1}{1 + O\left(\frac{p\log p}{n^2}\right) + O\left(\frac{\sqrt{p}\log p}{n}\right)}.$$

# Scalability insight



- Speedup improves with $\sqrt{p}$ process grids.
- Efficiency degrades slowly with $\log(p)$ broadcast costs.
- Increasing $n$ with $\sqrt{p}$ keeps memory per process fixed.

$$E(n,p) = \frac{1}{1 + O\left(\frac{p\log(p)}{n^2}\right) + O\left(\frac{\sqrt{p}\log(p)}{n}\right)}$$

## Scalability insight

Ignoring the $\log(p)$ term, which grows *very* slowly when $p$ is reasonably large, we notice the following: If we increase $p$ and we wish to maintain efficiency, we must increase $n$ with $\sqrt{p}$. Since **memory requirements grow with** $n^2$, and physical memory grows linearly with $p$ as nodes are added.

# Scalability insight



- Speedup improves with $\sqrt{p}$ process grids.
- Efficiency degrades slowly with $\log(p)$ broadcast costs.
- Increasing $n$ with $\sqrt{p}$ keeps memory per process fixed.

$$E(n,p) = \frac{1}{1 + O\left(\frac{p\log(p)}{n^2}\right) + O\left(\frac{\sqrt{p}\log(p)}{n}\right)}$$

**Scalability insight**

We conclude that **the method is scalable** in the following sense:
"If we maintain memory use *per node*, this algorithm will maintain efficiency, if $\log(p)$ is treated as a constant."

# Moving to the pipelined algorithm



We will present the benefits of **pipelining computations and communications**.

Let us consider implementing the **broadcast** as passing of a message around the logical ring that forms the row or column.

```
subroutine RING_Bcast(data, count, type, root, comm)
 implicit none
 integer, intent(in) :: count, type, root, comm
 real(kind=real64), intent(inout) :: data(*)
 ! Local variables
 integer :: me, np, next, prev, ierr
 call MPI_Comm_rank(comm, me, ierr)
 call MPI_Comm_size(comm, np, ierr)
```

# Moving to the pipelined algorithm



```
 if ( me /= root ) then
   prev = mod(me - 1 + np, np)
   call MPI_Recv(data, count, type, prev, MPI_ANY_TAG, comm,
   ↪  MPI_STATUS_IGNORE, ierr)
 end if
 if ( mod(me + 1, np) /= root ) then
   next = mod(me + 1, np)
   call MPI_Send(data, count, type, next, 0, comm, ierr)
 end if
end subroutine RING_Bcast
```

# Moving to the pipelined algorithm



In this case, the time complexity becomes:

- The time required for both the first column of $A_{r-1}$ and the first row of $B^{c-1}$ to reach process $(r-1, c-1)$:

$$(c-1)\left(\alpha + \frac{m}{r}\beta\right) + (r-1)\left(\alpha + \frac{n}{c}\beta\right)$$

# Moving to the pipelined algorithm



In this case, the time complexity becomes:

- The time required for both the first column of $A_{r-1}$ and the first row of $B^{c-1}$ to reach process $(r-1, c-1)$
- The time for performing the local update and passing the messages along the pipe

$$+k\left(\frac{2mn}{p}\gamma + \alpha + \frac{m}{r}\beta + \alpha + \frac{n}{c}\beta\right)$$

# Moving to the pipelined algorithm



In this case, the time complexity becomes:

- The time required for both the first column of $A_{r-1}$ and the first row of $B^{c-1}$ to reach process $(r-1, c-1)$
- The time for performing the local update and passing the messages along the pipe
- The time for the final messages (initiated at process $(r-1, c-1)$) to reach the end of the pipe:

$$+(c-2)\left(\alpha + \frac{m}{r}\beta\right) + (r-2)\left(\alpha + \frac{n}{c}\beta\right)$$

# Moving to the pipelined algorithm



In this case, the time complexity becomes:

- The time required for both the first column of $A_{r-1}$ and the first row of $B^{c-1}$ to reach process $(r-1, c-1)$
- The time for performing the local update and passing the messages along the pipe
- The time for the final messages (initiated at process $(r-1, c-1)$) to reach the end of the pipe
- the time for the final update at the node at the end of the pipe (process $(r-1, c-2)$ or $(r-2, c-1)$):

$$+\frac{2mn}{p}\gamma$$

# Moving to the pipelined algorithm



In this case, the time complexity becomes:

- **+** The time required for both the first column of $A_{r-1}$ and the first row of $B^{c-1}$ to reach process $(r-1, c-1)$
- **+** The time for performing the local update and passing the messages along the pipe
- **+** The time for the final messages (initiated at process $(r-1, c-1)$) to reach the end of the pipe
- **+** the time for the final update at the node at the end of the pipe (process $(r-1, c-2)$ or $(r-2, c-1)$)
- **=** Summing all contributions, we obtain:

$$= \frac{2mn(k+1)}{p}\gamma + (k+2c-3)\left(\alpha + \frac{m}{r}\beta\right) + (k+2r-3)\left(\alpha + \frac{n}{c}\beta\right)$$

# Comparing the two approaches



For the **minimum-spanning-tree broadcast** we had a cost of

$$T(m,n,k,p) = \frac{2mnk}{p}\gamma + k\big(\lceil\log(c)\rceil + \lceil\log(r)\rceil\big)\alpha + \lceil\log(c)\rceil\frac{mk}{r}\beta + \lceil\log(r)\rceil\frac{nk}{c}\beta,$$

For the **pipelined ring broadcast** we have a cost of

$$T(m,n,k,p) = \frac{2mn(k+1)}{p}\gamma + (k+2c-3)\left(\alpha + \frac{m}{r}\beta\right) + (k+2r-3)\left(\alpha + \frac{n}{c}\beta\right),$$

Notice that for large $k$, the "log" factors we had in the tree approach are removed.

# Scalability of the pipelined approach



To **establish the scalability**, we again analyse the case where $m = n = k$ and $r = c = \sqrt{p}$.

This changes the complexity to approximately

$$\frac{2n^3}{p}\gamma + 2(n + 2\sqrt{p} - 3)\left(\alpha + \frac{n}{\sqrt{p}}\beta\right).$$

The **speedup** is

$$S(n,p) = \frac{2n^3\gamma}{\frac{2n^3}{p}\gamma + 2(n + 2\sqrt{p} - 3)\left(\alpha + \frac{n}{\sqrt{p}}\beta\right)} \approx \frac{p}{1 + \frac{p}{n^2}\alpha + \frac{\sqrt{p}}{n}\beta}$$

The corresponding **efficiency** is

$$E(n,p) = \frac{S(n,p)}{p} = \frac{1}{1 + O\left(\frac{p}{n^2}\right) + O\left(\frac{\sqrt{p}}{n}\right)}$$

# Scalability of the pipelined approach



To **establish the scalability**, we again analyse the case where $m = n = k$ and $r = c = \sqrt{p}$.

The **speedup** is

$$S(n,p) = \frac{2n^3\gamma}{\frac{2n^3}{p}\gamma + 2(n + 2\sqrt{p} - 3)\left(\alpha + \frac{n}{\sqrt{p}}\beta\right)} \approx \frac{p}{1 + \frac{p}{n^2}\alpha + \frac{\sqrt{p}}{n}\beta}$$

The corresponding **efficiency** is

$$E(n,p) = \frac{S(n,p)}{p} = \frac{1}{1 + O\left(\frac{p}{n^2}\right) + O\left(\frac{\sqrt{p}}{n}\right)}$$

The $\log(p)$ term has disappeared and the method is again scalable in the sense that if we **maintain memory use per node**, this algorithm will maintain efficiency.

# Blocking to further improve performance



We can reformulate the algorithm to use matrix-matrix multiplcations, instead of **rank-one updates**.

## Memory acces bandwidth

Recall that matrix-matrix multiplications have a much higher **computational intensity** (flops per memory access), hence they better exploit memory hierarchies and achieve higher performance on modern architectures.

# Blocking to further improve performance



We can reformulate the algorithm to use matrix-matrix multiplcations, instead of **rank-one updates**.

## Memory acces bandwidth

Recall that matrix-matrix multiplications have a much higher **computational intensity** (flops per memory access), hence they better exploit memory hierarchies and achieve higher performance on modern architectures.

# Blocking to further improve performance



We can reformulate the algorithm to use matrix-matrix multiplcations, instead of **rank-one updates**.

## Memory acces bandwidth

Recall that matrix-matrix multiplications have a much higher **computational intensity** (flops per memory access), hence they better exploit memory hierarchies and achieve higher performance on modern architectures.

- From the previous explanation, we change that each panel of $A$ and $B$ consist now of a block of columns/rows.

# Blocking to further improve performance



We can reformulate the algorithm to use matrix-matrix multiplcations, instead of **rank-one updates**.

## Memory acces bandwidth

Recall that matrix-matrix multiplications have a much higher **computational intensity** (flops per memory access), hence they better exploit memory hierarchies and achieve higher performance on modern architectures.

- From the previous explanation, we change that each panel of $A$ and $B$ consist now of a block of columns/rows.
- An **additional advantage of blocking** is that it reduces the number of messages incurred $\Rightarrow$ lower latency cost/communication overhead.

# Some practical considerations

## 2 PBLAS and ScaLAPACK

- Choose $nb$ to balance computation and communication.
- Square-ish grids minimize communication volume.
- Use optimized local BLAS for the inner GEMM.
- Synchronize only for timing, not for correctness.
- The routine is called via `PDGEMM`.
- $A$, $B$, and $C$ are distributed with 2D block-cyclic layout.

```
CALL PDGEMM(TRANSA, TRANSB, M, N, K, ALPHA,
            A, IA, JA, DESC_A,
            B, IB, JB, DESC_B,
            BETA, C, IC, JC, DESC_C)
```

# PDGEMM parameters



```
CALL PDGEMM(TRANSA, TRANSB, M, N, K, ALPHA,
            A, IA, JA, DESC_A,
            B, IB, JB, DESC_B,
            BETA, C, IC, JC, DESC_C)
```

- `M, N, K` define the sizes of $C$, $A$, and $B$.
- `IA, JA` and `IB, JB` select submatrices.
- `DESC_A`, `DESC_B`, `DESC_C` hold distribution metadata.
- `ALPHA` and `BETA` are scalars.

# Creating descriptors for PDGEMM



Assume $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$, $C \in \mathbb{R}^{m \times n}$.

```
call descinit(descA, m, k, nb, nb, 0, 0, ictxt, lldA, info)
call descinit(descB, k, n, nb, nb, 0, 0, ictxt, lldB, info)
call descinit(descC, m, n, nb, nb, 0, 0, ictxt, lldC, info)
```

- Block sizes are typically identical for all matrices.
- `lld*` are the local leading dimensions.

# Calling PDGEMM



```
call MPI_Barrier(MPI_COMM_WORLD, ierr)
t0 = MPI_Wtime()
do i = 1, nreps
  call pdgemm('N', 'N', m, n, k, alpha, A, 1, 1, descA, &
              B, 1, 1, descB, beta, C, 1, 1, descC)
end do
call MPI_Barrier(MPI_COMM_WORLD, ierr)
t1 = MPI_Wtime()
elapsed_time = t1 - t0
```

- Synchronize before and after to measure wall-clock time.
- `nreps` improves timing stability.

# GFLOPS for PDGEMM



The operation count for GEMM is

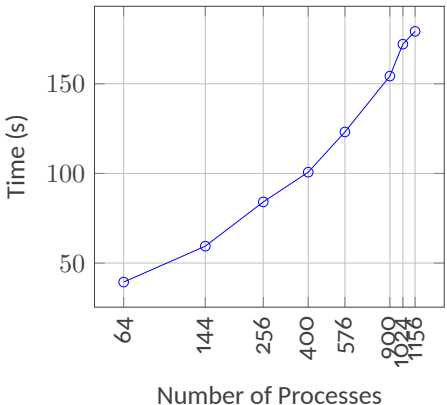
$$\text{FLOPS} = 2 \cdot m \cdot n \cdot k.$$

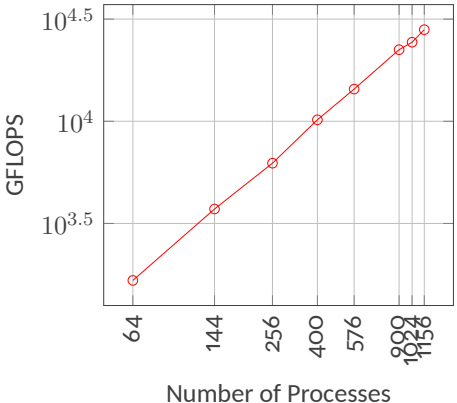
```
gflops = (2.0d0 * dble(m) * dble(n) * dble(k) * dble(nreps)) &
         / max_elapsed / 1.0d9
avg_time = max_elapsed / dble(nreps)
avg_gflops = (2.0d0 * dble(m) * dble(n) * dble(k)) / avg_time / 1.0d9
```

- Use the max time across processes for a conservative metric.

# PDGEMM performance example





# Conclusions, summary, and next steps



Today we have:

- Discussed the distributed matrix-vector multiplication and its scalability.
- Discussed the distributed matrix-matrix multiplication and its scalability.
- Presented practical aspects of using ScaLAPACK routines.

Next time:

- We will discuss more complex parallel algorithms for linear algebra.
- We will stat looking into GPU acceleration.

# High Performance Linear Algebra

## Lecture 13: LAPACK and ScaLAPACK numerical algorithms

Ph.D. program in High Performance Scientific Computing

**Fabio Durastante** **Pasqua D'Ambra** **Salvatore Filippone**

Monday 02, 2026 — 14.00:16.00

Dipartimento
di Matematica
Università di Pisa

# A recap of the previous lectures and today's plan

## 1 Last time on High Performance Linear Algebra

We have

- Reviewed the basic concepts of shared memory programming with OpenMP,
- Reviewed the basic concepts of distributed memory programming with MPI,
- We have discussed the BLAS for dense matrices and vectors:
  - All three levels of the BLAS in shared memory,
  - All three levels of the BLAS in distributed memory.

The plan for the next two lectures is to:

- Look at the LAPACK library for dense linear algebra in shared memory,
- Look at the ScaLAPACK library for dense linear algebra in distributed memory.

# Table of Contents

2 LAPACK: Linear Algebra PACKage


- ▶ LAPACK: Linear Algebra PACKage
  - Systems of linear equations
    - Storage schemes
  - Least-Squares Problems
    - Generalized Linear Least Squares Problems
  - Eigenproblems and Singular Value Decomposition
    - Symmetric Eigenproblems (SEP)
    - Nonsymmetric Eigenproblems (NEP)
    - Singular Value Decomposition (SVD)
    - Generalized Symmetric Definite Eigenproblems (GSEP)
    - Generalized Nonsymmetric Eigenproblems (GNEP)
    - Generalized Singular Value Decomposition (GSVD)
  - The Computational Routines

# LAPACK: Linear Algebra PACKage



- LAPACK is a software library for numerical linear algebra that provides routines for:
  - solving systems of linear equations,
  - least squares problems,
  - eigenvalue problems,
  - singular value decomposition,
  - and other related problems.
- It is designed to be efficient on modern computer architectures, taking advantage of cache memory and vectorization.
- LAPACK is written in Fortran and is widely used in scientific computing applications.
- It is built on top of the BLAS (Basic Linear Algebra Subprograms) library, which provides low-level routines for performing basic linear algebra operations.

# LAPACK: solving systems of linear equations



Two types of driver routines are provided for solving systems of linear equations:

- a simple driver (name ending `-SV`), which solves the system $AX = B$ by factorizing $A$ and overwriting $B$ with the solution $X$;
- an expert driver (name ending `-SVX`), which can also perform the following functions (some of them optionally):
  - solve $A^\top X = B$ or $A^H X = B$ (unless $A$ is symmetric or Hermitian);
  - estimate the condition number of $A$, check for near-singularity, and check for pivot growth;
  - refine the solution and compute forward and backward error bounds;
  - equilibrate the system if $A$ is poorly scaled.

The **expert driver** requires roughly twice as much storage as the simple driver in order to perform these extra functions.

# LAPACK: solving systems of linear equations



Two types of driver routines are provided for solving systems of linear equations:

- a simple driver (name ending `-SV`), which solves the system $AX = B$ by factorizing $A$ and overwriting $B$ with the solution $X$;
- an expert driver (name ending `-SVX`), which can also perform the following functions (some of them optionally):
  - solve $A^\top X = B$ or $A^H X = B$ (unless $A$ is symmetric or Hermitian);
  - estimate the condition number of $A$, check for near-singularity, and check for pivot growth;
  - refine the solution and compute forward and backward error bounds;
  - equilibrate the system if $A$ is poorly scaled.

Both types of driver routines can handle **multiple right hand sides** (the columns of $B$).

# LAPACK: solving systems of linear equations



Two types of driver routines are provided for solving systems of linear equations:

- a simple driver (name ending −SV), which solves the system $AX = B$ by factorizing $A$ and overwriting $B$ with the solution $X$;
- an expert driver (name ending −SVX), which can also perform the following functions (some of them optionally):
  - solve $A^\top X = B$ or $A^H X = B$ (unless $A$ is symmetric or Hermitian);
  - estimate the condition number of $A$, check for near-singularity, and check for pivot growth;
  - refine the solution and compute forward and backward error bounds;
  - equilibrate the system if $A$ is poorly scaled.

Different driver routines are provided to take advantage of special properties or storage schemes of the matrix $A$, for example, if $A$ is *symmetric*, *triangular*, *banded*, or *tridiagonal*.

# LAPACK: the simple drivers ?GESV



For a general (non-symmetric, non-Hermitian) matrix $A$, the simple driver routine to solve the system $AX = B$ is ?GESV, where the ? is the data-type placeholder.

```
SUBROUTINE ?GESV( N, NRHS, A, LDA, IPIV, B, LDB, INFO )
  INTEGER           N, NRHS, LDA, LDB, INFO
  INTEGER           IPIV( * )
  ?                 A( LDA, * ), B( LDB, * )
END SUBROUTINE ?GESV
```

- N is the order of the matrix $A$,
- NRHS is the number of right hand sides (the number of columns of $B$),
- A is the coefficient matrix $A$ (on entry) and its LU factorization (on exit),
- IPIV is an integer array of pivot indices,
- B is the right hand side matrix $B$ (on entry) and the solution matrix $X$ (on exit),
- INFO is an integer output variable that indicates success or failure of the routine.

# LAPACK: the simple drivers DGESV



Let us look at an example of usage of the simple driver DGESV to **solve** a **system of linear equations**.

A classical source of dense linear systems is the discretization of **integral equations**:

$$\int_a^b K(x,y)u(y)\mathrm{d}y = f(x), \quad x \in [a,b].$$

Using a quadrature rule with nodes $y_j$ and weights $w_j$, we can approximate the integral as

$$\sum_{j=1}^{n} w_j K(x_i,y_j)u(y_j) \approx f(x_i), \quad i = 1,\ldots,n,$$

which leads to the linear system

$$Au = f, \quad A_{ij} = w_j K(x_i,y_j).$$

# Discretization of a Stable Volterra Equation

## 2 LAPACK: Linear Algebra PACKage

We consider the Volterra integral equation of the **second kind**:

$$u(s) - \int_0^s (s-t)u(t)\mathrm{d}t = \sin(s), \quad s \in [0,1].$$

We discretize the domain into $n$ nodes $s_i = \frac{i-1}{n-1}$ with step size $h = \frac{1}{n-1}$. Using the **composite trapezoidal rule**, the integral is approximated for each node $s_i$:

$$\int_0^{s_i} (s_i - t)u(t)\mathrm{d}t \approx \sum_{j=1}^{i} w_{ij}(s_i - t_j)u(t_j), \quad t_j = s_j,$$

where the weights are $w_{ij} = \frac{h}{2}$ for the endpoints ($j = 1, i$) and $w_{ij} = h$ otherwise. This leads to a **lower triangular** linear system $(I - A)u = f$:

$$u_i - \sum_{j=1}^{i} A_{ij}u_j = f_i, \quad A_{ij} = w_{ij}(s_i - t_j), \quad i = 1, \ldots, n.$$

# LAPACK: writing the function building the matrix $A$ and $\mathbf{b}$



We need to write a function that builds the matrix $A$ and the right-hand side vector $b$.

```
subroutine build_system( n_size, mat, rhs, xplot_vec, sol_vec )
  use iso_fortran_env, only: dp => real64
  integer, intent(in) :: n_size
  real(dp), intent(out) :: mat(n_size,n_size), rhs(n_size), xplot_vec(n_size),
  ↪  sol_vec(n_size)
end subroutine build_system
```

- `n_size` is the size of the system,
- `mat` is the matrix $A$,
- `rhs` is the right-hand side vector $\mathbf{b}$,
- `xplot_vec` is the vector of $x$ coordinates for plotting,
- `sol_vec` is the vector of the exact solution for comparison.

# LAPACK: writing the function building the matrix $A$ and b



The implementation of the function is as follows:

```
integer :: i, j
real(dp) :: s_i, t_j, h

h = 1.0_dp / real(n_size - 1, dp)
mat = 0.0_dp
!$omp parallel do private(i,j,s_i,t_j) shared(mat,rhs,xplot_vec,sol_vec,h)
do i = 1, n_size
    s_i = real(i-1, dp) * h
    rhs(i) = sin(s_i)  ! f(s)
    xplot_vec(i) = s_i ! x coordinates for plotting
    sol_vec(i) = 0.5*sinh(s_i) + 0.5*sin(s_i)
    mat(i,i) = 1.0_dp  ! Identity part: x(s)
```

# LAPACK: writing the function building the matrix $A$ and $\mathbf{b}$



```
    do j = 1, i
      t_j = real(j-1, dp) * h ! Kernel K(s, t) = s - t
      if (j == 1 .or. j == i) then
          mat(i,j) = mat(i,j) - (s_i - t_j) * (h / 2.0_dp)
      else
          mat(i,j) = mat(i,j) - (s_i - t_j) * h
      end if
    end do
end do
!$omp end parallel do
```

- We initialize the matrix and vectors,
- We use OpenMP to parallelize the outer loop over $i$,
- We compute the entries of the matrix $A$ and the right-hand side vector $\mathbf{b}$.

# LAPACK: the solution step



The remaining part of the program uses the LAPACK routine `DGESV` to solve the linear system, after reading the matrix size from the command line and using the `build_system` subroutine to create the matrix and right-hand side vector.

```
program linear_system_solve
   use, intrinsic :: iso_fortran_env, only: wp => real64, error_unit
   implicit none
   character(len=20) :: arg
   integer :: n, info
   real(wp), allocatable :: A(:,:), b(:), x(:), xplot(:), sol(:)
   integer, allocatable :: ipiv(:)

   ! Read matrix size from command line arguments
   if (command_argument_count() < 1) then
```

# LAPACK: the solution step



```
    write(error_unit, *) "Usage: linear_system_solve <matrix_size>"
    stop 1
end if
call get_command_argument(1, arg)
read(arg, *) n

! Allocate arrays
allocate(A(n,n), b(n), x(n), xplot(n), sol(n), ipiv(n))

! Initialize matrix A and vector b
call build_system(n, A, b, xplot, sol)
! Solve the linear system A*x = b using LAPACK
x = b
```

# LAPACK: the solution step



```
call dgesv(n, 1, A, n, ipiv, x, n, info)

! Output the solution vector x to file "solution.out"
open(unit=10, file="solution.out", status="replace", action="write",
↪  iostat=info)
if (info /= 0) then
   write(error_unit, *) "Error opening output file"
   stop 1
end if
write(10, '(A)') "x computed exact"
do n = 1, size(x)
   write(10, '(E24.16,E24.16,E24.16)') xplot(n), x(n), sol(n)
end do
```

# LAPACK: the solution step



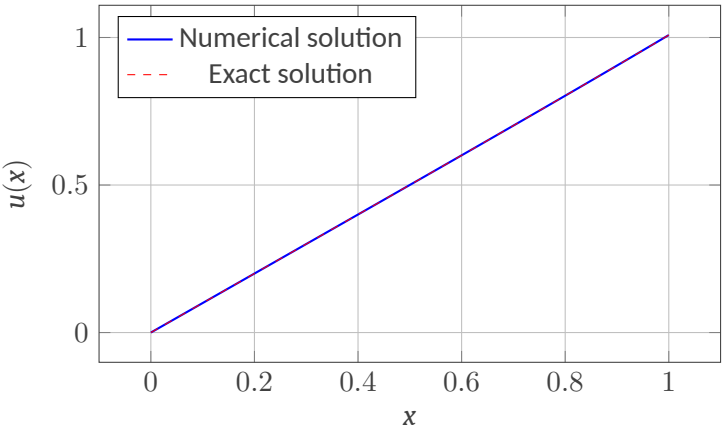
```
close(10)

! Deallocate arrays
deallocate(A, b, x, xplot, sol, ipiv)
stop 0
```

- The program reads the matrix size from command line arguments,
- Allocates the necessary arrays,
- Calls the `build_system` subroutine to initialize the matrix and vector,
- Uses `DGESV` to solve the linear system,
- Outputs the computed solution and exact solution to a file for comparison.

# Visualization of the solution of the integral equation



Computing the solution with $n = 100$ nodes and plotting the numerical solution against the exact solution:



# A remark on integral equations and dense linear systems



- For large-scale problems, storing and manipulating dense matrices can be memory-intensive and computationally expensive.
- However, the matrices arising from integral equations often have **special structures** that can be exploited:
  - **Low-rank approximations**: The matrix may be well-approximated by a low-rank matrix, reducing storage and computation.
  - **Hierarchical matrices** (H-matrices): Exploit block-wise low-rank structure for efficient storage and computation.
  - **Sparse representations**: Using techniques like the Fast Multipole Method (FMM) to avoid explicit matrix construction.
- These alternative storage formats can significantly reduce memory requirements and computational complexity compared to standard dense matrix representations.

# LAPACK: the other linear system routines



| Type of matrix and storage scheme | Operation | Single precision real | Single precision complex | Double precision real | Double precision complex |
|---|---|---|---|---|---|
| General | simple driver | SGESV | CGESV | DGESV | ZGESV |
| | expert driver | SGESVX | CGESVX | DGESVX | ZGESVX |
| General band | simple driver | SGBSV | CGBSV | DGBSV | ZGBSV |
| | expert driver | SGBSVX | CGBSVX | DGBSVX | ZGBSVX |
| General tridiagonal | simple driver | SGTSV | CGTSV | DGTSV | ZGTSV |
| | expert driver | SGTSVX | CGTSVX | DGTSVX | ZGTSVX |
| Sym./Herm. pos. def. | simple driver | SPOSV | CPOSV | DPOSV | ZPOSV |
| | expert driver | SPOSVX | CPOSVX | DPOSVX | ZPOSVX |

# LAPACK: the other linear system routines



| Type of matrix and storage scheme | Operation | Single precision real | Single precision complex | Double precision real | Double precision complex |
|---|---|---|---|---|---|
| Sym./Herm. pos. def. (packed) | simple driver | SPPSV | CPPSV | DPPSV | ZPPSV |
| | expert driver | SPPSVX | CPPSVX | DPPSVX | ZPPSVX |
| Sym./Herm. pos. def. band | simple driver | SPBSV | CPBSV | DPBSV | ZPBSV |
| | expert driver | SPBSVX | CPBSVX | DPBSVX | ZPBSVX |
| Sym./Herm. pos. def. tridiagonal | simple driver | SPTSV | CPTSV | DPTSV | ZPTSV |
| | expert driver | SPTSVX | CPTSVX | DPTSVX | ZPTSVX |
| Sym./Herm. indefinite | simple driver | SSYSV | CHESV | DSYSV | ZHESV |
| | expert driver | SSYSVX | CHESVX | DSYSVX | ZHESVX |

# LAPACK: the other linear system routines



| Type of matrix and storage scheme | Operation | Single precision real | Single precision complex | Double precision real | Double precision complex |
|---|---|---|---|---|---|
| Complex symmetric | simple driver | | CSYSV | | ZSYSV |
| | expert driver | | CSYSVX | | ZSYSVX |
| Sym./Herm. indefinite (packed) | simple driver | SSPSV | CHPSV | DSPSV | ZHPSV |
| | expert driver | SSPSVX | CHPSVX | DSPSVX | ZHPSVX |
| Complex symmetric (packed) | simple driver | | CSPSV | | ZSPSV |
| | expert driver | | CSPSVX | | ZSPSVX |

- The table summarizes the LAPACK routines for solving systems of linear equations for various types of matrices and **storage schemes**.

# LAPACK: storage schemes



Generally, LAPACK supports different storage schemes for matrices to optimize memory usage and computational efficiency. The main storage schemes are:

- **Full storage**: The entire matrix is stored in a two-dimensional array. This is the most straightforward representation but can be inefficient for large matrices.
- **Banded storage**: Only the non-zero bands of a banded matrix are stored,
- **Packed storage**: Only the non-zero elements of symmetric or Hermitian matrices are stored in a one-dimensional array,

We have already seen examples of **full storage**, let us briefly discuss the other storage schemes.

# LAPACK: banded storage scheme



In the **banded storage scheme**, only the non-zero bands of a banded matrix are stored in a compact form.
A banded matrix has non-zero elements only within a certain bandwidth around the main diagonal.

- `KL` is the number of subdiagonals (non-zero elements below the main diagonal),
- `KU` is the number of superdiagonals (non-zero elements above the main diagonal),
- `LDAB` is the leading dimension of the array `AB`, which must be at least `2*KL+KU+1`,
- `AB(KL+KU+1+i-j,j) = A(i,j)` for `max(1,j-KU)<=i<=min(N,j+KL)`

For example, a matrix with $KL = 1$ (one subdiagonal) and $KU = 2$ (two superdiagonals) would be stored in a compact form requiring only $2 \times 1 + 2 + 1 = 5$ rows instead of storing the full matrix.

# LAPACK: banded storage scheme example.



Consider the following banded matrix $A$ with $KL = 1$ and $KU = 2$:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 \\ 0 & a_{32} & a_{33} & a_{34} & a_{35} \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{bmatrix}$$

In **banded storage**, this matrix would be stored in the array `AB` as follows:

$$AB = \begin{bmatrix} 0 & 0 & a_{13} & a_{24} & a_{35} \\ a_{12} & a_{23} & a_{34} & a_{45} & 0 \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & 0 \end{bmatrix}$$

Here,

- the first row contains the second superdiagonal,
- the second row contains the first superdiagonal,
- the third row contains the main diagonal,
- the fourth row contains the first subdiagonal.

# LAPACK: banded storage scheme example.



Consider the following banded matrix $A$ with $KL = 1$ and $KU = 2$:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 \\ 0 & a_{32} & a_{33} & a_{34} & a_{35} \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{bmatrix}$$

In **banded storage**, this matrix would be stored in the array `AB` as follows:

$$AB = \begin{bmatrix} 0 & 0 & a_{13} & a_{24} & a_{35} \\ a_{12} & a_{23} & a_{34} & a_{45} & 0 \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & 0 \end{bmatrix}$$

Here,

- the first row contains the second superdiagonal,
- the second row contains the first superdiagonal,
- the third row contains the main diagonal,
- the fourth row contains the first subdiagonal.

# LAPACK: banded storage scheme example.



Consider the following banded matrix $A$ with $KL = 1$ and $KU = 2$:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 \\ 0 & a_{32} & a_{33} & a_{34} & a_{35} \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{bmatrix}$$

In **banded storage**, this matrix would be stored in the array `AB` as follows:

$$AB = \begin{bmatrix} 0 & 0 & a_{13} & a_{24} & a_{35} \\ a_{12} & a_{23} & a_{34} & a_{45} & 0 \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & 0 \end{bmatrix}$$

Here,

- the first row contains the second superdiagonal,
- the second row contains the first superdiagonal,
- the third row contains the main diagonal,
- the fourth row contains the first subdiagonal.

# LAPACK: banded storage scheme example.



Consider the following banded matrix $A$ with $KL = 1$ and $KU = 2$:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 \\ 0 & a_{32} & a_{33} & a_{34} & a_{35} \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{bmatrix}$$

In **banded storage**, this matrix would be stored in the array `AB` as follows:

$$AB = \begin{bmatrix} 0 & 0 & a_{13} & a_{24} & a_{35} \\ a_{12} & a_{23} & a_{34} & a_{45} & 0 \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & 0 \end{bmatrix}$$

Here,

- the first row contains the second superdiagonal,
- the second row contains the first superdiagonal,
- the third row contains the main diagonal,
- the fourth row contains the first subdiagonal.

# LAPACK: routine for banded storage



The LAPACK routine for solving systems of linear equations with banded storage is `?GBSV` for the simple driver and `?GBSVX` for the expert driver.

- These routines take as input the banded matrix in the compact form,
- They perform LU factorization with partial pivoting to solve the system efficiently,
- They are particularly useful when dealing with large banded matrices, as they reduce memory usage and computational time compared to full storage methods.

```
SUBROUTINE ?GBSV( N, NRHS, KL, KU, AB, LDAB, IPIV, B, LDB, INFO )
  INTEGER           N, NRHS, KL, KU, LDAB, LDB, INFO
  INTEGER           IPIV( * )
  ?                 AB( LDAB, * ), B( LDB, * )
END SUBROUTINE ?GBSV
```

# LAPACK: routine for banded storage



The LAPACK routine for solving systems of linear equations with banded storage is `?GBSV` for the simple driver and `?GBSVX` for the expert driver.

- These routines take as input the banded matrix in the compact form,
- They perform LU factorization with partial pivoting to solve the system efficiently,
- They are particularly useful when dealing with large banded matrices, as they reduce memory usage and computational time compared to full storage methods.

**On exit**, details of the factorization: $U$ is stored as an *upper triangular band matrix* with `KL+KU` superdiagonals in rows `1` to `KL+KU+1`, and *the multipliers* used during the factorization *are stored in rows* `KL+KU+2` to `2*KL+KU+1`.

# LAPACK: the tridiagonal case



A special case of banded matrices is the **tridiagonal matrix**, which has non-zero elements only on the main diagonal and the first sub- and super-diagonals.
The LAPACK routine for solving systems of linear equations with tridiagonal matrices is `?GTSV` for the simple driver and `?GTSVX` for the expert driver.

```
SUBROUTINE ?GTSV( N, NRHS, DL, D, DU, B, LDB, INFO )
  INTEGER          N, NRHS, LDB, INFO
  ?                DL( * ), D( * ), DU( * ), B( LDB, * )
END SUBROUTINE ?GTSV
```

- `DL` is the subdiagonal elements of size `N-1`,
- `D` is the main diagonal elements of size `N`,
- `DU` is the superdiagonal elements of size `N-1`.

# Lapack: packed storage scheme



In the **packed storage scheme**, only the non-zero elements of symmetric or Hermitian matrices are stored in a one-dimensional array. This is particularly useful for large symmetric or Hermitian matrices, as it reduces memory usage significantly.

For a **symmetric matrix**, only the *upper* or *lower* triangular part needs to be stored, as the other part can be inferred due to symmetry.

- → The packed storage format stores the elements column-wise (or row-wise) in a one-dimensional array,
- → This format is efficient for both storage and computation, especially when combined with LAPACK routines designed to work with packed storage.

# LAPACK: routine for packed storage



The LAPACK routine for solving systems of linear equations with packed storage is `?PPSV` for the simple driver and `?PPSVX` for the expert driver.

```
SUBROUTINE ?PPSV( UPLO, N, NRHS, AP, B, LDB, INFO )
  CHARACTER          UPLO
  INTEGER           N, NRHS, LDB, INFO
  ?                 AP( * ), B( LDB, * )
END SUBROUTINE ?PPSV
```

- `UPLO` indicates whether the upper or lower triangular part of the matrix is stored,
- `AP` is the one-dimensional array containing the packed storage of the matrix.

# LAPACK: routine for packed storage



The LAPACK routine for solving systems of linear equations with packed storage is `?PPSV` for the simple driver and `?PPSVX` for the expert driver.

```
SUBROUTINE ?PPSV( UPLO, N, NRHS, AP, B, LDB, INFO )
  CHARACTER          UPLO
  INTEGER           N, NRHS, LDB, INFO
  ?                 AP( * ), B( LDB, * )
END SUBROUTINE ?PPSV
```

- `UPLO` indicates whether the upper or lower triangular part of the matrix is stored,
- `AP` is the one-dimensional array containing the packed storage of the matrix.

If `UPLO = 'U'`, the **upper triangular** part of the matrix is stored column-wise in `AP` as follows:

$$AP = [a_{11}, a_{12}, a_{22}, a_{13}, a_{23}, a_{33}, \ldots, a_{1N}, a_{2N}, \ldots, a_{NN}]$$

# LAPACK: routine for packed storage



The LAPACK routine for solving systems of linear equations with packed storage is `?PPSV` for the simple driver and `?PPSVX` for the expert driver.

```
SUBROUTINE ?PPSV( UPLO, N, NRHS, AP, B, LDB, INFO )
  CHARACTER          UPLO
  INTEGER           N, NRHS, LDB, INFO
  ?                 AP( * ), B( LDB, * )
END SUBROUTINE ?PPSV
```

- `UPLO` indicates whether the upper or lower triangular part of the matrix is stored,
- `AP` is the one-dimensional array containing the packed storage of the matrix.

If `UPLO = 'L'`, the **lower triangular** part of the matrix is stored column-wise in `AP` as follows:

$$AP = [a_{11}, a_{21}, a_{22}, a_{31}, a_{32}, a_{33}, \ldots, a_{N1}, a_{N2}, \ldots, a_{NN}]$$

# LAPACK: routine for packed storage



The LAPACK routine for solving systems of linear equations with packed storage is `?PPSV` for the simple driver and `?PPSVX` for the expert driver.

```
SUBROUTINE ?PPSV( UPLO, N, NRHS, AP, B, LDB, INFO )
  CHARACTER           UPLO
  INTEGER            N, NRHS, LDB, INFO
  ?                  AP( * ), B( LDB, * )
END SUBROUTINE ?PPSV
```

- `UPLO` indicates whether the upper or lower triangular part of the matrix is stored,
- `AP` is the one-dimensional array containing the packed storage of the matrix.
- `N` is the order of the matrix,
- `NRHS` is the number of right hand sides,
- `B` is the right hand side matrix.
- `LDB` is the leading dimension of the array `B`.
- `INFO` is an integer output variable that indicates success or failure of the routine.

# LAPACK: routine for packed storage



The LAPACK routine for solving systems of linear equations with packed storage is `?PPSV` for the simple driver and `?PPSVX` for the expert driver.

```
SUBROUTINE ?PPSV( UPLO, N, NRHS, AP, B, LDB, INFO )
  CHARACTER          UPLO
  INTEGER           N, NRHS, LDB, INFO
  ?                 AP( * ), B( LDB, * )
END SUBROUTINE ?PPSV
```

On **exit**, the solution matrix $X$ overwrites the right hand side matrix $B$.

The routine uses the Cholesky factorization to solve the system efficiently while taking advantage of the packed storage format. The factor $U$ or $L$ from the Cholesky factorization $A = U^\top U$ or $A = LL^\top$, in the same storage format as $A$.

# LAPACK: Linear Least Squares Problems



The **linear least squares problem** is:

$$\underset{x}{\text{minimize}} \, \|b - Ax\|_2$$

where $A$ is an $m$-by-$n$ matrix, $b$ is a given $m$ element vector, and $x$ is the $n$ element solution vector.

In the most usual case $m \geq n$ and $\text{rank}(A) = n$, the solution to the problem is unique, and the problem is referred to as finding a least squares solution to an overdetermined system of linear equations.

# LAPACK: Linear Least Squares Problems



The **linear least squares problem** is:

$$\underset{x}{\text{minimize}} \, \|b - Ax\|_2$$

where $A$ is an $m$-by-$n$ matrix, $b$ is a given $m$ element vector, and $x$ is the $n$ element solution vector.

When $m < n$ and $\text{rank}(A) = m$, there are infinitely many solutions $x$ that exactly satisfy $b - Ax = 0$. In this case, it is useful to find the unique solution $x$ which minimizes $\|x\|_2$, referred to as finding a minimum norm solution to an underdetermined system of linear equations.

# LAPACK: Linear Least Squares Problems



The **linear least squares problem** is:

$$\underset{x}{\text{minimize}} \, \|b - Ax\|_2$$

where $A$ is an $m$-by-$n$ matrix, $b$ is a given $m$ element vector, and $x$ is the $n$ element solution vector.

In the general case when $\text{rank}(A) < \min(m, n)$ (i.e., $A$ may be rank-deficient), we seek the minimum norm least squares solution $x$ which minimizes both $\|x\|_2$ and $\|b - Ax\|_2$.

# LAPACK: Driver routines for least squares problems



The driver routine `DGELS` solves the linear least squares problem assuming $\text{rank}(A) = \min(m, n)$, i.e., $A$ has **full rank**.

- ➔ It finds a least squares solution of an overdetermined system when $m > n$,
- ➔ It finds a minimum norm solution of an underdetermined system when $m < n$,
- ➔ It uses QR or LQ factorization of $A$,
- ➔ It allows $A$ to be replaced by $A^\top$ (or $A^H$ if $A$ is complex).

For rank-deficient matrices, several routines are available:

- `DGELSY` — uses **complete orthogonal factorization**,
- `DGELSS` — uses **singular value decomposition** (SVD),
- `DGELSD` — uses **divide-and-conquer SVD** (faster for large problems).

# LAPACK: LLS driver routines summary



| Operation | Single precision | | Double precision | |
|---|---|---|---|---|
| | real | complex | real | complex |
| Solve LLS using QR or LQ | SGELS | CGELS | DGELS | ZGELS |
| Solve LLS using COF | SGELSY | CGELSY | DGELSY | ZGELSY |
| Solve LLS using SVD | SGELSS | CGELSS | DGELSS | ZGELSS |
| Solve LLS using DC-SVD | SGELSD | CGELSD | DGELSD | ZGELSD |

- All routines can handle multiple right hand sides stored as columns of $B$ and $X$.
- Each right hand side is solved independently, i.e., we do not find:

$$\arg\min ||B - AX||_2^2.$$

# LLS: The different algorithms

## 2 LAPACK: Linear Algebra PACKage

- **QR and LQ factorization**: $A = QR$ or $A = LQ$ with $Q$ orthogonal and $R$ (or $L$) triangular.
  - Suitable for full-rank matrices,
  - Efficient for both overdetermined and underdetermined systems,
  - Utilizes orthogonal transformations to minimize numerical errors,
  - Cost: $O(mn^2)$ for $m \geq n$ and $O(nm^2)$ for $m < n$.
- **Complete Orthogonal Factorization (COF)**: $A = Q \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix} P^T$, with $Q$ orthogonal, $P$ a permutation matrix, and $R_{11}$ upper triangular, and $R_{12}$ dense,
  - Handles rank-deficient matrices,
  - Decomposes $A$ into orthogonal and triangular matrices,
  - Provides a stable solution even when $A$ is ill-conditioned,
  - Cost: $O(mn^2)$ for $m \geq n$ and $O(nm^2)$ for $m < n$.

# LLS: The different algorithms

## 2 LAPACK: Linear Algebra PACKage

- **Singular Value Decomposition (SVD)**: $A = U\Sigma V^T$, with $U$ and $V$ orthogonal and $\Sigma$ diagonal,
  - Robust method for rank-deficient matrices,
  - Decomposes $A$ into singular values and vectors,
  - Minimizes the effect of small singular values on the solution,
  - Cost: $O(mn^2 + n^3)$ for $m \geq n$ and $O(nm^2 + m^3)$ for $m < n$.
- **Divide-and-Conquer SVD**:
  - An efficient variant of SVD for large problems,
  - Divides the matrix into smaller submatrices,
  - Combines results to obtain the final solution,
  - Cost: $O(mn^2)$ for $m \geq n$ and $O(nm^2)$ for $m < n$ (faster than traditional SVD for large-scale problems).

# LAPACK: Generalized Linear Least Squares Problems



Driver routines are provided for two types of generalized linear least squares problems.

The **first** is the linear equality-constrained least squares problem (LSE):

$$\min_x \|c - Ax\|_2 \quad \text{subject to} \quad Bx = d$$

where $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{p \times n}$, $c$ is a given $m$-vector, $d$ is a given $p$-vector, with $p \leq n \leq m + p$.

- The routine `DGGLSE` solves this problem using the **generalized RQ factorization**,
- Assumes $B$ has full row rank $p$ and $\begin{bmatrix} A \\ B \end{bmatrix}$ has full column rank $n$,
- Under these assumptions, the problem has a unique solution.

# LAPACK: Generalized Linear Model Problem



The **second** is the general linear model problem (GLM):

$$\min_x \|y\|_2 \quad \text{subject to} \quad d = Ax + By$$

where $A \in \mathbb{R}^{n \times m}$, $B \in \mathbb{R}^{n \times p}$, and $d$ is a given $n$-vector, with $m \leq n \leq m + p$.

- When $B = I$, the problem reduces to an ordinary linear least squares problem,
- When $B$ is square and nonsingular, it is equivalent to the **weighted linear least squares problem**:

$$\min_x \|B^{-1}(d - Ax)\|_2$$

The routine `DGGGLM` solves this problem using the **generalized QR factorization**:

- Assumes $A$ has full column rank $m$ and $(A, B)$ has full row rank $n$,
- Under these assumptions, there are unique solutions $x$ and $y$.

# LAPACK: Generalized Linear Model Problem



| Operation | Single precision | | Double precision | |
|---|---|---|---|---|
| | real | complex | real | complex |
| Solve LSE using GRQ | SGGLSE | CGGLSE | DGGLSE | ZGGLSE |
| Solve GLM using GQR | SGGGLM | CGGGLM | DGGGLM | ZGGGLM |

- The GRQ decomposes the pair $(A, B)$ into orthogonal and triangular matrices as

$$A = RQ \text{ and } B = ZTQ, \;\; Q, Z \text{ orthogonal}, \;\; R, T \text{ upper triangular}$$

- The GQR decomposes the pair $(A, B)$ into orthogonal and triangular matrices as

$$A = QR \text{ and } B = QTZ, \;\; Q, Z \text{ orthogonal}, \;\; R, T \text{ upper triangular};$$

# Eigenproblems and Singular Value Decomposition

## 2 LAPACK: Linear Algebra PACKage

LAPACK provides a comprehensive set of routines for solving various types of eigenvalue problems and performing singular value decomposition (SVD).

These routines are designed to handle **different matrix types**, including general, symmetric, Hermitian, and banded matrices.

Key features of LAPACK's eigenproblem and SVD routines include:

- Efficient algorithms for computing eigenvalues and eigenvectors,
- Support for both real and complex matrices,
- Routines for computing the SVD of general matrices,
- Specialized routines for symmetric and Hermitian matrices to exploit their properties for improved performance.

# LAPACK: Symmetric Eigenproblems (SEP)



The **symmetric eigenvalue problem** is to find the **eigenvalues** $\lambda$ and corresponding **eigenvectors** $z \neq 0$, such that:

$$Az = \lambda z, \quad A = A^\top, \quad \text{where } A \text{ is real.}$$

For the **Hermitian eigenvalue problem** we have:

$$Az = \lambda z, \quad A = A^H.$$

For both problems the eigenvalues $\lambda$ are real.

When all eigenvalues and eigenvectors have been computed, we write:

$$A = Z\Lambda Z^\top$$

where $\Lambda$ is a diagonal matrix whose diagonal elements are the eigenvalues, and $Z$ is an orthogonal (or unitary) matrix whose columns are the eigenvectors. This is the classical **spectral factorization** of $A$.

# Driver routines for symmetric eigenproblems



There are four types of driver routines for symmetric and Hermitian eigenproblems:

- A simple driver (name ending `-EV`) computes all the eigenvalues and (optionally) eigenvectors,
- An expert driver (name ending `-EVX`) computes all or a selected subset of the eigenvalues and eigenvectors,
- A divide-and-conquer driver (name ending `-EVD`) solves the same problem as the simple driver but is much faster for large matrices,
- A **relatively robust representation** (RRR) driver (name ending `-EVR`) is the fastest algorithm and uses the least workspace.

Different driver routines are provided to take advantage of special structure or storage of the matrix $A$.

# LAPACK: Driver routines for symmetric eigenproblems



| Function and storage scheme | Single precision | | Double precision | |
|---|---|---|---|---|
| | real | complex | real | complex |
| simple driver | SSYEV | CHEEV | DSYEV | ZHEEV |
| divide and conquer driver | SSYEVD | CHEEVD | DSYEVD | ZHEEVD |
| expert driver | SSYEVX | CHEEVX | DSYEVX | ZHEEVX |
| RRR driver | SSYEVR | CHEEVR | DSYEVR | ZHEEVR |
| simple driver (packed) | SSPEV | CHPEV | DSPEV | ZHPEV |
| divide and conquer (packed) | SSPEVD | CHPEVD | DSPEVD | ZHPEVD |
| expert driver (packed) | SSPEVX | CHPEVX | DSPEVX | ZHPEVX |

# LAPACK: Driver routines for symmetric eigenproblems



| Function and storage scheme | Single precision | | Double precision | |
|---|---|---|---|---|
| | real | complex | real | complex |
| simple driver (band) | SSBEV | CHBEV | DSBEV | ZHBEV |
| divide and conquer (band) | SSBEVD | CHBEVD | DSBEVD | ZHBEVD |
| expert driver (band) | SSBEVX | CHBEVX | DSBEVX | ZHBEVX |
| simple driver (tridiagonal) | SSTEV | | DSTEV | |
| divide and conquer (tridiagonal) | SSTEVD | | DSTEVD | |
| expert driver (tridiagonal) | SSTEVX | | DSTEVX | |
| RRR driver (tridiagonal) | SSTEVR | | DSTEVR | |

# Nonsymmetric Eigenproblems (NEP)



The **nonsymmetric eigenvalue problem** is to find the **eigenvalues** $\lambda$ and corresponding **right eigenvectors** $v \neq 0$, such that:

$$Av = \lambda v.$$

A real matrix $A$ may have complex eigenvalues, occurring as complex conjugate pairs. A vector $u \neq 0$ satisfying:

$$u^H A = \lambda u^H$$

is called a **left eigenvector** of $A$.

This problem can be solved via the **Schur factorization** of $A$:

$$A = ZTZ^\top \quad \text{(real case)}, \quad A = ZTZ^H \quad \text{(complex case)}$$

where $Z$ is orthogonal (or unitary) and $T$ is an upper quasi-triangular (or triangular) matrix.

# LAPACK: Driver routines for nonsymmetric eigenproblems



| Function | Single precision | | Double precision | |
|---|---|---|---|---|
| | real | complex | real | complex |
| simple driver for Schur factorization | SGEES | CGEES | DGEES | ZGEES |
| expert driver for Schur factorization | SGEESX | CGEESX | DGEESX | ZGEESX |
| simple driver for eigenvalues/vectors | SGEEV | CGEEV | DGEEV | ZGEEV |
| expert driver for eigenvalues/vectors | SGEEVX | CGEEVX | DGEEVX | ZGEEVX |

# LAPACK: Driver routines for nonsymmetric eigenproblems



Two pairs of drivers are provided:

- `xGEES` and `xGEESX`: compute the Schur factorization of $A$, with optional ordering of eigenvalues,
- `xGEEV` and `xGEEVX`: compute all eigenvalues and (optionally) right or left eigenvectors.

- The **expert drivers** (`xGEESX` and `xGEEVX`) can additionally balance the matrix and compute condition numbers for the eigenvalues or eigenvectors.

# Singular Value Decomposition (SVD)



The **singular value decomposition** of an $m$-by-$n$ matrix $A$ is given by:

$$A = U\Sigma V^\top \quad (A = U\Sigma V^H \text{ in the complex case})$$

where $U$ and $V$ are orthogonal (unitary) and $\Sigma$ is an $m$-by-$n$ diagonal matrix with real diagonal elements $\sigma_i$ such that:

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_{\min(m,n)} \geq 0.$$

The $\sigma_i$ are the **singular values** of $A$ and the first $\min(m, n)$ columns of $U$ and $V$ are the **left** and **right singular vectors** of $A$.

# Driver routines for SVD



Two types of driver routines are provided for the SVD:

- A simple driver `xGESVD` computes all the singular values and (optionally) left and/or right singular vectors,
- A divide-and-conquer driver `xGESDD` solves the same problem but is much faster for large matrices.

- The divide-and-conquer driver uses more workspace but is significantly faster than the simple driver for large matrices.

# Generalized Symmetric Definite Eigenproblems (GSEP)



Drivers are provided to compute all the eigenvalues and (optionally) the eigenvectors of the following types of problems:

1. $Az = \lambda Bz$
2. $ABz = \lambda z$
3. $BAz = \lambda z$

where $A$ and $B$ are symmetric or Hermitian and $B$ is positive definite.

For all these problems the eigenvalues $\lambda$ are real. The matrices $Z$ of computed eigenvectors satisfy:

- $Z^\top AZ = \Lambda$ (problem types 1 and 3) or $Z^{-1}AZ^{-\top} = I$ (problem type 2),
- $Z^\top BZ = I$ (problem types 1 and 2) or $Z^\top B^{-1}Z = I$ (problem type 3),

where $\Lambda$ is a diagonal matrix with the eigenvalues on the diagonal.

# Driver routines for GSEP



Three types of driver routines are provided for generalized symmetric and Hermitian eigenproblems:

- A simple driver (name ending `-GV`) computes all the eigenvalues and (optionally) eigenvectors,
- An expert driver (name ending `-GVX`) computes all or a selected subset of the eigenvalues and eigenvectors. If few enough eigenvalues or eigenvectors are desired, the expert driver is faster than the simple driver,
- A divide-and-conquer driver (name ending `-GVD`) solves the same problem as the simple driver but is much faster for large matrices, although it uses more workspace.

Different driver routines are provided to take advantage of special structure or storage of the matrices $A$ and $B$.

# LAPACK: Driver routines for GSEP



| Function and storage scheme | Single precision | | Double precision | |
|---|---|---|---|---|
| | real | complex | real | complex |
| simple driver | SSYGV | CHEGV | DSYGV | ZHEGV |
| divide and conquer driver | SSYGVD | CHEGVD | DSYGVD | ZHEGVD |
| expert driver | SSYGVX | CHEGVX | DSYGVX | ZHEGVX |
| simple driver (packed) | SSPGV | CHPGV | DSPGV | ZHPGV |
| divide and conquer (packed) | SSPGVD | CHPGVD | DSPGVD | ZHPGVD |
| expert driver (packed) | SSPGVX | CHPGVX | DSPGVX | ZHPGVX |
| simple driver (band) | SSBGV | CHBGV | DSBGV | ZHBGV |
| divide and conquer (band) | SSBGVD | CHBGVD | DSBGVD | ZHBGVD |
| expert driver (band) | SSBGVX | CHBGVX | DSBGVX | ZHBGVX |

# Generalized Nonsymmetric Eigenproblems (GNEP)



Given a matrix pair $(A, B)$, where $A$ and $B$ are square $n \times n$ matrices, the **generalized nonsymmetric eigenvalue problem** is to find the **eigenvalues** $\lambda$ and corresponding **eigenvectors** $x \neq 0$ such that:

$$Ax = \lambda Bx$$

or to find the eigenvalues $\mu$ and corresponding eigenvectors $y \neq 0$ such that:

$$\mu Ay = By$$

These problems are equivalent with $\mu = 1/\lambda$ and $x = y$ if neither $\lambda$ nor $\mu$ is zero.

# Matrix Pencils and Eigenvalue Representation

## 2 LAPACK: Linear Algebra PACKage

To deal with cases where $\lambda$ or $\mu$ is zero or nearly so, LAPACK routines return two values, $\alpha$ and $\beta$, for each eigenvalue, such that:

$$\lambda = \alpha/\beta \quad \text{and} \quad \mu = \beta/\alpha$$

Vectors $u \neq 0$ or $v \neq 0$ satisfying:

$$u^\top A = \lambda u^\top B \quad \text{or} \quad \mu v^\top A = v^\top B$$

are called **left eigenvectors**.

The matrix pencil $A - \lambda B$ is used to refer to the generalized eigenproblem. The problem is called:

- **Regular** if $\det(A - \lambda B) \not\equiv 0$ for all $\lambda$,
- **Singular** if $\det(A - \lambda B) \equiv 0$ for all $\lambda$ (signaled by $\alpha = \beta = 0$).

# Generalized Schur Decomposition



The generalized nonsymmetric eigenvalue problem can be solved via the **generalized Schur decomposition** of the matrix pair $(A, B)$.

In the **real case**:

$$A = QSZ^\top, \quad B = QTZ^\top$$

In the **complex case**:

$$A = QSZ^H, \quad B = QTZ^H$$

where $Q$ and $Z$ are orthogonal (or unitary), $T$ is upper triangular, and $S$ is upper quasi-triangular with $1 \times 1$ and $2 \times 2$ diagonal blocks.

The columns of $Q$ and $Z$ are called **left and right generalized Schur vectors** and span pairs of **deflating subspaces** of $A$ and $B$.

# Driver routines for GNEP



Two pairs of drivers are provided:

- `xGGES` and `xGGESX`: compute the generalized Schur decomposition of $(A, B)$, with optional ordering of eigenvalues,
- `xGGEV` and `xGGEVX`: compute all generalized eigenvalues and (optionally) right or left eigenvectors.

The **expert drivers** (`xGGESX` and `xGGEVX`) can additionally:

- Balance the matrix pair to improve conditioning,
- Compute condition numbers for the eigenvalues or eigenvectors.

# Summary of GNEP driver routines



| Function | Single precision | | Double precision | |
|---|---|---|---|---|
| | real | complex | real | complex |
| simple driver (Schur) | SGGES | CGGES | DGGES | ZGGES |
| expert driver (Schur) | SGGESX | CGGESX | DGGESX | ZGGESX |
| simple driver (eigenvalues) | SGGEV | CGGEV | DGGEV | ZGGEV |
| expert driver (eigenvalues) | SGGEVX | CGGEVX | DGGEVX | ZGGEVX |

# Generalized Singular Value Decomposition (GSVD)



The **generalized (or quotient) singular value decomposition** of an $m \times n$ matrix $A$ and a $p \times n$ matrix $B$ is given by:

$$A = U\Sigma_1[0, R]Q^\top \quad \text{and} \quad B = V\Sigma_2[0, R]Q^\top$$

where:

- $U$ is $m \times m$, $V$ is $p \times p$, $Q$ is $n \times n$, all orthogonal (or unitary for complex),
- $R$ is $r \times r$, upper triangular and nonsingular, where $r$ is the rank of $\begin{bmatrix} A \\ B \end{bmatrix}$,
- $\Sigma_1$ is $m \times r$ and $\Sigma_2$ is $p \times r$, both real, nonnegative, and diagonal.

The ratios $\alpha_1/\beta_1, \ldots, \alpha_r/\beta_r$ are called the **generalized singular values** of the pair $(A, B)$.

# Special Cases of GSVD



Important special cases of the generalized singular value decomposition include:

- If $B$ is square and nonsingular, then $r = n$ and the GSVD is equivalent to the SVD of $AB^{-1}$,
- If the columns of $\begin{bmatrix} A^\top & B^\top \end{bmatrix}^\top$ are orthonormal, then $r = n$, $R = I$, and the GSVD is equivalent to the **Cosine-Sine (CS) decomposition**,
- The generalized eigenvalues of $A^\top A - \lambda B^\top B$ can be expressed in terms of the GSVD.

# Driver routine for GSVD



A single driver routine `xGGSVD` computes the generalized singular value decomposition of $A$ and $B$.

| Function | Single precision | | Double precision | |
|---|---|---|---|---|
| | real | complex | real | complex |
| GSVD | SGGSVD | CGGSVD | DGGSVD | ZGGSVD |

- The method is based on the generalized QR and RQ factorizations,
- It is useful for solving certain least squares and generalized eigenvalue problems,
- The GSVD provides a unified framework for understanding relationships between different matrix decompositions.

# Computational Routines

## 2 LAPACK: Linear Algebra PACKage

Driver routines call a sequence of **computational routines** (also called auxiliary routines) to perform specific tasks:

**Factorization routines** compute matrix factorizations (LU, QR, Cholesky, etc.),

**Solver routines** solve systems using a precomputed factorization,

**Eigenvalue routines** reduce matrices to condensed forms (tridiagonal, Hessenberg),

**Utility routines** perform auxiliary operations (scaling, orthogonal transformations).

- Users can call computational routines directly for **finer control** and **better performance**,
- Useful when you need to **reuse a factorization** for multiple operations,
- Allows **combining different algorithms** in a custom sequence.

# LAPACK: Factorization Routines



Examples of factorization computational routines:

- `xGETRF`: LU factorization with partial pivoting,
- `xGETRS`: solve using LU factorization,
- `xGEQRF`: QR factorization,
- `xGERQF`: RQ factorization,
- `xGELQF`: LQ factorization,
- `xORMQR` (or `xUNMQR`): apply orthogonal transformation from QR,
- `xPOTRF`: Cholesky factorization.

Combining computational routines allows implementing **custom algorithms**:

```
! Compute QR factorization
call dgeqrf(m, n, A, lda, tau, work, lwork, info)
! Apply Q to a vector
call dormqr('L', 'T', m, nrhs, n, A, lda, tau, B, ldb, work, lwork, info)
```

# LAPACK: Solver Routines



Solver routines use precomputed factorizations to solve linear systems efficiently:

- `xGETRS`: solve using LU factorization from `xGETRF`,
- `xPOTRS`: solve using Cholesky factorization from `xPOTRF`,
- `xTRTRS`: solve triangular systems,
- `xGBTRS`: solve banded systems using factorization from `xGBTRF`,
- `xGTTRS`: solve tridiagonal systems using factorization from `xGTTRF`.

These routines typically perform $O(n^2)$ operations instead of $O(n^3)$ for factorization:

```
! Factorize once
call dgetrf(n, n, A, lda, ipiv, info)
! Solve for multiple right-hand sides
do i = 1, nrhs
  call dgetrs('N', n, 1, A, lda, ipiv, B(:,i), n, info)
end do
```

# LAPACK: Solver Routines



Solver routines use precomputed factorizations to solve linear systems efficiently:

- `xGETRS`: solve using LU factorization from `xGETRF`,
- `xPOTRS`: solve using Cholesky factorization from `xPOTRF`,
- `xTRTRS`: solve triangular systems,
- `xGBTRS`: solve banded systems using factorization from `xGBTRF`,
- `xGTTRS`: solve tridiagonal systems using factorization from `xGTTRF`.

- Separating factorization and solving provides significant performance gains for multiple systems,
- Essential for iterative refinement and other advanced techniques.

# LAPACK: Eigenvalue Reduction and Schur Manipulation



**Reducing to Hessenberg form** is often the first step in computing eigenvalues of nonsymmetric matrices:

- `xGEHRD`: reduce a general matrix to upper Hessenberg form,
- `xORGHR` (or `xUNGHR`): generate the orthogonal matrix from the reduction,
- Hessenberg form has zeros below the first subdiagonal, reducing subsequent eigenvalue computations.

**Schur factorization manipulation**:

- `xTREXC`: reorder eigenvalues in Schur factorization by exchanging diagonal blocks,
- `xTRSEN`: reorder and compute condition numbers for selected eigenvalues,
- Useful for isolating specific eigenvalues or organizing them by magnitude or stability.

# LAPACK: Eigenvalue Reduction and Schur Manipulation



**Solving Sylvester equations**:

- `xTRSYL`: solve the generalized Sylvester equation $AX + XB = C$ or related forms,
- Requires Schur factorizations of $A$ and $B$ as input,
- Applications include model reduction, control theory, and matrix equation solutions.

```
! Reduce to Hessenberg form
call dgehrd(n, 1, n, A, lda, tau, work, lwork, info)
! Reorder Schur form
call dtrexc('V', n, T, ldt, Q, ldq, ifst, ilst, work, info)
! Solve Sylvester equation: AX + XB = C
call dtrsyl('N', 'N', 1, m, n, A, lda, B, ldb, C, ldc, scale, info)
```

# Why Use Computational Routines?



**Factorization reuse** Compute factorization once, use it for multiple right-hand sides or operations

**Memory efficiency** Control allocation and reuse of intermediate arrays

**Algorithm customization** Combine different routines to implement specialized algorithms

**Performance** Avoid redundant computations when multiple related operations are needed

**Stability control** Apply equilibration or scaling before factorization for better conditioning

**Example use case**: Solve multiple systems $Ax_i = b_i$ where $b_i$ depends on $x_{i-1}$:

1. Compute LU factorization of $A$ once using `DGETRF`,
2. For each $b_i$, call `DGETRS` with the precomputed factors,
3. Much faster than calling `DGESV` repeatedly.

# Conclusions

3 Conclusions and next step

- LAPACK provides a suite of routines for solving LA problems efficiently,
- Driver routines offer high-level interfaces for common tasks,
- Computational routines allow fine-grained control,
- Understanding both types of routines enables users to optimize performance and tailor solutions to specific needs.

- Next step: look at the **ScaLAPACK** library for distributed-memory parallel computing,
- ScaLAPACK extends LAPACK's capabilities to large-scale problems across multiple processors.

# High Performance Linear Algebra

## Lecture 14: LAPACK and ScaLAPACK numerical algorithms

Ph.D. program in High Performance Scientific Computing

**Fabio Durastante** **Pasqua D'Ambra** **Salvatore Filippone**

Tuesday 03, 2026 — 14.00:16.00

Dipartimento
di Matematica
Università di Pisa

# Table of Contents

1 ScaLAPACK Numerical Routines

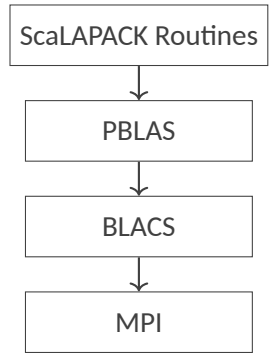

- ScaLAPACK Numerical Routines
  - Linear System Solution
    - The Divide and Conquer Algorithm
  - Linear Least Squares Problems
  - Symmetric Eigenproblems and Singular Value Decomposition
  - Generalized Symmetric Definite Eigenproblems
  - Computational Routines
    - Linear Equations
    - Orthogonal factorization and linear least-square
    - Generalized Orthogonal Factorizations
    - Symmetric Eigenproblems
    - General Eigenproblems
    - Singular Value Decomposition
    - Generalized Symmetric Definite Eigenproblems

# Summary of ScaLAPACK Infrastructure



**ScaLAPACK** = Scalable Linear Algebra PACKage

- Distributed-memory extension of LAPACK for parallel computing
- Built on three key layers:
  1. **MPI** (Message Passing Interface) — low-level communication
  2. **BLACS** (Basic Linear Algebra Communication Subprograms) — higher-level communication primitives
  3. **PBLAS** (Parallel BLAS) — distributed matrix operations
- Matrix distributed across process grid using 2D block-cyclic layout




Each layer **abstracts communication details**, enabling scalable algorithms

# Initialization: process grid and matrix distribution



**Process grid setup**:

```
CALL BLACS_GRIDINIT( ICTXT, 'R', NPROW, NPCOL )
```

where `ICTXT` is the BLACS context, `NPROW` and `NPCOL` define the process grid dimensions.

**Matrix descriptor creation**:

```
CALL DESCINIT( DESC, M, N, MB, NB, RSRC, CSRC, ICTXT, LLD, INFO )
```

where `DESC` is the descriptor array for the distributed matrix, `M`, `N` are global matrix dimensions, `MB`, `NB` are block sizes, and `RSRC`, `CSRC` specify the process owning the first block.

**Information on the local part of the matrix** can be obtained using:

```
CALL NUMROC( N, NB, IPROC, ISRCPROC, NPROCS )
```

which computes the number of rows or columns of the distributed matrix owned by a specific process.

# ScaLAPACK: Driver Routines for Linear Systems



Two types of driver routines are provided for solving systems of linear equations:

- **Simple driver** (name ending `-SV`):
  - Solves the system $AX = B$ by factorizing $A$ and overwriting $B$ with the solution $X$
- **Expert driver** (name ending `-SVX`):
  - Solve $A^\top X = B$ or $A^H X = B$ (unless $A$ is symmetric or Hermitian)
  - Estimate the condition number of $A$, check for near-singularity and pivot growth
  - Refine the solution and compute forward and backward error bounds
  - Equilibrate the system if $A$ is poorly scaled

Expert driver requires roughly twice as much storage to perform these extra functions.

Both types handle multiple right-hand sides (columns of $B$). Different drivers exploit special properties or storage schemes of matrix $A$.

**Note:** For band/tridiagonal matrices (`PxDBTRF`, `PxDTTRF`, `PxGBTRF`, `PxPBTRF`, `PxPTTRF`), the factorization differs from LAPACK due to additional permutations for parallelism.

# ScaLAPACK: Linear System Solution Drivers



| Type of matrix and storage scheme | Operation | Single precision | | Double precision | |
|---|---|---|---|---|---|
| | | Real | Complex | Real | Complex |
| general (partial pivoting) | simple driver | PSGESV | PCGESV | PDGESV | PZGESV |
| | expert driver | PSGESVX | PCGESVX | PDGESVX | PZGESVX |
| general band (partial pivoting) | simple driver | PSGBSV | PCGBSV | PDGBSV | PZGBSV |
| general band (no pivoting) | simple driver | PSDBSV | PCDBSV | PDDBSV | PZDBSV |
| general tridiagonal (no pivoting) | simple driver | PSDTSV | PCDTSV | PDDTSV | PZDTSV |

# ScaLAPACK: Linear System Solution Drivers



| Type of matrix and storage scheme | Operation | Single precision | | Double precision | |
|---|---|---|---|---|---|
| | | Real | Complex | Real | Complex |
| symmetric/Hermitian positive definite | simple driver | PSPOSV | PCPOSV | PDPOSV | PZPOSV |
| | expert driver | PSPOSVX | PCPOSVX | PDPOSVX | PZPOSVX |
| symmetric/Hermitian positive definite band | simple driver | PSPBSV | PCPBSV | PDPBSV | PZPBSV |
| symmetric/Hermitian positive definite tridiagonal | simple driver | PSPTSV | PCPTSV | PDPTSV | PZPTSV |

# Divide and Conquer for Banded Linear Systems



The algorithm we discuss is based on the **divide and conquer** strategy introduced in

- J. Dongarra and L. Johnsson. Solving banded systems on a parallel processor. Parallel Computing, 5:219–246, 1987,
- A. Cleary and J. Dongarra. Implementation in ScaLAPACK of Divide-and-Conquer Algorithms for Banded and Tridiagonal Linear Systems. Computer Science Dept. Technical Report CS-97-358, University of Tennessee, Knoxville, TN, April 1997. (Also LAPACK Working Note 125).

The main idea is to **partition the banded matrix into smaller submatrices**, solve the smaller systems independently, and then combine the solutions to obtain the solution of the original system.

# Divide and Conquer for Banded Linear Systems



$$A\mathbf{x} = \mathbf{b},\ \text{lower bandwidth } \beta_l,\ \text{upper bandwidth } \beta_u$$

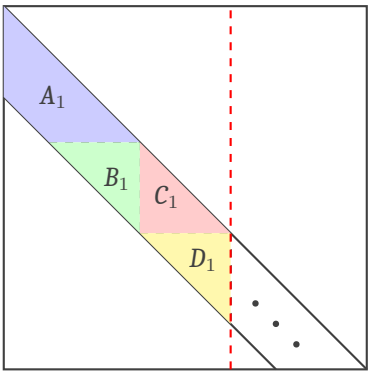
The algorithm follows these steps:

1. First produce a reordering: $PA(P^{-1}P)\mathbf{x} = P\mathbf{b}$, where $P$ is a permutation matrix,
2. The reorderd matrix $PAP^{-1}$ is factore as $LU$ or $LL^\top$ via Gaussian Elimination/Chokesky,
3. Solve the system $LU\mathbf{x}' = \mathbf{b}'$ (or $LL^\top\mathbf{x}' = \mathbf{b}'$) where $\mathbf{x}' = P\mathbf{x}$ and $\mathbf{b}' = P\mathbf{b}$,
   - 3.1 Solve $L\mathbf{z} = \mathbf{b}'$ via forward substitution,
   - 3.2 Solve $U\mathbf{x}' = \mathbf{z}$ via back substitution,
4. Finally, recover the solution $\mathbf{x} = P^{-1}\mathbf{x}'$.

Find a good $n \times n$ permutation matrix $P$ that reorders $A$ to allow exploitation of parallelism,

# The Symmetric and Positive Definite Case

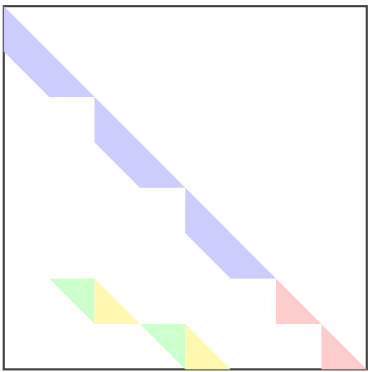





- User inputs the matrix in **lower triangular form**,
- Each processor stores a contiguous set of columns of the matrix
- We partition each process matrix:
  - $A_i$: "trapezoidal" block along the diagonal of size $O_i$,
  - $B_i$, $C_i$, $D_i$: lower triangular blocks of size $\beta \times \beta$,
  - The **last processor** has only the $A_i$ block.

# The Symmetric and Positive Definite Case





The **reordering goes** as follows:

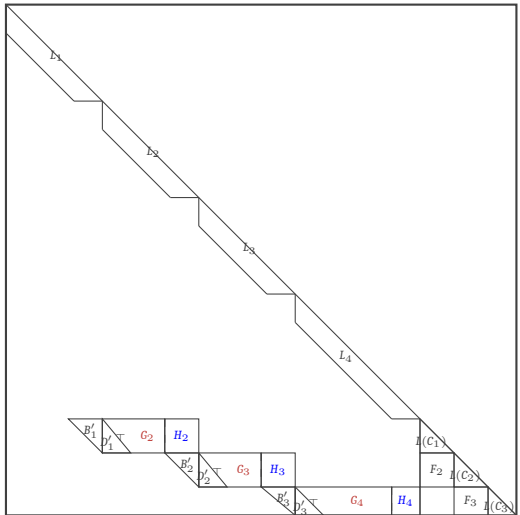
- Number the equations in the $A_i$ first, keeping the same relative order,
- Number the equations in the $C_i$ next, keeping the same relative order,

The **Cholesky factorization** of the reordered matrix can be computed with sequential block operations.

⚠ We do not physically reorder the matrix but, rather, we base block operations on the reordering.

# Cholesky Factor of the Permuted Matrix






- **Factorization**: largely computed with sequential block operations, minimal communication required.
- **Fill-in**: $G_i$ and $H_i$ represent fill-in, doubling nonzeros compared to sequential algorithms.
- **Operation Count**: $O(4N\beta^2)$
  - $A_i$ factorization: $N\beta^2$.
  - Forming $G_i$: $2N\beta^2$.
  - Updating $C_i$ with $G_i$: $N\beta^2$.
- **Total**: Approx. $4\times$ operations of the sequential algorithm.

# The three phases of the Divide and Conquer Algorithm



**Phase 1** Formation of the **reduced system**.
Each processor does computations independently (for the most part) with local parts and then combines to form the Schur complement system corresponding to the parts already factored.
The Schur complement is often called the reduced system.

**Phase 2** The reduced system is solved, and the answers are communicated back to all of the processors.

**Phase 3** The solutions from Phase 2 are applied in a backsolution process.

# Phase 1: Formation of the Reduced System



We look at the *i*-th processor, first and last processors have special cases.

- **Communication Step:** $D_i$ is sent to processor $i+1$. Since this is a small communication, it is completely overlapped with subsequent computations.

At this point, portions of the matrix are stored locally.

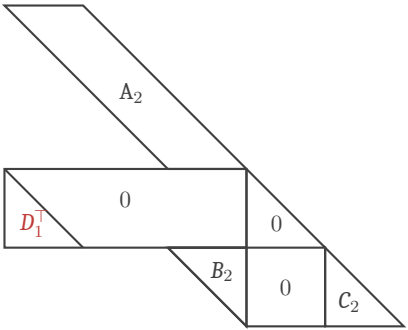
- We treat local computations as a **frontal computation**.
- We perform $O_i$ factorization steps and apply them to the remaining submatrix of size $2\beta$.
- This submatrix is subsequently used in **Phase 2** to form the reduced system.

The "divide" in the algorithm's name stems from the reordering allowing each defined front to be independent.

Only the $2\beta$ update equations at the end of each front need be coordinated with other processors.

# Phase 1: Continued

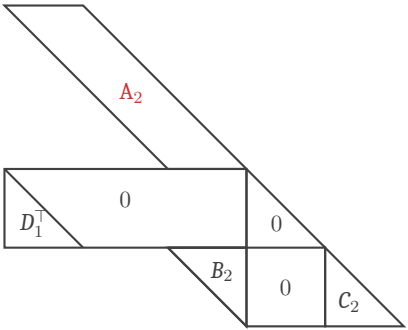





- Start the exchange of $D_i$ block with process $i + 1$.

# Phase 1: Continued



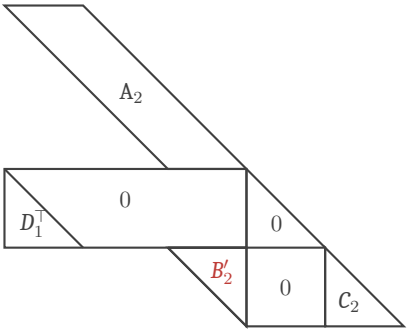



```
call dpbtrf( ... , A_i, ... )
```

- Start the exchange of $D_i$ block with process $i+1$.
- Take $O_i$ steps of **Cholesky factorization** on the local front:
  - Factor $A_i$ to get $L(A_i)$,

# Phase 1: Continued



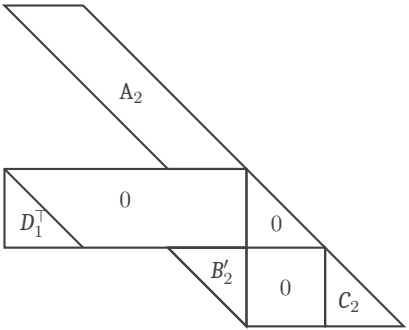



```
call dtrtrs( ... , B_i, ... )
```

- Start the exchange of $D_i$ block with process $i+1$.
- Take $O_i$ steps of **Cholesky factorization** on the local front:
  - Factor $A_i$ to get $L(A_i)$,
  - Update $B_i$ using $L_i = L(A_i)$: $L_i {B'_i}^\top = B_i^\top$,

# Phase 1: Continued



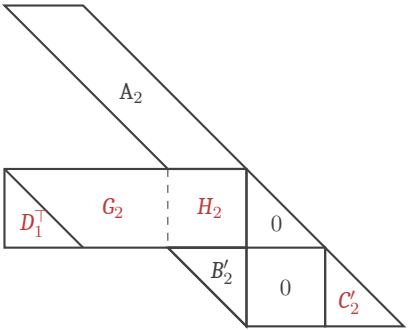



```
call dtrmm( ... , B_i, ... )
```

- Start the exchange of $D_i$ block with process $i+1$.
- Take $O_i$ steps of **Cholesky factorization** on the local front:
  - Factor $A_i$ to get $L(A_i)$,
  - Update $B_i$ using $L_i = L(A_i)$: $L_i {B_i'}^\top = B_i^\top$,
  - Update $C_i$ using $B_i'$ as $C_i' = C_i - B_i' {B_i'}^\top$.

# Phase 1: Continued



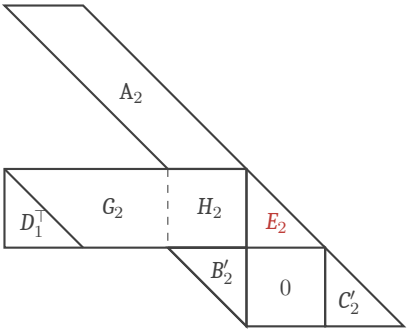



```
call dtbtrs( ... , B_i, ... )
```

- Start the exchange of $D_i$ block with process $i+1$.
- Take $O_i$ steps of **Cholesky factorization** on the local front:
  - Factor $A_i$ to get $L(A_i)$,
  - Update $B_i$ using $L_i = L(A_i)$: $L_i {B'_i}^\top = B_i^\top$,
  - Update $C_i$ using $B'_i$ as $C'_i = C_i - B'_i {B'_i}^\top$.
- The exchanged $D_i$ is **now needed**:
  - Compute $G_i$ from $L_i G_i^\top = D_i$,

# Phase 1: Continued



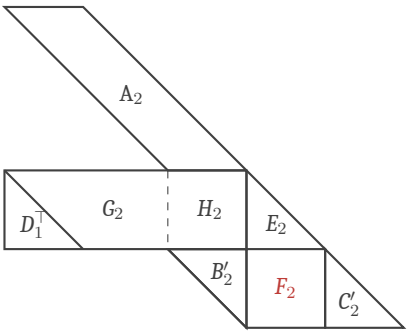



```
call dsyrk( ... , G_i, ... )
```

- Start the exchange of $D_i$ block with process $i+1$.
- Take $O_i$ steps of **Cholesky factorization** on the local front:
  - Factor $A_i$ to get $L(A_i)$,
  - Update $B_i$ using $L_i = L(A_i)$: $L_i {B'_i}^\top = B_i^\top$,
  - Update $C_i$ using $B'_i$ as $C'_i = C_i - B'_i {B'_i}^\top$.
- The exchanged $D_i$ is **now needed**:
  - Compute $G_i$ from $L_i G_i^\top = D_i$,
  - The matrix $E_i$ represents the contribution from processor $i$ to the diagonal block of the reduced system stored on process $i-i$, that is $C'_{i-1}$: $E_i = G_i G_i^\top$

# Phase 1: Continued





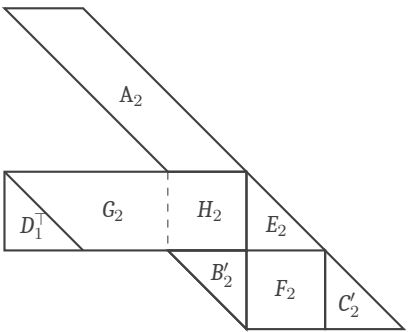

```
call dtrmm( ... , B_i, ... )
```

— The local computation is finished by computing $F_i$, using $B_i'$ and the last $\beta$ columns of $G_i$, which we have labelled $H_i$:

$$F_i^\top = H_i B_i'^\top$$

# Phase 1: Continued






— The local computation is finished by computing $F_i$, using $B_i'$ and the last $\beta$ columns of $G_i$, which we have labelled $H_i$:

$$F_i^\top = H_i B_i'^\top$$

- The processor is now ready for **Phase 2**.

# Phase 2: Solution of the Reduced System



Phase 2 consists of the forming and factorization of the Schur complement matrix.

- Each processor contributes three blocks of size $\beta \times \beta$ to this system: $E_i$, $F_i$, $C_i'$.
- Each $C_i'$ is added to $E_{i+1}$ to form the diagonal blocks of the matrix,.
- The $F_i$ form the off-diagonal blocks.

The resultant system is **block tridiagonal**, with $P - 1$ blocks.

Several methods for factoring the reduced system have been proposed:

- **For small $P$ or small $\beta$**: Perform an all-to-all broadcast of each processor's portion of the reduced system.
  - 👍 Disadvantage: Entire reduced system ends up on each processor.
  - 👍 Advantage: Only one (expensive) communication step.
  - 👎 Disadvantage: Redundant computation (serial algorithm), will not scale.
- **Parallel Solution**: For larger problems, ScaLAPACK uses a parallel block tridiagonal solver scaling as $O(\log P)$ or $P$.

# Phase 2: Solution of the Reduced System



We use a block formulation of **odd-even (or cyclic) reduction**.

- This algorithm has $\log_2 P$ stages.
- At each stage, the **odd-numbered blocks** are used to "eliminate" the **even-numbered blocks**.
- The process decreases the number of blocks left by a factor of **two** at each stage.
- Symmetry is maintained throughout (Cholesky factorization of a symmetric permutation of the reduced system).

**Reordering Strategy:**

- Blocks are ordered so that even-numbered blocks in Step 1 are first, those in Step 2 correspond to second, and so on.
- Results in an elimination tree of minimal height.

⚠ Implementation requires additional space allocation for fill-in created by the reordering (though of much lower order than Phase 1 fill-in).

# Phase 3: Backsolution



- Phase 3 is specific to the solution step (not factorization).
- Operations performed mirror the factorization steps (Phase 1 and 2) but operate on the **right-hand sides**.

**Procedure** At the end of Phase 2, processors hold portions of the solution to the reduced system.

**Communication** Each processor distributes $2\beta$ elements of this solution to neighboring processors.

**Computation** Partial solutions are back-substituted into locally stored factors. This is a completely local computation stage.

- Structure is similar to factorization but simpler.
- Uses block operations from **LAPACK** and **BLAS**.
- Multiple right-hand sides are handled efficiently in this context.

# Running an example



We want to **run an example** of the parallel divide and conquer algorithm:

```
call pdpbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
```

Where:

- `uplo`: 'L' for lower triangular storage,
- `n`: order of the matrix,
- `bw`: bandwidth of the matrix,
- `nrhs`: number of right-hand sides,
- `a`: local array containing the lower triangular part of the matrix,
- `ja`: global column index of the first local column of `a`,
- `desca`: array descriptor for matrix `a`,
- `b`: local array containing the right-hand side(s),
- `ib`: global row index of the first local row of `b`,
- `descb`: array descriptor for matrix `b`,
- `work`: workspace array,
- `lwork`: size of the workspace array,
- `info`: output status variable.

# The construction of the input matrix



- Initialize matrix $A$ in banded format,
- Create a symmetric positive definite banded matrix,
- In packed banded format for `UPLO='U'`:
  $A_{i,j}$ is stored in `A(BW+1+i-j, j)` for $\max(1, j - \text{BW}) \leq i \leq j$.

```
a = 0.0_real64
do j = 1, loc_n_a
  ! Global column index
  jloc = (mycol * nb) + j
  if (jloc <= n) then
      ! Diagonal element (make it dominant for positive definiteness)
      a(bw + 1 + (j-1)*lld_a) = 4.0_real64 + 2.0_real64 * real(bw, real64)
```

# The construction of the input matrix



```
      ! Off-diagonal elements
      do i = max(1, jloc - bw), jloc - 1
        if (i >= 1 .and. i < jloc) then
            iloc = bw + 1 + i - jloc
            if (iloc >= 1 .and. iloc <= bw + 1) then
              a(iloc + (j-1)*lld_a) = -1.0_real64
            end if
        end if
      end do
  end if
end do
```

# The construction of the input matrix



The previous code build the following matrix for $n = 8$ and $bw = 2$:

$$A = \begin{bmatrix} 6 & -1 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 6 & -1 & -1 & 0 & 0 & 0 & 0 \\ -1 & -1 & 6 & -1 & -1 & 0 & 0 & 0 \\ 0 & -1 & -1 & 6 & -1 & -1 & 0 & 0 \\ 0 & 0 & -1 & -1 & 6 & -1 & -1 & 0 \\ 0 & 0 & 0 & -1 & -1 & 6 & -1 & -1 \\ 0 & 0 & 0 & 0 & -1 & -1 & 6 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 6 \end{bmatrix}$$

which is symmetric positive definite and banded with bandwidth 2.

# The work spaces



- ScaLAPACK routines use a **workspace array** `WORK` of size `LWORK` to store temporary data.
- The size of `LWORK` depends on the routine and the problem size.
- If `LWORK` is too small, the routine returns an error.

For `PDPBSV`, the minimal size is:

$$\texttt{LWORK} \geq (\texttt{NB} + 2 \cdot \texttt{BW}) \cdot \texttt{BW} + \max((\texttt{BW} \cdot \texttt{NRHS}), \texttt{BW}^2)$$

- `NB`: Block size,
- `BW`: Bandwidth,
- `NRHS`: Number of right-hand sides.

**Query mechanism**: If `LWORK = -1`, the routine calculates the optimal size and returns it in `WORK(1)`. This is the recommended way to allocate the workspace.

# The work spaces



- ScaLAPACK routines use a **workspace array** `WORK` of size `LWORK` to store temporary data.
- The size of `LWORK` depends on the routine and the problem size.
- If `LWORK` is too small, the routine returns an error.

For `PDPBSV`, the minimal size is:

$$\texttt{LWORK} \geq (\texttt{NB} + 2 \cdot \texttt{BW}) \cdot \texttt{BW} + \max((\texttt{BW} \cdot \texttt{NRHS}), \texttt{BW}^2)$$

- `NB`: Block size,
- `BW`: Bandwidth,
- `NRHS`: Number of right-hand sides.

```
lwork = (nb + 2*bw) * bw + max(bw*nrhs, bw*bw)
allocate(work(lwork), stat=info)
```

# Calling the ScaLAPACK routine and measure time



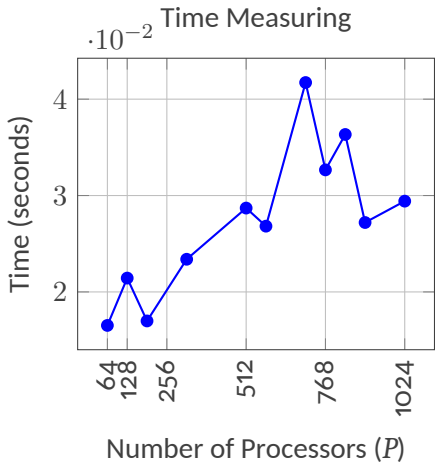
Finally, we can call the ScaLAPACK routine to solve the system $A\mathbf{x} = \mathbf{b}$:

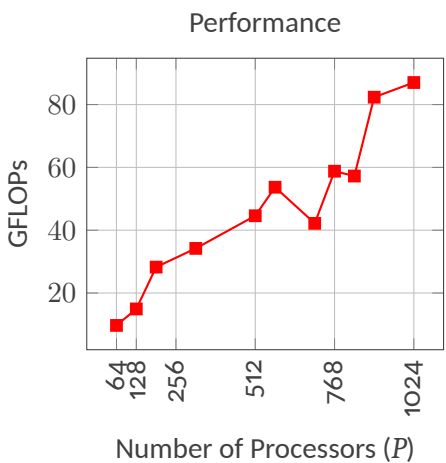
```
! Start timing
t_start = mpi_wtime()
! Call PDPBSV to solve the system
call pdpbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, &
            work, lwork, info)
! End timing
t_end = mpi_wtime()
t_elapsed = t_end - t_start
! All reduce to get maximum time across all processors
call mpi_allreduce(mpi_in_place, t_elapsed, 1, mpi_double_precision, &
                   mpi_max, mpi_comm_world, ierr)
```

# Scaling Results

1 ScaLAPACK Numerical Routines

We run the **weak scaling** on the Amelia cluster at IAC-CNR, with Intel 2023.



# Scaling Results



**Observations**:

- The time for the solution of the linear system remains constant as we increase the number of processors $P$ while keeping the problem size $N$ proportional to $P$.
- This is the expected behavior for a **weak scaling** experiment:
  - $N = P \times$ local_size.
  - Ideal weak scaling behavior: execution time is constant.
- The **GFLOPs** metric increases linearly with $P$, showing that we are able to effectively use the added computational power.

The divide and conquer algorithm for banded systems in ScaLAPACK exhibits good parallel scalability, effectively handling the inherent dependencies of the banded structure through the hierarchical decomposition.

# Linear Least Squares Problems



The **linear least squares (LLS) problem** is defined as finding $x$ that minimizes:

$$\min_x \|b - Ax\|_2$$

where $A$ is an $m \times n$ matrix and $b$ is an $m$-element vector.

**Overdetermined ($m \geq n$)**

- If full rank ($n$), the solution is unique.
- "Least squares solution".

**Underdetermined ($m < n$)**

- If full rank ($m$), infinite solutions satisfy $b - Ax = 0$.
- We seek the **minimum norm solution** which minimizes $\|x\|_2$.

In the **rank-deficient** case ($rank(A) < \min(m, n)$), we seek the **minimum norm least squares solution** which minimizes both $\|b - Ax\|_2$ and $\|x\|_2$.

# ScaLAPACK LLS Driver: PxGELS



The driver routine `PxGELS` solves the LLS problem assuming $A$ has **full rank**.

- Finds the LLS solution for $m \geq n$ and the minimum norm solution for $m < n$.
- Uses **QR factorization** or **LQ factorization** of $A$.
- Can handle $A$ or $A^H$ (or $A^\top$ for real matrices).
- Handles multiple right-hand sides (columns of $B$) in a single call.

**Driver Routines for Linear Least Squares:**

| Problem Type | Driver | Single precision | | Double precision | |
|---|---|---|---|---|---|
| | | Real | Complex | Real | Complex |
| Full rank LLS | `PxGELS` | `PSGELS` | `PCGELS` | `PDGELS` | `PZGELS` |

# Symmetric Eigenproblems (SEP)



The **symmetric/Hermitian eigenvalue problem** is to find the **eigenvalues** $\lambda$ and corresponding **eigenvectors** $z \neq 0$ such that:

$$Az = \lambda z, \quad \text{where } A = A^\top \text{ (symmetric) or } A = A^H \text{ (Hermitian).}$$

The eigenvalues $\lambda$ are always real.
When all eigenvalues and eigenvectors are computed, we can write the **spectral factorization** $A = Z\Lambda Z^H$, where $\Lambda$ is diagonal containing eigenvalues, and $Z$ is orthogonal (or unitary) containing eigenvectors.

Two types of driver routines are provided:

- **Simple driver** (name ending `-EV`):
  - Computes all eigenvalues and (optionally) eigenvectors.
- **Expert driver** (name ending `-EVX`):
  - Computes either all or a selected subset of eigenvalues.
  - Optionally computes corresponding eigenvectors.

# Singular Value Decomposition (SVD)



The **Singular Value Decomposition (SVD)** of an $m \times n$ matrix $A$ is given by:

$$A = U\Sigma V^\top \quad (\text{or } A = U\Sigma V^H \text{ for complex})$$

where $U$ and $V$ are orthogonal (unitary) and $\Sigma$ is an $m$-by-$n$ diagonal matrix with real diagonal elements $\sigma_i$, such that:

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{\min(m,n)} \geq 0$$

The $\sigma_i$ are the **singular values**, and the first $\min(m, n)$ columns of $U$ and $V$ are the **left** and **right singular vectors**, satisfying:

$$Av_i = \sigma_i u_i \quad \text{and} \quad A^\top u_i = \sigma_i v_i$$

**Driver Routine:** `PxGESVD`

# Singular Value Decomposition (SVD)



- Computes the "economy size" or "thin" SVD.
- If $m > n$: only the first $n$ columns of $U$ are computed.

| Operation | Driver | Single precision | | Double precision | |
|---|---|---|---|---|---|
| | | Real | Complex | Real | Complex |
| Thin SVD | PxGESVD | PSGESVD | PCGESVD | PDGESVD | PZGESVD |

# Generalized Symmetric Definite Eigenproblems (GSEP)



An **expert driver** is provided to compute all (or selected) eigenvalues and (optionally) eigenvectors for the following problems:

1. $Az = \lambda Bz$
2. $ABz = \lambda z$
3. $BAz = \lambda z$

where $A$ and $B$ are symmetric (or Hermitian) and $B$ is positive definite.

**Properties**:

- The eigenvalues $\lambda$ are real.
- The matrix of eigenvectors $Z$ satisfies orthogonality conditions relative to $B$ (e.g., $Z^H BZ = I$ for type 1).

# Generalized Symmetric Definite Eigenproblems (GSEP)



An **expert driver** is provided to compute all (or selected) eigenvalues and (optionally) eigenvectors for the following problems:

1. $Az = \lambda Bz$
2. $ABz = \lambda z$
3. $BAz = \lambda z$

where $A$ and $B$ are symmetric (or Hermitian) and $B$ is positive definite.

**Driver Routines (Expert)**:

| Matrix Type | Single precision | | Double precision | |
|---|---|---|---|---|
| | Real | Complex | Real | Complex |
| Symmetric | PSSYGVX | – | PDSYGVX | – |
| Hermitian | – | PCHEGVX | – | PZHEGVX |

# Computational Routines Overview

1 ScaLAPACK Numerical Routines

ScaLAPACK provides a suite of **computational routines** for solving linear algebra problems, including:

- Solving systems of linear equations,
- Computing matrix factorizations,
- Estimating condition numbers,
- Refining solutions and computing error bounds,
- Computing matrix inverses,
- Performing matrix equilibration.

These routines are designed to work with distributed matrices and leverage parallel computing architectures for efficiency and scalability.

# Computational Routines for Linear Equations



**Linear System Notation:**

$$AX = B$$

where $A$ is the coefficient matrix, $B$ is the right-hand side, and $X$ is the solution.

ScaLAPACK provides computational routines for factorizing $A$ based on its properties:

- **General matrices**: $LU$ factorization with partial pivoting ($A = PLU$).
- **Symmetric/Hermitian positive definite**: Cholesky factorization ($A = U^H U$ or $A = LL^H$).
- **Band matrices**: Generalized band factorizations.
- **Tridiagonal matrices**: Specialized $LU$ or $LDL^H$ factorizations.

# Tasks Performed by Computational Routines



The last three characters of the routine name indicate the task:

- `PxyyTRF` **Factorize** the matrix (e.g., $LU$, Cholesky).
- `PxyyTRS` Use the factorization to **solve** $AX = B$ via forward/backward substitution.
- `PxyyCON` Estimate the reciprocal of the **condition number** $\kappa(A)$.
- `PxyyRFS` **Refine the solution** and compute error bounds (iterative refinement).
- `PxyyTRI` Compute the **matrix inverse** $A^{-1}$ using the factorization.
- `PxyyEQU` Compute **equilibration** scaling factors to improve condition number.

*Note:* Not all routines are available for all matrix types (e.g., inversion is not provided for band matrices as the inverse is generally dense).

# Summary of Computational Routines

1 ScaLAPACK Numerical Routines

| Matrix type | Factorize | Solve | Condition Estimate | Error Bounds | Invert | Equilibrate |
|---|---|---|---|---|---|---|
| General | PxGETRF | PxGETRS | PxGECON | PxGERFS | PxGETRI | PxGEEQU |
| General Band | PxGBTRF | PxGBTRS | PxGBCON | PxGBRFS | – | PxGBEQU |
| General Tridiagonal | PxDTTRF | PxDTTRS | – | – | – | – |
| Sym./Herm. Pos. Def. | PxPOTRF | PxPOTRS | PxPOCON | PxPORFS | PxPOTRI | PxPOEQU |
| Sym./Herm. Pos. Def. Band | PxPBTRF | PxPBTRS | PxPBCON | PxPBRFS | – | PxPBEQU |
| Sym./Herm. Pos. Def. Tridiagonal | PxPTTRF | PxPTTRS | – | – | – | – |
| Triangular | – | PxTRTRS | PxTRCON | PxTRRFS | PxTRTRI | – |

# Computational Routines for Orthogonal Factorization and LLS



ScaLAPACK provides routines for **orthogonal factorizations** and solving **linear least squares (LLS)** problems.

**Orthogonal Factorizations:**

- QR factorization ($A = QR$) for general matrices.
- LQ factorization ($A = LQ$) for general matrices.

**Linear Least Squares Problems:**

- Solve overdetermined systems ($m \geq n$) to find the least squares solution.
- Solve underdetermined systems ($m < n$) to find the minimum norm solution.

# QR Factorization



The most common of the factorizations is the **QR factorization** given by:

$$A = QR$$

where $R$ is $n \times n$ upper triangular and $Q$ is $m \times m$ orthogonal (or unitary). If $A$ is of full rank $n$, then $R$ is nonsingular.

It is often written as:

$$A = \begin{pmatrix} Q_1 & Q_2 \end{pmatrix} \begin{pmatrix} R_1 \\ 0 \end{pmatrix} = Q_1 R_1$$

where $Q_1$ consists of the first $n$ columns of $Q$, and $Q_2$ the remaining $m - n$ columns.
If $m < n$:

$$A = Q \begin{pmatrix} R_1 & R_2 \end{pmatrix}$$

where $R_1$ is upper triangular and $R_2$ is rectangular.

# ScaLAPACK support for QR



The routine `PxGEQRF` computes the QR factorization.

- $Q$ is not formed explicitly but represented as a **product of elementary reflectors** ($H_i = I - \tau v v^H$).
- `PxORGQR` (or `PxUNGQR`): Generates all or part of $Q$.
- `PxORMQR` (or `PxUNMQR`): Pre- or post-multiplies a matrix by $Q$ or $Q^H$.

# Orthogonal or Unitary Matrices in ScaLAPACK



ScaLAPACK represents an orthogonal (or unitary) matrix $Q$ as a product of **elementary reflectors** (Householder matrices):

$$Q = H_1 H_2 \cdots H_k$$

where each $H_i = I - \tau v v^H$.

- $\tau$ is a scalar and $v$ is the **Householder vector**.
- $v_1 = 1$, so it does not need to be stored.

**Working with $Q$:** Most users do not operate on $H_i$ directly but use provided routines:

- **Generate** $Q$: Routines ending in `-ORG` / `-UNG` (e.g., `PDORGQR`) form $Q$ explicitly.
- **Apply** $Q$: Routines ending in `-ORM` / `-UNM` (e.g., `PDORMQR`) compute $Q^T C$ or $QC$ without forming $Q$.

*Note:* In complex arithmetic, elementary reflectors are unitary but not Hermitian. This allows reducing complex Hermitian matrices to **real symmetric tridiagonal** form.

# Solving Linear Least Squares with QR



When $m \geq n$ and $A$ has full rank, the LLS problem minimizes $\|b - Ax\|_2$.

$$\|b - Ax\|_2 = \|Q^H b - Q^H Ax\|_2 = \left\| \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} - \begin{pmatrix} R_1 \\ 0 \end{pmatrix} x \right\|_2$$

**Steps:**

1. Compute $c = Q^H b$ using `PxORMQR`,
2. Solve the upper triangular system $R_1 x = c_1$ using `PxTRTRS`,
3. The residual vector is $r = Q \begin{pmatrix} 0 \\ c_2 \end{pmatrix}$, its norm is $\|c_2\|_2$.

# LQ Factorization



The **LQ factorization** is given by:

$$A = \begin{pmatrix} L & 0 \end{pmatrix} \begin{pmatrix} Q_1 \\ Q_2 \end{pmatrix} = LQ_1$$

where $L$ is $m \times m$ lower triangular, $Q$ is $n \times n$ orthogonal (or unitary), $Q_1$ consists of the first $m$ rows of $Q$, and $Q_2$ the remaining $n - m$ rows.

**ScaLAPACK Routines:**

- `PxGELQF`: Computes the factorization. $Q$ is represented as a product of elementary reflectors.
- `PxORGLQ` (or `PxUNGLQ`): Generates all or part of $Q$.
- `PxORMLQ` (or `PxUNMLQ`): Pre- or post-multiplies a matrix by $Q$ or $Q^H$.

The LQ factorization of $A$ corresponds to the QR factorization of $A^\top$ (or $A^H$):

$$A = LQ \iff A^\top = Q^\top L^\top$$

# Solving Underdetermined Systems with LQ



The LQ factorization is used to find the **minimum norm solution** of an **underdetermined** system ($m < n$) with rank $m$.
The solution is given by:

$$x = Q^H \begin{pmatrix} L^{-1}b \\ 0 \end{pmatrix}$$

**Computational Steps:**

1. Solve the lower triangular system $Ly = b$ for $y$ using `PxTRTRS`,
2. Form the vector $\tilde{y} = \begin{pmatrix} y \\ 0 \end{pmatrix}$,
3. Compute $x = Q^H \tilde{y}$ using `PxORMLQ` (or `PxUNMLQ`).

# QR Factorization with Column Pivoting



If $A$ is not of full rank, or the rank is in doubt, we can perform a **QR factorization with column pivoting**:

$$AP = QR$$

where $P$ is a permutation matrix.

- $P$ is chosen so that $|r_{11}| \geq |r_{22}| \geq \cdots \geq |r_{nn}|$.
- And for each $k$, the leading submatrix $R_{11}$ of size $k \times k$ is well-conditioned, while the trailing submatrix $R_{22}$ is negligible.

**Rank Determination:** If $R_{22}$ is negligible, then $k$ is the **effective rank** of $A$.

**Basic Solution to LLS:**

$$x = P \begin{pmatrix} R_{11}^{-1} c_1 \\ 0 \end{pmatrix}$$

where $c_1$ consists of the first $k$ elements of $c = Q^H b$.

# ScaLAPACK routine for QR with Pivoting



The routine `PxGEQPF` computes the QR factorization with column pivoting.

- It does not attempt to determine the rank of $A$ automatically (user must inspect diagonal of $R$).
- $Q$ is represented in the same way as in `PxGEQRF`.
- Routines `PxORMQR` (real) or `PxUNMQR` (complex) can be used to apply $Q$.

# Complete Orthogonal Factorization



QR with column pivoting does not compute a **minimum norm** solution for rank-deficient LLS unless $R_{12} = 0$.

To solve this, we apply further orthogonal transformations from the right to the upper trapezoidal matrix $R$ (using `PxTZRZF`) to eliminate $R_{12}$:

$$RP = \begin{pmatrix} T_{11} & 0 \\ 0 & 0 \end{pmatrix} Z$$

This yields the **complete orthogonal factorization**:

$$AP = Q \begin{pmatrix} T_{11} & 0 \\ 0 & 0 \end{pmatrix} Z$$

# Complete Orthogonal Factorization



From this, the minimum norm solution is obtained as:

$$x = PZ^H \begin{pmatrix} T_{11}^{-1} c_1 \\ 0 \end{pmatrix}$$

**Software Details:**

- $Z$ is represented as a product of elementary reflectors.
- `PxORMRZ` (or `PxUNMRZ`) is provided to multiple a matrix by $Z$ or $Z^H$.

# Summary of Orthogonal Factorization Routines (QR)



All the factorization routines discussed here (except PxTZRZF) allow arbitrary $m$ and $n$, so that in some cases the matrices $R$ or $L$ are trapezoidal rather than triangular.

A routine that performs **pivoting** is provided **only for the QR factorization**.

Computational routines for QR factorization

| Task | Single precision | | Double precision | |
|---|---|---|---|---|
| | Real | Complex | Real | Complex |
| **QR factorization and related operations** | | | | |
| Factorize generic | PSGEQRF | PCGEQRF | PDGEQRF | PZGEQRF |
| Factorize w/ pivoting | PSGEQPF | PCGEQPF | PDGEQPF | PZGEQPF |
| Apply $Q$ | PSORMQR | PCUNMQR | PDORMQR | PZUNMQR |
| Generate $Q$ | PSORGQR | PCUNGQR | PDORGQR | PZUNGQR |

# Summary of Orthogonal Factorization Routines (LQ)



Computational routines for LQ factorization

| Task | Single precision | | Double precision | |
|---|---|---|---|---|
| | Real | Complex | Real | Complex |
| **LQ factorization and related operations** | | | | |
| Factorize generic | PSGELQF | PCGELQF | PDGELQF | PZGELQF |
| Apply $Q$ | PSORMLQ | PCUNMLQ | PDORMLQ | PZUNMLQ |
| Generate $Q$ | PSORGLQ | PCUNGLQ | PDORGLQ | PZUNGLQ |

# Generalized QR (GQR) Factorization



The **Generalized QR (GQR) factorization** of an $n \times m$ matrix $A$ and an $n \times p$ matrix $B$ is given by the pair of factorizations:

$$A = QR, \quad B = QTZ$$

where $Q$ ($n \times n$) and $Z$ ($p \times p$) are orthogonal (or unitary) matrices. $R$ and $T$ are generally upper triangular (or trapezoidal) matrices.

## Implicit Factorization

If $B$ is square and nonsingular, the GQR factorization implicitly gives the QR factorization of $B^{-1}A$:

$$B^{-1}A = Z^H(T^{-1}R)$$

without explicitly computing the inverse or the matrix product.

# Generalized QR (GQR) Factorization



The **Generalized QR (GQR) factorization** of an $n \times m$ matrix $A$ and an $n \times p$ matrix $B$ is given by the pair of factorizations:

$$A = QR, \quad B = QTZ$$

where $Q$ ($n \times n$) and $Z$ ($p \times p$) are orthogonal (or unitary) matrices. $R$ and $T$ are generally upper triangular (or trapezoidal) matrices.

**ScaLAPACK Routine**: `PxGGQRF`

- Algorithms proceeds by computing the **QR factorization** of $A$ and then the **RQ factorization** of $Q^H B$.
- $Q$ and $Z$ can be formed explicitly or used to multiply other matrices (like standard QR).

# Generalized RQ (GRQ) Factorization



The **Generalized RQ (GRQ) factorization** of an $m \times n$ matrix $A$ and a $p \times n$ matrix $B$ is given by the pair of factorizations:

$$A = RQ, \quad B = ZTQ$$

where $Q$ ($n \times n$) and $Z$ ($p \times p$) are orthogonal (or unitary) matrices.

**Structure:**

- $R$ is upper trapezoidal (triangular) with structure similar to the $R$ in $RQ$.
- $T$ is upper trapezoidal (triangular) with structure similar to the $R$ in $QR$.

## Implicit Factorization

If $B$ is square and nonsingular, the GRQ factorization implicitly gives the RQ of $AB^{-1}$:

$$AB^{-1} = (RT^{-1})Z^H$$

without explicitly computing the inverse or the product.

# Generalized RQ (GRQ) Factorization



The **Generalized RQ (GRQ) factorization** of an $m \times n$ matrix $A$ and a $p \times n$ matrix $B$ is given by the pair of factorizations:

$$A = RQ, \quad B = ZTQ$$

where $Q$ ($n \times n$) and $Z$ ($p \times p$) are orthogonal (or unitary) matrices.

**Structure:**

- $R$ is upper trapezoidal (triangular) with structure similar to the $R$ in $RQ$.
- $T$ is upper trapezoidal (triangular) with structure similar to the $R$ in $QR$.

**ScaLAPACK Routine**: `PxGGRQF`

- Computes the **RQ factorization** of $A$ first, then the **QR factorization** of $BQ^H$.
- $Q$ and $Z$ can be formed explicitly or used to multiply other matrices.

# Computational Routines for Symmetric Eigenproblems



**Problem Definition**: Let $A$ be a real symmetric or complex Hermitian $N \times N$ matrix. Find eigenvalues $\lambda$ and non-zero eigenvectors $z$ such that:

$$Az = \lambda z$$

**Computation Stages**:

1. **Reduction to Tridiagonal Form**:
$$A = QTQ^H$$
where $Q$ is orthogonal (unitary) and $T$ is real symmetric tridiagonal.
2. **Solve Tridiagonal Problem**: Compute eigenvalues/vectors of $T$.
$$T = S\Lambda S^\top$$
The eigenvectors of $A$ are recovered as $Z = QS$.

# Reduction to Tridiagonal Form



The reduction $A = QTQ^H$ is performed by:

- `PxSYTRD`: Real symmetric matrices.
- `PxHETRD`: Complex Hermitian matrices.

**Handling $Q$:**

- The matrix $Q$ is represented as a product of elementary reflectors.
- `PxORMTR` (Real) / `PxUNMTR` (Complex): Multiplies a matrix by $Q$ without forming it explicitly.
- Used to transform eigenvectors of $T$ back to eigenvectors of $A$.

# Solving the Tridiagonal Problem



ScaLAPACK provides specific routines for the tridiagonal phase:

`xSTEQR2` Modified version of LAPACK's `xSTEQR`.

- Computes all eigenvalues and (optionally) eigenvectors using implicit QL or QR.
- Optimized look-ahead and partial updates for parallel execution.

`PxSTEBZ` Uses **bisection**.

- Computes some or all eigenvalues (e.g., in an interval $[a, b]$ or indices $i$ to $j$).
- Tunable accuracy vs. speed.

`PxSTEIN` Uses **inverse iteration**.

- Computes eigenvectors given accurate eigenvalues.
- Performs reorthogonalization to ensure vector quality (limited by workspace).

# Summary of Computational Routines for SEP



| Task | Single precision | | Double precision | |
|---|---|---|---|---|
| | Real | Complex | Real | Complex |
| **Tridiagonal reduction** | | | | |
| Factorize $A = QTQ^T$ | PSSYTRD | PCHETRD | PDSYTRD | PZHETRD |
| Multiply by $Q$ | PSORMTR | PCUNMTR | PDORMTR | PZUNMTR |
| **Tridiagonal solvers** | | | | |
| All eigs (QR/QL) | SSTEQR2 | CSTEQR2 | DSTEQR2 | ZSTEQR2 |
| Selected eigs (Bisection) | PSSTEBZ | PCSTEBZ | PDSTEBZ | PZSTEBZ |
| Selected vectors (Inv. It.) | PSSTEIN | PCSTEIN | PDSTEIN | PZSTEIN |

# Nonsymmetric Eigenproblems (NEP)



**Problem Definition**: Let $A$ be a square $n \times n$ matrix.

- A scalar $\lambda$ is an **eigenvalue** and a non-zero vector $v$ is a **right eigenvector** if:

$$Av = \lambda v$$

- A non-zero vector $u$ is a **left eigenvector** if:

$$u^H A = \lambda u^H$$

**Goal**: The basic task is to compute all $n$ eigenvalues $\lambda$ and, optionally, their associated right eigenvectors $v$ and/or left eigenvectors $u$.

# Schur Factorization



A fundamental step in solving NEP is the **Schur factorization**:

$$A = ZTZ^H \quad (\text{or } A = ZTZ^\top \text{ for real matrices})$$

- **Complex Case**: $Z$ is unitary, and $T$ is upper triangular. The eigenvalues appear on the diagonal of $T$.
- **Real Case**: $Z$ is orthogonal, and $T$ is **upper quasi-triangular**.
  - $T$ has 1-by-1 and 2-by-2 blocks on the diagonal.
  - Complex conjugate eigenvalues correspond to the 2-by-2 blocks.
- The columns of $Z$ are called the **Schur vectors** of $A$.

# Computational Stages for NEP



The computation is typically performed in two stages:

1. **Reduction to Upper Hessenberg Form**:

   $$A = QHQ^H$$

   where $H$ is **upper Hessenberg** (zero below the first subdiagonal) and $Q$ is orthogonal/unitary.

2. **Schur Factorization of $H$**:

   $$H = STS^H$$

   where $T$ is the Schur form. The Schur vectors of the original matrix $A$ are recovered as $Z = QS$.

# Stage 1: Reduction to Hessenberg Form



**Factorization Routine**: `PxGEHRD`

- Reduces a general matrix $A$ to upper Hessenberg form $H$.
- Represents the orthogonal matrix $Q$ in a factored form (product of elementary reflectors).

**Orthogonal/Unitary Matrix Operations**:

- `PxORMHR` (Real) / `PxUNMHR` (Complex).
- Used to multiply another matrix by $Q$ (or $Q^H$) without explicitly forming $Q$.

# Stage 2: Schur Factorization



**Routine**: `PxLAHQR`

- Computes the Schur factorization of the upper Hessenberg matrix $H$.
- Eigenvalues are obtained from the diagonal of $T$.

**Parallel Algorithm Strategy**:

- Unlike LAPACK's `xLAHQR` (single double shift) or `xHSEQR` (single large multi-shift), ScaLAPACK uses **multiple double shifts**.
- Shifts are spaced apart to allow parallelism across several processor rows/columns.
- Shifts are applied in a block fashion to maximize performance.

# Heuristics for Eigenvalue Computations: Shifting



The convergence of the QR algorithm (or similar iterative methods like the Francis double-shift) depends on the ratio of eigenvalues $|\lambda_i|/|\lambda_j|$.

**Goal**: Accelerate convergence by subtracting a scalar shift $\sigma$:

$$A - \sigma I = QR \quad \longrightarrow \quad A_{\text{new}} = RQ + \sigma I$$

Ideally, if $\sigma \approx \lambda_n$, the deflation happens very quickly.

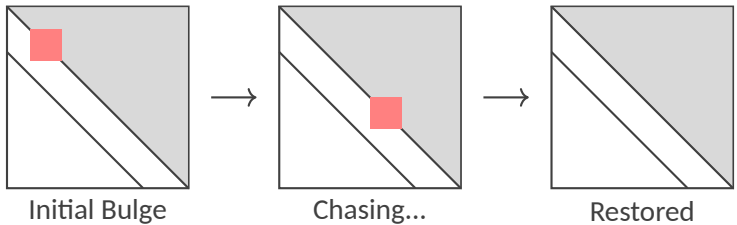
**Techniques**:

- **Francis Double Shift**: Standard for real non-symmetric matrices. Uses a $2 \times 2$ block from the bottom corner to perform implicitly shifted steps with complex conjugate shifts, keeping arithmetic real.
- **Aggressive Early Deflation**: Looks for convergence in a large window at the bottom of the matrix, deflating multiple eigenvalues at once.

# Parallelism and Bulge Chasing



The implicit shift strategy creates a “bulge” (non-zero entries outside the Hessenberg form) that must be chased down the diagonal to restore the form.




**Parallel Approach (ScaLAPACK)**:

- **Multi-shift**: Instead of chasing one bulge, introduce multiple bulges (chains of shifts) simultaneously.
- These bulges can be chased by different processors in a pipelined fashion, increasing the arithmetic intensity (BLAS 3) and parallelism.

# Summary of Computational Routines for NEP



| Task | Single precision | | Double precision | |
|---|---|---|---|---|
| | Real | Complex | Real | Complex |
| **Hessenberg reduction** | | | | |
| Factorize $A = QHQ^H$ | PSGEHRD | PCGEHRD | PDGEHRD | PZGEHRD |
| Multiply by $Q$ | PSORMHR | PCUNMHR | PDORMHR | PZUNMHR |
| **Schur factorization** | | | | |
| Compute $H = STS^H$ | PSLAHQR | PCLAHQR | PDLAHQR | PZLAHQR |

# Computational Stages for SVD



Let $A$ be an $m \times n$ matrix. The computation of the SVD proceeds in two main stages:

1. **Reduction to Bidiagonal Form**:
$$A = QBP^H$$
where $B$ is real bidiagonal, and $Q$ ($m \times m$) and $P$ ($n \times n$) are orthogonal (unitary).
   - If $m \geq n$, $B$ is upper bidiagonal.
   - If $m < n$, $B$ is lower bidiagonal.
2. **SVD of the Bidiagonal Matrix**:
$$B = U_1 \Sigma V_1^H$$
where $U_1$ and $V_1$ are orthogonal, and $\Sigma$ contains the singular values.

The singular vectors of $A$ are then $U = QU_1$ and $V = PV_1$.

# Computational Routines for SVD



**Reduction Routine**: `PxGEBRD`

- Reduces $A$ to bidiagonal form $B$.
- Represents $Q$ and $P$ as products of elementary reflectors.

**Applying $Q$ and $P$**:

- `PxORMBR` (Real) / `PxUNMBR` (Complex).
- Routine to multiply a given matrix by $Q$ or $P$ (or their transposes).

**Solving the Bidiagonal Problem**:

- ScaLAPACK typically utilizes the **LAPACK** routine `xBDSQR` to compute the SVD of the bidiagonal matrix $B$.

# Optimization for Non-Square Matrices



If $m \gg n$ or $n \gg m$, it is more efficient to perform a preliminary QR or LQ factorization.

**Case** $m \gg n$:

1. Compute QR factorization: $A = QR$ (using `PxGEQRF`).
2. Compute SVD of the $n \times n$ matrix $R$.

**Case** $n \gg m$:

1. Compute LQ factorization: $A = LQ$ (using `PxGELQF`).
2. Compute SVD of the $m \times m$ matrix $L$.

*Note:* The driver routine `PxGESVD` automatically handles these paths.

# Rank-Deficient Linear Least Squares



The SVD is used to solve rank-deficient LLS problems ($\min \|b - Ax\|_2$) by finding the **minimum norm solution**.

Let $k$ be the **effective rank** of $A$ (number of singular values $\sigma_i >$ threshold).
The solution is given by:

$$x = V_k \Sigma_k^{-1} c_1$$

where:

- $\Sigma_k$ is the leading $k \times k$ submatrix of $\Sigma$,
- $V_k$ consists of the first $k$ columns of $V$,
- $c_1$ consists of the first $k$ elements of $c = U^H b$.

`PxORMBR` (or `PxUNMBR`) is used to compute $U^H b$.

# Summary of SVD Computational Routines



| Task | Single precision | | Double precision | |
|---|---|---|---|---|
| | Real | Complex | Real | Complex |
| **Bidiagonal reduction** | | | | |
| Factorize $A = QBP^H$ | PSGEBRD | PCGEBRD | PDGEBRD | PZGEBRD |
| Multiply by $Q$ or $P$ | PSORMBR | PCUNMBR | PDORMBR | PZUNMBR |
| **Bidiagonal SVD (LAPACK)** | | | | |
| SVD of $B$ | SBDSQR | CBDSQR | DBDSQR | ZBDSQR |

# Computational Routines for GSEP

## 1 ScaLAPACK Numerical Routines

**Reduction to Standard Form**: The generalized problems are reduced to the standard symmetric eigenvalue problem $Cy = \lambda y$ using the Cholesky factorization of $B$ ($B = U^H U$ or $B = LL^H$).

**Reduction Strategy**:

- Type 1 ($Az = \lambda Bz$): $C = U^{-H} A U^{-1}$ or $L^{-1} A L^{-H}$. $z = U^{-1} y$ or $L^{-H} y$.
- Type 2 ($ABz = \lambda z$): $C = UAU^H$ or $L^H AL$. $z = U^{-1} y$ or $L^{-H} y$.
- Type 3 ($BAz = \lambda z$): $C = UAU^H$ or $L^H AL$. $z = U^H y$ or $Ly$.

**ScaLAPACK Routine**: `PxyyGST`

- Overwrites $A$ with the standard matrix $C$.
- After reduction, standard SEP routines (e.g., `PxSYTRD`) are used on $C$.

# Summary of Computational Routines for GSEP



| Task | Single precision | | Double precision | |
|---|---|---|---|---|
| | Real | Complex | Real | Complex |
| **Reduction to standard form** | | | | |
| Compute $C$ from $A, B$ | PSSYGST | PCHEGST | PDSYGST | PZHEGST |

**Note on Eigenvectors**: No special routines are needed to recover eigenvectors $z$ from $y$. These are simple triangular solves or matrix-vector multiplications handled by PBLAS (e.g., PxTRSV or PxTRMM).

# Conclusions



**ScaLAPACK** provides a comprehensive suite of routines for distributed-memory parallel computation of a wide range of linear algebra problems, including:

- Orthogonal factorizations (QR, LQ, GQR, GRQ).
- Symmetric and nonsymmetric eigenproblems (SEP, NEP, GSEP).
- Singular Value Decomposition (SVD).

The library leverages efficient algorithms and parallelism strategies to ensure scalability and performance on large-scale systems.

These tools permits to implement **high-performance applications** in scientific computing and engineering that require robust linear algebra capabilities, e.g., *model reduction*, *computation of matrix functions*, and *solving large-scale optimization problems*.

# High Performance Linear Algebra

## Lecture 15: Some hints on BLAS on GPUs

Ph.D. program in High Performance Scientific Computing

**Fabio Durastante** **Pasqua D'Ambra** **Salvatore Filippone**

Thursday 5, 2026 — 16.00:18.00

Dipartimento
di Matematica
Università di Pisa

# HPLA up to now and today's plan

1 GPU BLAS libraries

Up to now, we have seen:

- The design principles of the BLAS library,
- The implementation of the BLAS library on CPUs and shared memory systems,
- Some performance considerations on CPU BLAS implementations,
- The implementation of the BLAS library on distributed memory systems.
- Some performance considerations on distributed memory BLAS implementations.
- LAPACK and ScaLAPACK libraries for dense linear algebra.

Today we will look at some implementations of the BLAS library on GPUs.

# Table of Contents

2 GPGPU Computing and Dense Linear Algebra


- ▶ GPGPU Computing and Dense Linear Algebra
  - The CUDA Programming Model
  - The CUDA platform
    - A CUDA hello-world program
  - Implementing BLAS 1 with CUDA
  - The cuBLAS library
    - The AXPY operation
    - The 2-norm
    - The GEMV operation
    - The GEMM Operation
  - cuSOLVER for LAPACK-like workloads
  - cuSOLVERMp: Library for Distributed Dense Linear Algebra

# GPGPU Computing



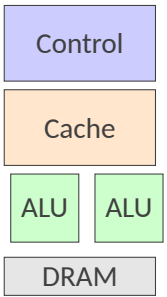
- **GPGPU** stands for **G**eneral-**P**urpose computing on **G**raphics **P**rocessing **U**nits.
- Originally designed for image rendering, GPUs have evolved into highly parallel, multi-threaded, many-core processors with tremendous computational power and very high memory bandwidth.
- Why leverage GPUs for scientific computing?
  - High arithmetic intensity.
  - Massive parallelism (thousands of cores).
  - Energy efficiency (FLOPS per Watt).
- Dense Linear Algebra (DLA) is a prime candidate for GPU acceleration due to its regular memory access patterns and high computational density.

# CPU vs. GPU Architecture



**CPU (Latency Oriented)**

- Optimized for serial performance.
- Complex control logic.
- Large caches to minimize latency.
- Fewer, powerful cores.



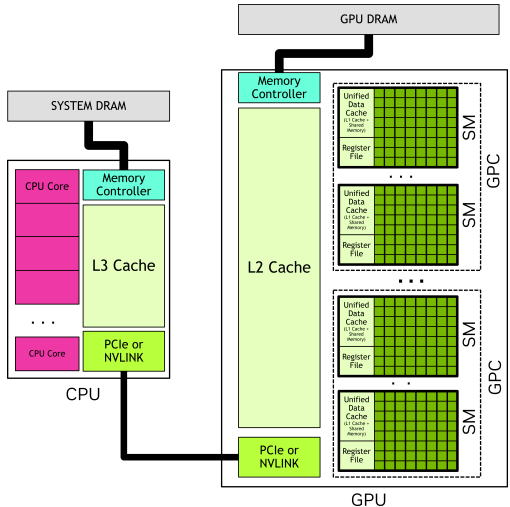

**GPU (Throughput Oriented)**

- Optimized for parallel performance.
- Simple control logic.
- Small caches (latency hidden by thread switching).
- Many simpler cores (SIMT).

# CPU vs. GPU Architecture



- **High-Level View**: GPU = collection of **Streaming Multiprocessors (SMs)**.
- **Organization**: SMs are grouped into Graphics Processing Clusters (GPCs).
- **Inside an SM**:
  - Register file.
  - Unified data cache (L1 cache + Shared Memory).
  - Functional units (CUDA cores, Tensor cores).
- **Flexibility**: The split between L1 and Shared Memory is configurable at runtime.



# Programming Models for GPUs



To utilize GPUs for linear algebra, we need specific programming models:

**CUDA (Compute Unified Device Architecture)** Proprietary NVIDIA platform. The de-facto standard for HPC on NVIDIA GPUs. Provides low-level control and high performance.

**HIP (Heterogeneous-Compute Interface for Portability)** AMD's answer to CUDA, allowing code to run on AMD hardware.

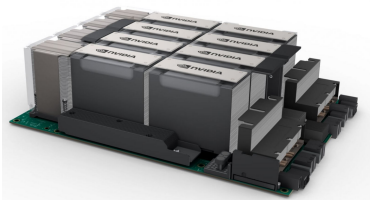
**OpenCL / SYCL / OneAPI** Open standards for cross-platform parallel programming.

**OpenACC / OpenMP** Directive-based approaches (pragmas) to offload computations to accelerators without rewriting the entire codebase.

Most vendor-provided BLAS libraries (cuBLAS, rocBLAS) are highly optimized using the native models (CUDA/HIP).

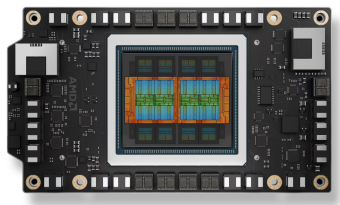
# How much does it costs?





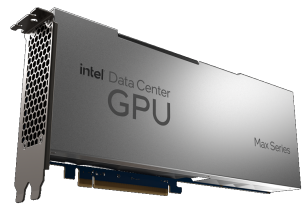

NVIDIA HGX200

- 141GB of HBM3e
- 4.8TB/s of bandwidth
- 4 petaFLOPS for FP8
- $ ∼ 300.000 $




AMD Instinct™ MI355X GPUs

- 288GB of HBM3e
- 8 TB/s of bandwidth
- 5 petaFLOPS for FP8
- $ ∼220.000 €




Intel® GPU Max 1550

- 128 GB of HBM2e
- 3.28 TB/s of bandwidth
- 52.43 TFLOPS for FP8
- $ ∼8.550 €



The CUDA programming model assumes a heterogeneous system:

- **Host**: The CPU and its memory (system memory).
- **Device**: The GPU and its own memory.

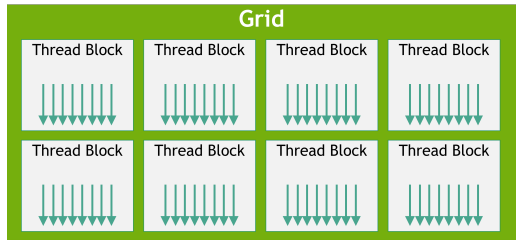
Typically, a CUDA program flows as follows:

1. **Copy data** from Host memory to Device memory.
2. **Launch Kernel**: The CPU instructs the GPU to execute a function (kernel) on the data.
3. **Execute**: The GPU executes the kernel in parallel across many threads.
4. **Copy results** from Device memory back to Host memory.

# Thread blocks and grids



- **High count:** Kernels launch millions of threads, organized into **Blocks**.
- **Structure:** blocks form a **Grid**.
  - Blocks in a grid have the same size.
  - Can be 1D, 2D, or 3D for easy mapping to data.
- **Identity:** Threads use built-in variables to find their coordinates in the Block/Grid.



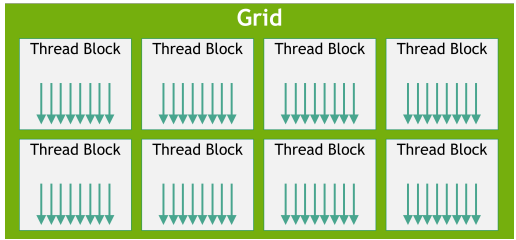

The CUDA programming model enables arbitrarily large grids to run on GPUs of any size. To achieve this, it requires that there be **no data dependencies** between threads in different thread blocks.

# Thread blocks and grids



- **Hardware Mapping:**
  - A block executes on a single **SM**.
  - Enables fast synch and **Shared Memory** usage within a block.
  - Grids scale to millions of blocks, automatically scheduled on available SMs.

Threads within a block run on the same SM, while different thread blocks are scheduled among available SMs in any order. This allows the execution to be parallel or serial, ensuring scalability across different hardware.

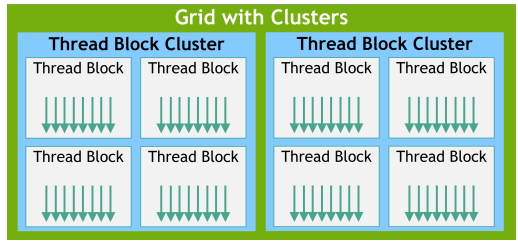



The CUDA programming model enables arbitrarily large grids to run on GPUs of any size. To achieve this, it requires that there be **no data dependencies** between threads in different thread blocks.

# Thread blocks and grids



- **Clusters (CC $\geq$ 9.0):** An optional hierarchy level grouping thread blocks.
- **Execution:** Guaranteed to run on a single **GPC** (Graphics Processing Cluster).
- **Benefits:**
  - Synchronization between blocks in the same cluster.
  - **Distributed Shared Memory:** Threads can access the shared memory of all blocks in the cluster.




The CUDA programming model enables arbitrarily large grids to run on GPUs of any size. To achieve this, it requires that there be **no data dependencies** between threads in different thread blocks.

# Warps and SIMT



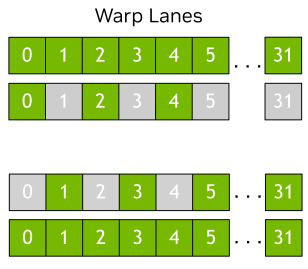
- **Warps**:
  - Threads within a block are grouped into bundles of **32 threads** called *warps*.
  - Warps are the fundamental unit of scheduling on an SM.
- **SIMT (Single-Instruction, Multiple-Threads)**:
  - All threads in a warp execute the **same instruction** at the same time.
  - Each thread has its own instruction address counter and register state.
  - *Lock-step execution*: Ideally, all 32 threads progress together.
- **Recommendation**:
  - Configure thread block sizes to be multiples of 32.
  - If not, the last warp will have inactive lanes, wasting computational resources.

# Warp Divergence



- Threads in a warp can follow different execution paths (e.g., `if-else` statements).
- **Divergence**: If threads diverge, the warp serially executes each branch path.
- Threads not on the current path are **masked off** (inactive).
- **Impact**: Significantly reduces parallel efficiency (active threads < 32).
- **Optimization**: Maximize utilization by ensuring threads in a warp follow the same control flow.

```
if(threadIdx.x%2 == 0)
   {a = r(t); }
else
   {a = q(t); }
y = f(a);
```



# GPU Memory Model Overview



Modern GPUs utilize a complex memory hierarchy to balance bandwidth and latency. We distinguish between:

- **Off-Chip Memory (DRAM)**:
  - **Global Memory**: The large DRAM attached to the GPU. Accessible by all SMs. High latency, high bandwidth.
  - **System Memory**: DRAM attached to the CPU (Host).
- **On-Chip Memory**:
  - **Registers**: Fastest memory, private to a thread.
  - **Shared Memory**: Programmable cache shared within a Thread Block (or Cluster).
  - **Caches**: L1 (per SM) and L2 (device-wide).

**Unified Addressing**: CPU and GPU share a single virtual memory space, allowing the unique identification of memory locations across devices.

# On-Chip Memory Details

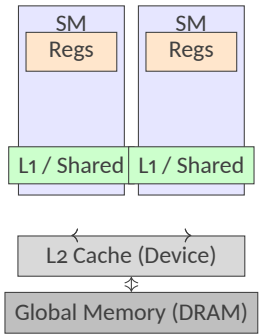


**Registers**

- Stores thread-local variables.
- Allocation is per-thread; total usage determines *occupancy* (how many blocks fit on an SM).

**Shared Memory**

- Visible to all threads in a Block (or Cluster on H100+).
- Used for inter-thread communication and data reuse.
- Physically shares storage with L1 Cache; the split is often configurable.



# The CUDA Platform Overview



The NVIDIA CUDA platform enables heterogeneous computing through a combination of hardware and software components.

**Key Components:**

- **Compute Capability (CC):**
  - Version number (X.Y) indicating supported features and hardware parameters.
  - Corresponds to the Streaming Multiprocessor (SM) version (e.g., CC 9.0 $\rightarrow$ `sm_90`).
- **NVIDIA Driver:**
  - acts as the ”OS” of the GPU.
  - Foundational; required for all GPU uses (CUDA, Vulkan, Direct3D).
- **CUDA Toolkit:**
  - Suite of libraries, headers, tools (e.g., `nvcc`), and the **CUDA Runtime**.

# CUDA APIs: Runtime vs. Driver



CUDA offers two main APIs for application development:

**CUDA Runtime API**

- High-level API.
- Handles common tasks (memory allocation, data transfer, kernel launching) easily.
- Language extensions (e.g., <<<...>>> syntax).
- Used in most CUDA applications.

**CUDA Driver API**

- Low-level API exposed directly by the driver.
- Grants finer control (e.g., context management).
- More verbose; conceptually similar to OpenCL.

- The Runtime API is implemented *on top* of the Driver API.
- Applications can mix both APIs (interoperability).

# Parallel Thread Execution (PTX)



- **What is PTX?**
  - A virtual Instruction Set Architecture (ISA).
  - An intermediate assembly language that abstracts the physical hardware.
- **Role in Compilation:**
  - High-level code (C++/Fortran) is compiled into PTX.
  - The graphics driver *Just-In-Time (JIT)* compiles PTX into machine code (SASS) valid for the specific GPU installed.
- **Benefit:**
  - *Forward compatibility*: Code compiled to PTX years ago can often run on new GPUs because the driver translates the virtual instructions to the new architecture.

# Binary Compatibility



NVIDIA GPUs guarantee binary compatibility under specific conditions:

- **Within Major Version:**
  - Binary code (`cubin`, e.g., `sm_86`) can run on GPUs with the *same major version* and equal or higher minor version.
  - *Example:* Code compiled for `sm_86` works on `sm_86` and `sm_89`, but **not** on `sm_80` (minor version too low).
- **Between Major Versions:**
  - Binaries are **not** compatible across major versions.
  - *Example:* Code for `sm_80` will not run on a Hopper GPU (`sm_90`).

**Note:** Binary compatibility applies only to binaries generated by official NVIDIA tools (e.g., `nvcc`).

# PTX Compatibility & Forward Compatibility



To support future hardware, GPU code can be embedded as PTX (virtual assembly) rather than just binary SASS.

- **Mechanism:**
  - Application stores PTX for a specific virtual architecture (e.g., `compute_80`).
  - At runtime, the driver **Just-In-Time (JIT)** compiles this PTX into binary code for the detected GPU.
- **Requirement:** The PTX version must be $\leq$ the GPU's compute capability.
- **Benefit:**
  - *Forward Compatibility*: An app compiled today for `compute_80` can run on a future architecture (e.g., `sm_120`) without recompilation.

# Just-in-Time (JIT) Compilation



- **Process**: The device driver translates loaded PTX into native binary code at application load time.
- **Trade-offs**:
  - 👎 Increases application load / startup time.
  - 👍 Allows code to run on GPUs that didn't exist when the app was built.
  - 👍 Benefits from newer compiler optimizations in updated drivers.
- **Compute Cache**:
  - The driver caches generated binaries to avoid recompiling on subsequent runs.
  - Cache is invalidated upon driver updates.
- **NVRTC**: A runtime compilation library that allows compiling CUDA C++ source code directly to PTX at runtime, offering even more flexibility.

# A Simple CUDA hell-world Program



```cpp
#include <iostream>
__global__ void helloFromGPU() {
    printf("Hello World from GPU!\n");
}
int main() {
    // Launch kernel with 1 block of 1 thread
    helloFromGPU<<<1, 1>>>();
    // Wait for GPU to finish
    cudaDeviceSynchronize();
    return 0;
}
```

- The `__global__` qualifier indicates a kernel function executed on the GPU.
- The `<<<1, 1>>>` syntax launches the kernel with 1 block of 1 thread.
- `cudaDeviceSynchronize()` ensures the CPU waits for the GPU before exiting.

# Compilation



To **compile** the CUDA program, use the NVIDIA CUDA Compiler `nvcc`:

```
nvcc -o hello_cuda hello_cuda.cu
```

This command generates an executable named `hello_cuda`.

We can specify the target GPU architecture using the `-arch` flag:

```
nvcc -arch=sm_89 -o hello_cuda hello_cuda.cu
```

This compiles the code for GPUs with Compute Capability 8.9 (The NVIDIA GeForce RTX 4060 on my laptop).

If we run the program, we should see:

```
Hello World from GPU!
```

# Compilation with CMake



To compile CUDA code with CMake, we need to enable CUDA support in our `CMakeLists.txt` file:

```
cmake_minimum_required(VERSION 3.18)
project(HelloCUDA LANGUAGES CXX CUDA)
add_executable(hello_cuda hello_cuda.cu)
set_target_properties(hello_cuda PROPERTIES
    CUDA_ARCHITECTURES "89"
)
```

- CUDA is treated as a first-class language in CMake.
- The `set_target_properties(target CUDA_ARCHITECTURES)` property specifies the target GPU architectures.

# Implementing DAXPY with CUDA



The DAXPY operation computes $\mathbf{y} \leftarrow \alpha\mathbf{x} + \mathbf{y}$ for vectors $\mathbf{x}$ and $\mathbf{y}$ and scalar $\alpha$. Here is a simple CUDA implementation:

```
__global__ void daxpy(int n, double alpha, const double *x, double *y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        y[i] += alpha * x[i];
    }
}
```

- Each thread computes one element of the result.
- The thread index $i$ is calculated using block and thread indices.
- A boundary check ensures we do not access out-of-bounds memory.

# Block and Thread Indexing



In the DAXPY kernel, each thread computes a single element of the output vector $\mathbf{y}$. The thread index $i$ is calculated as:

$$i = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}$$

**blockIdx.x** The index of the current block in the grid.

**blockDim.x** The number of threads per block.

**threadIdx.x** The index of the thread within its block.

This calculation allows us to **uniquely identify each thread** across the entire grid, enabling parallel computation of the DAXPY operation.

# Launching the DAXPY Kernel



To launch the DAXPY kernel from the host (CPU) code, we need to allocate memory on the GPU, copy data, and invoke the kernel:

```
// Host code
int n = 1<<20; // Vector size
double alpha = 2.0;
double *h_x = (double*)malloc(n * sizeof(double));
double *h_y = (double*)malloc(n * sizeof(double));
// Initialize h_x and h_y...
double *d_x, *d_y;
cudaMalloc(&d_x, n * sizeof(double));
cudaMalloc(&d_y, n * sizeof(double));
cudaMemcpy(d_x, h_x, n * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, h_y, n * sizeof(double), cudaMemcpyHostToDevice);
```

# Launching the DAXPY Kernel



```
// Launch kernel
int blockSize = 256;
int numBlocks = (n + blockSize - 1) / blockSize;
daxpy<<<numBlocks, blockSize>>>(n, alpha, d_x, d_y);
// Copy result back to host
cudaMemcpy(h_y, d_y, n * sizeof(double), cudaMemcpyDeviceToHost);
// Free device memory
cudaFree(d_x);
cudaFree(d_y);
```

- We allocate device memory using `cudaMalloc` and copy data with `cudaMemcpy`.
- The kernel is launched with a calculated number of blocks and threads per block.
- Finally, we copy the result back to the host and free device memory.

# Executing a Kernel



We execute a kernel on the GPU using the triple angle bracket syntax <<<...>>>:

```
daxpy<<<numBlocks, blockSize>>>(n, alpha, d_x, d_y);
```

Here:

- `numBlocks` specifies the number of thread blocks in the grid.
- `blockSize` specifies the number of threads per block.

**Example Calculation**:

- The `int n=1<<20;` sets the vector size to $2^{20} = 1,048,576$.
- For a vector of size $n = 1,000,000$ and a block size of 256:
- Number of blocks = $\lceil \frac{1,000,000}{256} \rceil = 3907$.

# What performances do we get out of this?



- The performance of our simple DAXPY implementation will depend on several factors:

  **Memory bandwidth** The speed of data transfer between global memory and the GPU cores.

  **Kernel launch overhead** The time taken to launch the kernel on the GPU.

  **Occupancy** How well the GPU's resources are utilized (number of active warps per SM).

- To measure performance, we can use CUDA events to time the kernel execution and calculate the achieved GFLOPS.
- Optimizations such as using shared memory, minimizing global memory accesses, and ensuring coalesced memory access patterns can significantly improve performance.

# CUDA Events for Timing



An approach to measure kernel execution time is to use **CUDA Events**:

- Create events: `cudaEventCreate`
- Record events: `cudaEventRecord`
- Synchronize: `cudaEventSynchronize`
- Elapsed time: `cudaEventElapsedTime`

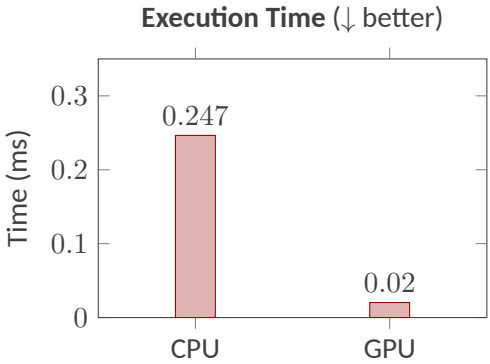
```
cudaEvent_t start, stop;
float ms;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);
// Launch kernel
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&ms, start, stop);
```

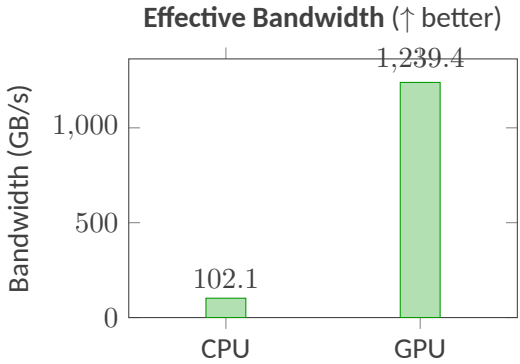
To calculate GFLOPS:

$$\text{GFLOPS} = \frac{2n}{ms \times 10^6}$$

# These are the results on my laptop (RTX 4060)






**Speedup:** 12.14 x

Vector size: $N = 2^{20}$ elements (8.0 MB)

# The cuBLAS Library



- **cuBLAS** is NVIDIA's GPU-accelerated implementation of the Basic Linear Algebra Subprograms (BLAS) library.
- It provides highly optimized routines for dense linear algebra operations, including:
  - Level 1 BLAS: Vector operations (e.g., DAXPY, DOT).
  - Level 2 BLAS: Matrix-vector operations (e.g., GEMV).
  - Level 3 BLAS: Matrix-matrix operations (e.g., GEMM).
- cuBLAS leverages the full capabilities of NVIDIA GPUs, including:
  - Efficient memory access patterns.
  - Use of shared memory and registers.
  - Optimized kernel launches and execution strategies.
- It is widely used in scientific computing, machine learning, and other high-performance applications requiring fast linear algebra computations.

# Using cuBLAS



To ensure maximum compatibility with existing Fortran environments, the cuBLAS library operates based on Column-Major storage and 1-based indexing.

**Implications for C/C++ Developers**:

- C and C++ utilize **Row-Major** storage by default.
- Consequently, native 2D array semantics (e.g., `double A[rows][cols]`) cannot be directly used with cuBLAS.

You can define two macros to help with indexing:

```
#define IDX2F(i,j,ld) ((((j)-1)*(ld))+((i)-1))
#define IDX2C(i,j,ld) (((j)*(ld))+(i))
```

# A couple of useful macros



Writing **error check code** is very repetitive. We can define a couple of macros to help us:

1) To wrap *CUDA runtime* API calls:

```
#define CHECK_CUDA(call)                                                    \
    do {                                                                    \
        cudaError_t err = call;                                             \
        if (err != cudaSuccess) {                                           \
            fprintf(stderr, "CUDA error %s:%d: %s\n",                       \
                    __FILE__, __LINE__, cudaGetErrorString(err));           \
            exit(EXIT_FAILURE);                                             \
        }                                                                   \
    } while (0)
```

This macro checks the return status of a CUDA runtime API call and prints an error message if the call fails.

# A couple of useful macros



Writing **error check code** is very repetitive. We can define a couple of macros to help us:

2) To wrap *cuBLAS runtime* API calls:

```
#define CHECK_CUBLAS(call)                                          \
    do {                                                            \
        cublasStatus_t status = call;                               \
        if (status != CUBLAS_STATUS_SUCCESS) {                      \
            fprintf(stderr, "cuBLAS error %s:%d: %d\n",             \
                    __FILE__, __LINE__, status);                    \
            exit(EXIT_FAILURE);                                     \
        }                                                           \
    } while (0)
```

This macro checks the return status of a cuBLAS API call and prints an error message if the call fails.

# cuBLAS example: DAXPY



To use cuBLAS the first thing we need to do is create a **cuBLAS handle**:

```
cublasHandle_t handle;
CHECK_CUBLAS(cublasCreate(&handle));
```

This handle is used to manage the cuBLAS library context and resources.

Then we can call the `cublasDaxpy` function to perform the DAXPY operation:

```
CHECK_CUBLAS(cublasDaxpy(handle, n, &alpha, d_x, 1, d_y, 1));
```

At this point an **interface** to the DAXPY operation should be **very familiar**!

Finally, we need to destroy the cuBLAS handle to free resources:

```
cublasDestroy(handle);
```

# The cuBLAS DAXPY interface



```
cublasStatus_t cublasDaxpy(cublasHandle_t handle, int n,
                           const double          *alpha,
                           const double          *x, int incx,
                           double                *y, int incy);
```

**handle** cuBLAS library context.

**n** Number of elements in vectors `x` and `y`.

**alpha** Pointer to the scalar multiplier.

**x** Pointer to the input vector $\mathbf{x}$.

**incx** Stride between elements in `x` (usually 1).

**y** Pointer to the input/output vector $\mathbf{y}$.

**incy** Stride between elements in `y` (usually 1).

# The cuBLAS DAXPY interface



```
cublasStatus_t cublasDaxpy(cublasHandle_t handle, int n,
                           const double          *alpha,
                           const double          *x, int incx,
                           double                *y, int incy);
```

**alpha** Pointer to the scalar multiplier.

`alpha` and `beta` parameters can be *passed by reference* on the host or the device.
When the pointer mode is set to `CUBLAS_POINTER_MODE_HOST`:

- The scalars can be on the stack or heap (not managed memory).
- The kernels are launched with the *value* of the scalar.
- Host memory can be freed immediately after the call.

# The cuBLAS DAXPY interface



```
cublasStatus_t cublasDaxpy(cublasHandle_t handle, int n,
                           const double          *alpha,
                           const double          *x, int incx,
                           double                *y, int incy);
```

**alpha** Pointer to the scalar multiplier.

`alpha` and `beta` parameters can be *passed by reference* on the host or the device.
When set to `CUBLAS_POINTER_MODE_DEVICE`:

- The scalars must be accessible on the device.
- Their values must not change until the kernel completes.
- Allows fully asynchronous execution, even if `alpha` is generated by a previous kernel (common in iterative solvers).

# The cuBLAS DAXPY interface



```
cublasStatus_t cublasDaxpy(cublasHandle_t handle, int n,
                           const double          *alpha,
                           const double          *x, int incx,
                           double                *y, int incy);
```

**alpha** Pointer to the scalar multiplier.

`alpha` and `beta` parameters can be *passed by reference* on the host or the device.
To **set the pointer mode**, use:

```
cublasSetPointerMode(handle, CUBLAS_POINTER_MODE_HOST);
```

The *default mode* is `CUBLAS_POINTER_MODE_HOST`.

# The cuBLAS 2-norm



The cuBLAS library provides the `cublasDnrm2` function to compute the Euclidean norm (2-norm) of a vector:

```
cublasStatus_t  cublasDnrm2(cublasHandle_t handle, int n,
                            const double *x, int incx, double *result)
```

**handle** cuBLAS library context.

**n** Number of elements in vector `x`.

**x** Pointer to the input vector $\mathbf{x}$.

**incx** Stride between elements in `x` (usually 1).

**result** Pointer to store the computed norm.

# The cuBLAS 2-norm



The cuBLAS library provides the `cublasDnrm2` function to compute the Euclidean norm (2-norm) of a vector:

```
cublasStatus_t  cublasDnrm2(cublasHandle_t handle, int n,
                            const double *x, int incx, double *result)
```

**result** Pointer to store the computed norm.

For the functions of this category, when the pointer mode is set to `CUBLAS_POINTER_MODE_HOST`, these functions block the CPU, until the GPU has completed its computation and the results have been copied back to the Host.

# The cuBLAS 2-norm



The cuBLAS library provides the `cublasDnrm2` function to compute the Euclidean norm (2-norm) of a vector:

```
cublasStatus_t  cublasDnrm2(cublasHandle_t handle, int n,
                            const double *x, int incx, double *result)
```

**result** Pointer to store the computed norm.

When the pointer mode is set to `CUBLAS_POINTER_MODE_DEVICE`, these functions return immediately. This requires proper synchronization in order to **read the result from the host**.

# The cuBLAS GEMV operation



The cuBLAS library provides the `cublasDgemv` function to perform the matrix-vector multiplication operation:

$$\mathbf{y} \leftarrow \alpha\,\mathrm{op}(\mathbf{A})\mathbf{x} + \beta\mathbf{y}$$

The function signature is:

```
cublasStatus_t cublasDgemv(cublasHandle_t handle,
                           cublasOperation_t trans, int m, int n,
                           const double           *alpha,
                           const double           *A, int lda,
                           const double           *x, int incx,
                           const double           *beta,
                           double *y, int incy)
```

# The cuBLAS GEMV operation



| | |
|---|---|
| handle | cuBLAS library context. |
| trans | Operation on matrix `A` (`CUBLAS_OP_N`, `CUBLAS_OP_T`, `CUBLAS_OP_C`). |
| m,n | Dimensions of matrix `A` (rows, columns). |
| alpha | Pointer to the scalar multiplier for $\mathrm{op}(\mathbf{A})\mathbf{x}$. |
| A | Pointer to the input matrix $\mathbf{A}$. |
| lda | Leading dimension of matrix `A` (usually `m`). |
| x | Pointer to the input vector $\mathbf{x}$. |
| incx | Stride between elements in `x` (usually 1). |
| beta | Pointer to the scalar multiplier for $\mathbf{y}$. |
| y | Pointer to the input/output vector $\mathbf{y}$. |
| incy | Stride between elements in `y` (usually 1). |

# We can run a comparison with OpenBLAS



We can create and populate matrices/vectors on the host,

```
const size_t bytes_A = (size_t)m * n * sizeof(double);
const size_t bytes_x = (size_t)n * sizeof(double);
const size_t bytes_y = (size_t)m * sizeof(double);
double *h_A = (double *)malloc(bytes_A);
double *h_x = (double *)malloc(bytes_x);
double *h_y = (double *)malloc(bytes_y);
double *h_y_cpu = (double *)malloc(bytes_y);
double *h_y_gpu = (double *)malloc(bytes_y);
```

# We can run a comparison with OpenBLAS



We can create and populate matrices/vectors on the host,

```
for (int col = 0; col < n; col++) {
  for (int row = 0; row < m; row++) {
    h_A[col * m + row] = 1.0 + (row + col) * 1e-6;
  }
}
for (int i = 0; i < n; i++) {
  h_x[i] = 1.0;
}
for (int i = 0; i < m; i++) {
  h_y[i] = 2.0;
  h_y_cpu[i] = 2.0;
}
```

# We can run a comparison with OpenBLAS



We can create and populate matrices/vectors on the host, copy them to the device,

```
double *d_A, *d_x, *d_y;
// Allocate device memory
CHECK_CUDA(cudaMalloc(&d_A, bytes_A));
CHECK_CUDA(cudaMalloc(&d_x, bytes_x));
CHECK_CUDA(cudaMalloc(&d_y, bytes_y));
// Copy data to device
CHECK_CUDA(cudaMemcpy(d_A, h_A, bytes_A, cudaMemcpyHostToDevice));
CHECK_CUDA(cudaMemcpy(d_x, h_x, bytes_x, cudaMemcpyHostToDevice));
CHECK_CUDA(cudaMemcpy(d_y, h_y, bytes_y, cudaMemcpyHostToDevice));
```

# We can run a comparison with OpenBLAS



We can create and populate matrices/vectors on the host, copy them to the device, and run the `cublasDgemv` function

```
int m = 4096;
int n = 4096;
const double alpha = 2.0;
const double beta = 1.0;
CHECK_CUBLAS(cublasDgemv(handle, CUBLAS_OP_N, m, n, &alpha, d_A,
  m, d_x, 1,
  &beta, d_y, 1));
```

- We run 10 warm-up iterations before timing the execution.
- We then run 100 timed iterations to measure performance.
- Finally, we copy the result back to the host for verification.

# We can run a comparison with OpenBLAS



We copy back the result to the host and verify correctness against OpenBLAS:

```
CHECK_CUDA(cudaMemcpy(h_y_gpu, d_y, bytes_y, cudaMemcpyDeviceToHost));
```

# We can run a comparison with OpenBLAS



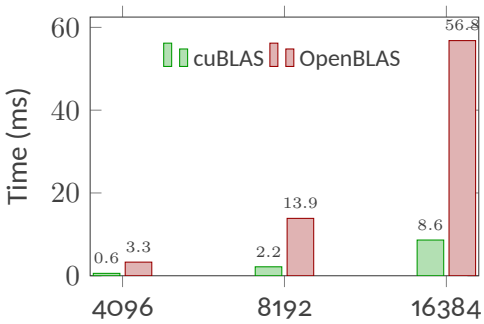
We copy back the result to the host and verify correctness against OpenBLAS:

```
CHECK_CUDA(cudaMemcpy(h_y_gpu, d_y, bytes_y, cudaMemcpyDeviceToHost));
cblas_dgemv(CblasColMajor, CblasNoTrans,m, n, alpha, h_A, m, h_x, 1, beta,
↪  h_y_cpu, 1);
// Verify correctness
int errors = 0;
for (int i = 0; i < m && errors < 10; i++) {
  double diff = fabs(h_y_cpu[i] - h_y_gpu[i]);
  if (diff > 1e-8) {
    fprintf(stderr, "Mismatch at %d: CPU %.12f GPU %.12f\n",
        i, h_y_cpu[i], h_y_gpu[i]);
    errors++;
  }
}
```

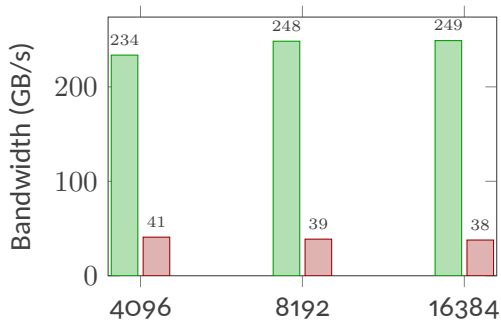
# Comparison with OpenBLAS (NVIDIA RTX 4060 GPU)



**Execution Time** (↓ better)




**Effective Bandwidth** (↑ better)




Speedup of roughly **6.5x**, saturating the GPU memory bandwidth at $\sim 249\,\mathrm{GB\,s^{-1}}$.

# The cuBLAS GEMM operation



The cuBLAS library provides the `cublasDgemm` function to perform the matrix-matrix multiplication operation:

$$\mathbf{C} \leftarrow \alpha\,\mathrm{op}(\mathbf{A})\,\mathrm{op}(\mathbf{B}) + \beta\mathbf{C}$$

The function signature is:

```
cublasStatus_t cublasDgemm(cublasHandle_t handle,
                           cublasOperation_t transa,
                           cublasOperation_t transb,
                           int m, int n, int k,
                           const double          *alpha,
                           const double          *A, int lda,
                           const double          *B, int ldb,
                           const double          *beta,
                           double          *C, int ldc)
```

# The cuBLAS GEMM operation



| | |
|---|---|
| handle | cuBLAS library context. |
| transa, transb | Operations on matrices `A` and `B` (`CUBLAS_OP_N`, `CUBLAS_OP_T`, `CUBLAS_OP_C`). |
| m,n,k | Dimensions of the matrices. |
| alpha | Pointer to the scalar multiplier for $\mathrm{op}(\mathbf{A})\ \mathrm{op}(\mathbf{B})$. |
| A | Pointer to the input matrix $\mathbf{A}$. |
| lda | Leading dimension of matrix `A` (usually `m`). |
| B | Pointer to the input matrix $\mathbf{B}$. |
| ldb | Leading dimension of matrix `B` (usually `k`). |
| beta | Pointer to the scalar multiplier for $\mathbf{C}$. |
| C | Pointer to the input/output matrix $\mathbf{C}$. |
| ldc | Leading dimension of matrix `C` (usually `m`). |

# Running a comparison with OpenBLAS



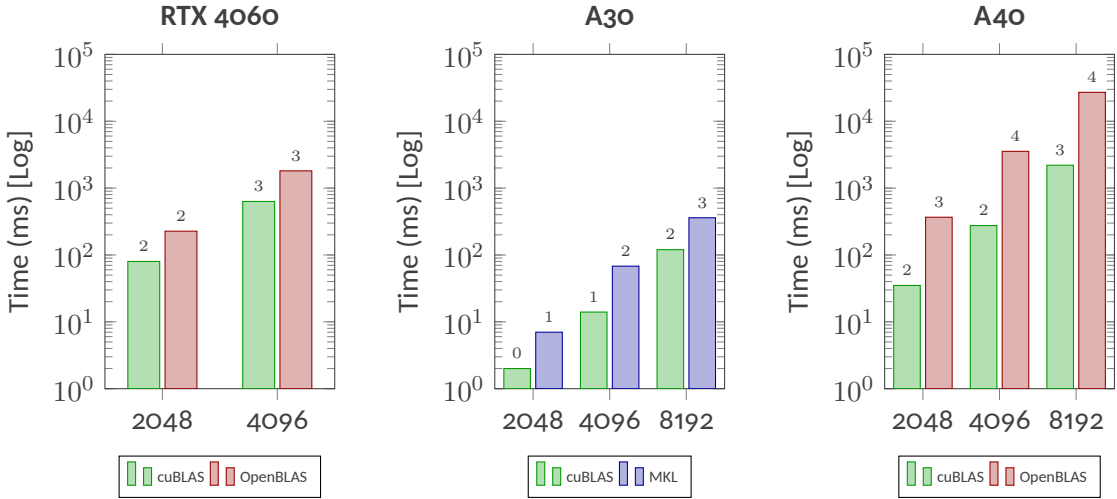
We can run a **similar comparison** as before, but this time using the `cublasDgemm` function to perform matrix-matrix multiplication.

The process involves:

1. Creating and populating matrices on the host.
2. Copying them to the device.
3. Running the `cublasDgemm` function.
4. Copying back the result to the host.
5. Verifying correctness against OpenBLAS.

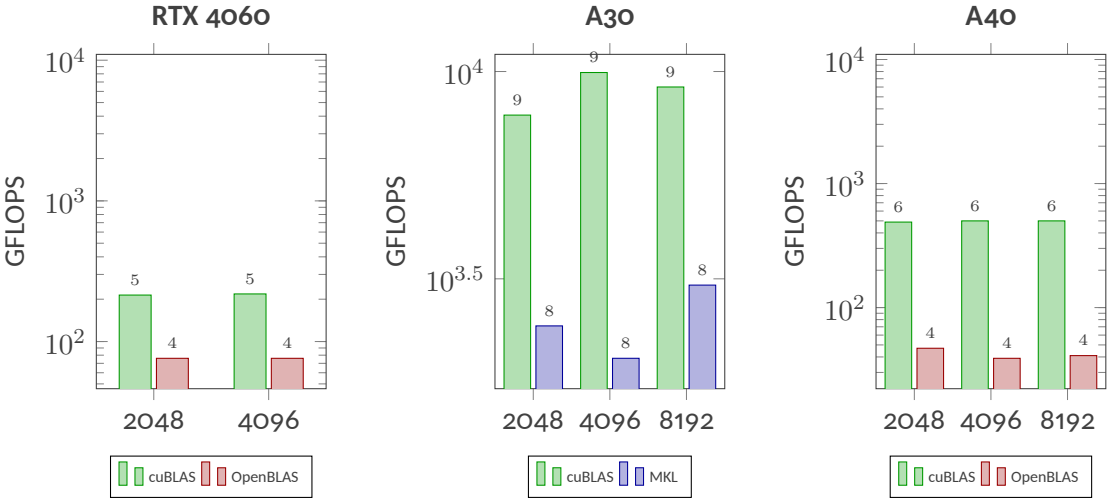
# DGEMM Performance: Throughput





# DGEMM Performance: Throughput





# Interpreting DGEMM Performance Results



- **A30: Near-Peak FP64 Performance**
  - Based on GA100 architecture with **FP64 Tensor Cores**.
  - cuBLAS DGEMM exploits Tensor Cores when matrix sizes and alignment allow.
  - Measured performance ($\sim$8–10 TFLOPS) reaches **80–90% of theoretical FP64 Tensor Core peak**.
  - Low memory bandwidth utilization confirms the kernel is **compute-bound**.
- **A40: Limited by Scalar FP64 Units**
  - Based on GA102 architecture with **no FP64 Tensor Cores**.
  - FP64 throughput is limited to **1/64 of FP32 rate**.
  - Observed $\sim$0.5 TFLOPS is consistent with the $\sim$1.1 TFLOPS hardware peak.
  - DGEMM performance is fundamentally constrained by narrow FP64 pipelines.
- **Key Takeaway**
  - Large FP64 performance gap reflects **architectural design choices**.
  - A30 is optimized for **HPC and numerical linear algebra**.

# cuSOLVER for LAPACK-like Workloads



- **cuSolverDN** is designed to solve dense linear systems of the form:

$$Ax = b$$

  where $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, and $x \in \mathbb{R}^n$.

- **Factorizations provided:**
  - **LU** with partial pivoting for general matrices.
  - **QR** factorization.
  - **Cholesky** for symmetric/Hermitian positive definite matrices.
  - **LDL** (Bunch-Kaufman) for symmetric indefinite matrices.
- **Decompositions:**
  - Singular Value Decomposition (SVD).
  - Bidiagonalization.

# cuSOLVER for LAPACK-like Workloads



- **cuSolverDN** is designed to solve dense linear systems of the form:

$$Ax = b$$

  where $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, and $x \in \mathbb{R}^n$.

- **Design Philosophy:**
  - Targets computationally intensive LAPACK routines.
  - Provides an API compatible with LAPACK.
  - The main idea is to allows users to accelerate bottlenecks on the GPU while keeping other parts of the code on the CPU.

# Solving a Linear System with cuSOLVERDN



To solve a linear system $Ax = b$ using the cuSOLVERDN library, the process typically involves two main steps, mirroring the LAPACK approach:

1. **Factorization:** Compute the LU factorization of the coefficient matrix $A$ using `cusolverDnDgetrf`. This decomposes $A$ into $P \cdot L \cdot U$.
2. **Solve:** Solve the system using the computed factors with `cusolverDnDgetrs`. This involves forward and backward substitutions.

# Solving a Linear System with cuSOLVERDN



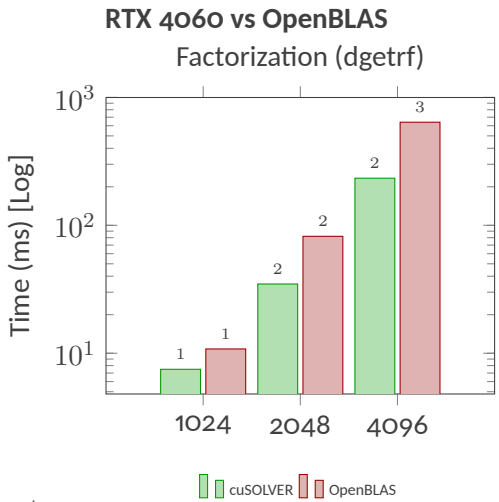
To solve a linear system $Ax = b$ using the cuSOLVERDN library, the process typically involves two main steps, mirroring the LAPACK approach.

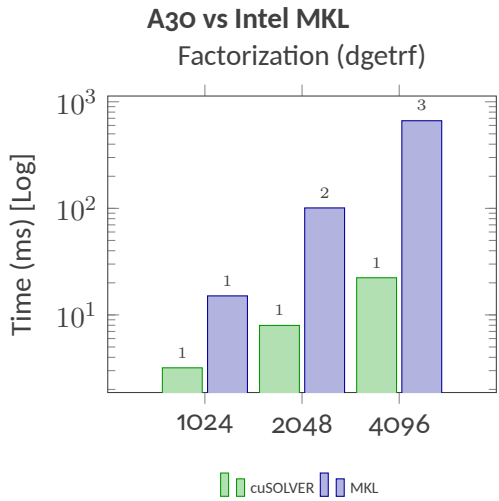
**Main Bottlenecks on GPUs:**

- **Pivoting:** The LU factorization requires partial pivoting for numerical stability. This involves finding the pivot element (reduction) and swapping rows (irregular memory access), which are sequential and memory-bound operations that hurt GPU parallelism.
- **Triangular Solves (TRSM):** The **forward** and **backward substitutions** in `getrs` have dependencies between rows/columns, limiting the available parallelism compared to matrix multiplication (GEMM).

# cuSOLVERDN Performance Factorization Throughput

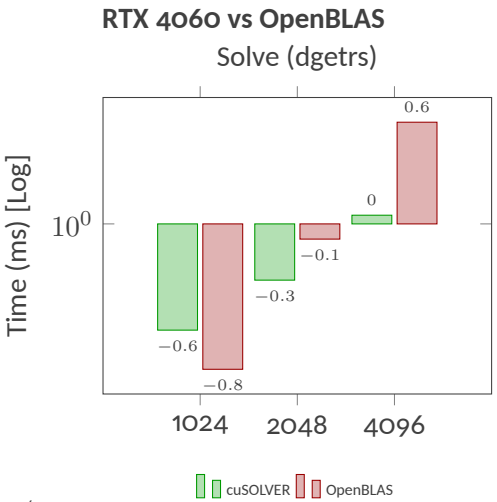


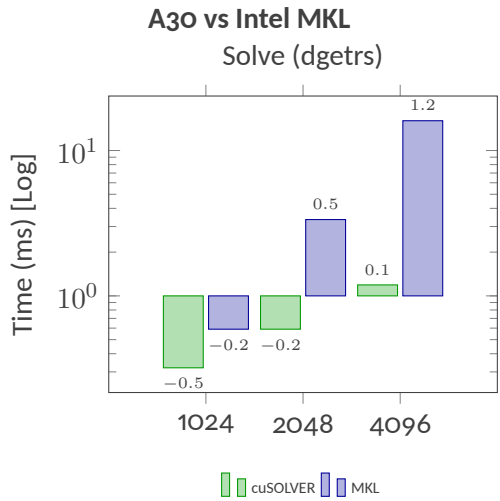


# cuSOLVERDN Performance Factorization Throughput

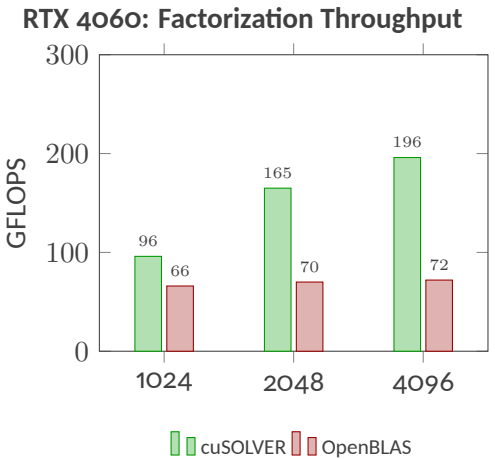


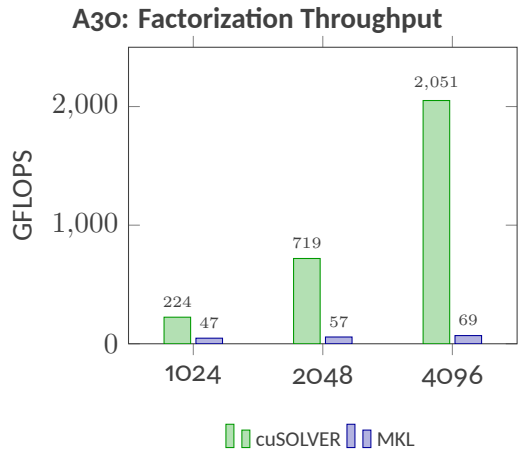


# cuSOLVERDN Performance Factorization Throughput





# Interpreting cuSOLVERDN Performance Results



- **Factorization (dgetrf):**
  - cuSOLVERDN achieves significant speedups over CPU libraries (OpenBLAS, MKL) for LU factorization.
  - The GPU's parallelism effectively accelerates the computationally intensive parts of the factorization.
  - However, performance is still limited by pivoting and memory-bound operations.
- **Solve (dgetrs):**
  - The solve step shows more modest speedups due to inherent sequential dependencies in triangular solves.
  - While GPUs can accelerate some parts, the limited parallelism restricts overall performance gains.
- **Overall Takeaway:**
  - cuSOLVERDN provides performance improvements for dense linear algebra tasks on GPUs.
  - The effectiveness varies between factorization and solve phases.

# The other cuSOLVERDN Routines



The cuSOLVERDN library also provides other LAPACK-like routines, each with its own performance characteristics on GPUs:

- **QR Factorization (dgeqrf)**: Similar performance characteristics to LU factorization, with speedups over CPU libraries.
- **Cholesky Factorization (dpotrf)**: Generally faster than LU due to the absence of pivoting, achieving higher throughput on GPUs.
- **SVD (dgesvd)**: More complex and computationally intensive, with performance gains depending on matrix size and GPU capabilities.
- **Bidiagonalization (dgebd2)**: Similar to SVD, with performance influenced by the algorithmic complexity and data movement.

# The other cuSOLVERDN Routines



The cuSOLVERDN library also provides other LAPACK-like routines, each with its own performance characteristics on GPUs:

- **QR Factorization (dgeqrf)**
- **Cholesky Factorization (dpotrf)**
- **SVD (dgesvd)**
- **Bidiagonalization (dgebd2)**

As we have said many times, it is crucial to:

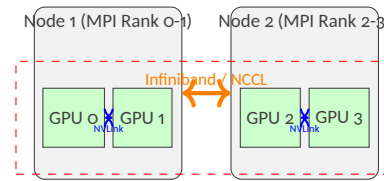
- Profile your specific workload.
- Understand the performance characteristics of each routine on your target GPU!
- Test always on your Workloads!

# cuSOLVERMp: Distributed Dense Linear Algebra



**cuSOLVERMp** is the NVIDIA library for distributed-memory dense linear algebra, designed to scale across multiple GPUs and nodes.

- **Multi-process, Multi-GPU**:
  - Follows the *One process per GPU* paradigm.
  - Seamless integration with MPI applications.
- **ScaLAPACK Compatibility**:
  - C interfaces designed to **mirror ScaLAPACK**.
  - Porting of legacy distributed CPU codes.
- **High Performance**:
  - Uses **NCCL** (NVIDIA Collective Communication Library) for inter-GPU communication.
  - **Tensor Core** accelerated math kernels.
- **Tools**: Built-in logging and tracing support.




cuSOLVERMp Logic

# cuSOLVERMp: Workflow and Data Layout



## Data Layout of Local Matrices

- cuSOLVERMp assumes that local matrices are stored in column-major format (compatible with Fortran/LAPACK).

## Workflow Overview

1. Create a NCCL communicator (NCCL Initialization).
2. Initialize the library handle: `cusolverMpCreate()`.
3. Initialize grid descriptors: `cusolverMpCreateDeviceGrid()`.
4. Initialize matrix descriptors: `cusolverMpCreateMatrixDesc()`.
5. Query the host and device buffer sizes for a given routine.
6. Allocate host and device workspace buffers.
7. Execute the routine to perform the desired computation.
8. Synchronize local stream: `cudaStreamSynchronize()`.
9. Cleanup resources (workspaces, descriptors, handle, communicator).

# cuSOLVERMp error control macro



Like for the other CUDA libraries, it is a good idea to define a macro for error checking:

```
#define CHECK_CUSOLVER_MP(call)                                    \
    do {                                                           \
        cusolverStatus_t status = call;                            \
        if (status != CUSOLVER_STATUS_SUCCESS) {                   \
            fprintf(stderr, "cusolverMp error %s:%d: %d\n",        \
                    __FILE__, __LINE__, status);                   \
            exit(EXIT_FAILURE);                                    \
        }                                                          \
    } while (0)
```

This works exactly like the error checking macros we have seen for cuBLAS and cuSOLVERDN, but adapted for the cuSOLVERMp API.

# cuSOLVERMp Initialization: Overview



The initialization process for cuSOLVERMp closely mirrors the distributed memory setup we saw with ScaLAPACK/BLACS, but with the addition of GPU-specific communication layers (NCCL).

**ScaLAPACK / BLACS**

1. Initialize MPI.
2. Initialize BLACS (Process Grid).
3. Create Context / descriptors.

**cuSOLVERMp**

1. Initialize MPI.
2. Set CUDA device context.
3. Initialize NCCL (GPU Comms).
4. Create library handle & Grids.

**Key Data Types:**

```
ncclUniqueId id; // Unique identifier for NCCL comm
ncclComm_t comm; // NCCL communicator handle
```

# Step 1 & 2: MPI and Device Setup



**Step 1: MPI Setup**

Standard MPI initialization, just like any distributed application.

```
MPI_Init(nullptr, nullptr);
int rank, nranks;
MPI_Comm_size(MPI_COMM_WORLD, &nranks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

**Step 2: CUDA Device Setup**

Assign a specific GPU to the MPI process (usually rank $i$ gets GPU $i \pmod{n}_{gpus}$).

```
const int local_device = getLocalDevice(); // e.g., rank % num_gpus
CUDA_CHECK(cudaSetDevice(local_device));
CUDA_CHECK(cudaFree(nullptr));  // Initialize CUDA context
```

# Step 3: NCCL Communicator Creation



This step is roughly equivalent to `Cblacs_gridinit`, but for the GPU interconnect.

```
ncclUniqueId id;
// Rank 0 generates the unique ID
if (rank == 0) {
    NCCL_CHECK(ncclGetUniqueId(&id));
}
// Broadcast the ID to all ranks so they can join the same communicator
MPI_CHECK(MPI_Bcast(&id, sizeof(id), MPI_BYTE, 0, MPI_COMM_WORLD));
// Initialize NCCL communicator
ncclComm_t comm;
NCCL_CHECK(ncclCommInitRank(&comm, nranks, id, rank));
```

# Step 4, 5 & 6: Handles and Grids



### Step 4: Stream Creation

```
cudaStream_t stream = nullptr;
CUDA_CHECK(cudaStreamCreate(&stream));
```

### Step 5: Library Handle

```
cusolverMpHandle_t handle = nullptr;
CUSOLVER_CHECK(cusolverMpCreate(&handle, local_device, stream));
```

### Step 6: Process Grid (Equivalent to BLACS Grid)

```
cusolverMpGrid_t grid = nullptr;
// Map columns of the process grid to columns of the matrix
CUSOLVER_CHECK(cusolverMpCreateDeviceGrid(handle, &grid, comm,
                nprow, npcol, CUSOLVERMP_GRID_MAPPING_COL_MAJOR));
```

# Cleanup and Synchronization



**Synchronization**

Since cuSOLVERMp is asynchronous, sync the stream before checking results or exiting.

```
CUDA_CHECK(cudaStreamSynchronize(stream));
```

**Cleanup:** *after we have done everything*

Proper destruction order is crucial (reverse of initialization).

```
// Destroy cuSOLVERMp objects first
CUSOLVER_CHECK(cusolverMpDestroyGrid(grid));
CUSOLVER_CHECK(cusolverMpDestroy(handle));
// Destroy NCCL communicator
NCCL_CHECK(ncclCommDestroy(comm));
// Clean up CUDA resources
CUDA_CHECK(cudaStreamDestroy(stream));
// Finalize MPI
MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();
```

# Matrix Management: Descriptors



Just like in ScaLAPACK (where we use `descinit`), cuSOLVERMp requires **matrix descriptors** to understand the **data distribution** (2D block-cyclic).

```
cusolverStatus_t cusolverMpCreateMatrixDesc(
    cusolverMpMatrixDescriptor_t *descr,
    cusolverMpGrid_t grid,
    cudaDataType dataType, // CUDA_R_64F for double
    int64_t M_A, int64_t N_A,            // Global Dimensions
    int64_t MB_A, int64_t NB_A,          // Block sizes
    uint32_t RSRC_A, uint32_t CSRC_A,    // Origin (usually 0,0)
    int64_t LLD_A);                      // Local Leading Dimension
```

- **Parallel with ScaLAPACK**: This is the direct equivalent of the array descriptor array `DESC_` (e.g., `DESC_A`).
- Defines how the global matrix is mapped to the process grid.

# Utility: Calculating Local Sizes



Before **allocating device memory**, we need to know how much memory the local portion of the distributed matrix requires.
**cuSOLVERMp** provides a helper equivalent to ScaLAPACK's `NUMROC`:

```
int64_t cusolverMpNUMROC(
    int64_t n,          // Global dimension (rows or cols)
    int64_t nb,         // Block size
    uint32_t iproc,     // My coordinate (row or col)
    uint32_t isrcproc,  // Source coordinate (usually 0)
    uint32_t nprocs);   // Total processes in dimension
```

# Utility: Calculating Local Sizes



Before **allocating device memory**, we need to know how much memory the local portion of the distributed matrix requires.
**cuSOLVERMp** provides a helper equivalent to ScaLAPACK's `NUMROC`:

```
int64_t m_local = cusolverMpNUMROC(M, MB, proc_row, 0, nprow);
int64_t n_local = cusolverMpNUMROC(N, NB, proc_col, 0, npcol);
// Allocate on GPU
CUDA_CHECK(cudaMalloc(&d_A, m_local * n_local * sizeof(double)));
```

# Data Distribution Utilities



To simplify testing and porting, cuSOLVERMp provides utilities to scatter/gather data between a single host process and the distributed device memory.

`cusolverMpMatrixScatterH2D`

- Scatters a global matrix (on Host) to distributed matrices (on Device).
- Only for testing/debugging (not high performance).

```
cusolverMpMatrixScatterH2D(handle, M, N, d_A, IA, JA, descrA,
                           root, h_src, h_ldsrc);
```

`cusolverMpMatrixGatherD2H`

- Gathers a distributed matrix (from Device) to a global matrix (on Host).
- Useful for verifying results.

# Data Distribution Utilities



To simplify testing and porting, cuSOLVERMp provides utilities to scatter/gather data between a single host process and the distributed device memory.

`cusolverMpMatrixScatterH2D`

- Scatters a global matrix (on Host) to distributed matrices (on Device).
- Only for testing/debugging (not high performance).

`cusolverMpMatrixGatherD2H`

- Gathers a distributed matrix (from Device) to a global matrix (on Host).
- Useful for verifying results.

```
cusolverMpMatrixGatherD2H(handle, M, N, d_A, IA, JA, descrA,
                          root, h_dst, h_lddst);
```

# Available Dense API



The available dense API are:

- `cusolverMpGetrf` which **computes the LU factorization** of a general matrix.
  - `cusolverMpGetrf_bufferSize` which computes the size of the workspace needed for `cusolverMpGetrf`.
- `cusolverMpGetrs` which **solves a system of linear equations** with a general matrix using the LU factorization computed by `cusolverMpGetrf`.
  - `cusolverMpGetrs_bufferSize` which computes the size of the workspace needed for `cusolverMpGetrs`.
- `cusolverMpPotrf` which computes the Cholesky factorization of a symmetric positive definite matrix.
  - `cusolverMpPotrf_bufferSize` which computes the size of the workspace needed for `cusolverMpPotrf`.

# Available Dense API



- `cusolverMpPotrs` which **solves a system of linear equations** with a **symmetric positive definite** matrix using the Cholesky factorization computed by `cusolverMpPotrf`.
  - `cusolverMpPotrs_bufferSize` which computes the size of the workspace needed for `cusolverMpPotrs`.
- `cusolverMpGeqrf` which **computes the QR factorization** of a general matrix.
  - `cusolverMpGeqrf_bufferSize` which computes the size of the workspace needed for `cusolverMpGeqrf`.
- `cusolverMpOrmqr` which multiplies a matrix by the orthogonal matrix Q from a QR factorization computed by `cusolverMpGeqrf`.
  - `cusolverMpOrmqr_bufferSize` which computes the size of the workspace needed for `cusolverMpOrmqr`.

# Available Dense API



- `cusolverMpGels` which **solves a system of linear equations** with a general matrix using the QR factorization computed by `cusolverMpGeqrf`.
  - `cusolverMpGels_bufferSize` which computes the size of the workspace needed for `cusolverMpGels`.
- `cusolverMpSytrd` which reduces a symmetric matrix to tridiagonal form (Schur Decomposition).
  - `cusolverMpSytrd_bufferSize` which computes the size of the workspace needed for `cusolverMpSytrd`.
- `cusolverMpStedc` which computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method.
  - `cusolverMpStedc_bufferSize` which computes the size of the workspace needed for `cusolverMpStedc`.

# Available Dense API



- `cusolverMpOrmtr` which multiplies a matrix by the orthogonal matrix Q from a symmetric tridiagonal reduction computed by `cusolverMpSytrd`.
  - `cusolverMpOrmtr_bufferSize` which computes the size of the workspace needed for `cusolverMpOrmtr`.
- `cusolverMpSyevd` which computes all eigenvalues and, optionally, eigenvectors of a symmetric matrix using the divide and conquer method.
  - `cusolverMpSyevd_bufferSize` which computes the size of the workspace needed for `cusolverMpSyevd`.
- `cusolverMpSygst` which reduces a symmetric-definite generalized eigenvalue problem to standard form.
  - `cusolverMpSygst_bufferSize` which computes the size of the workspace needed for `cusolverMpSygst`.

# Available Dense API



- `cusolverMpSygvd` which computes all eigenvalues and, optionally, eigenvectors of a symmetric-definite generalized eigenvalue problem.
  - `cusolverMpSygvd_bufferSize` which computes the size of the workspace needed for `cusolverMpSygvd`.

⚠ Each of these routines has a corresponding `_bufferSize` function that allows you to query the amount of workspace memory needed before performing the actual computation.

This is **crucial for efficient memory management** in distributed GPU environments.

# Weak Scaling (Large Matrix): Throughput



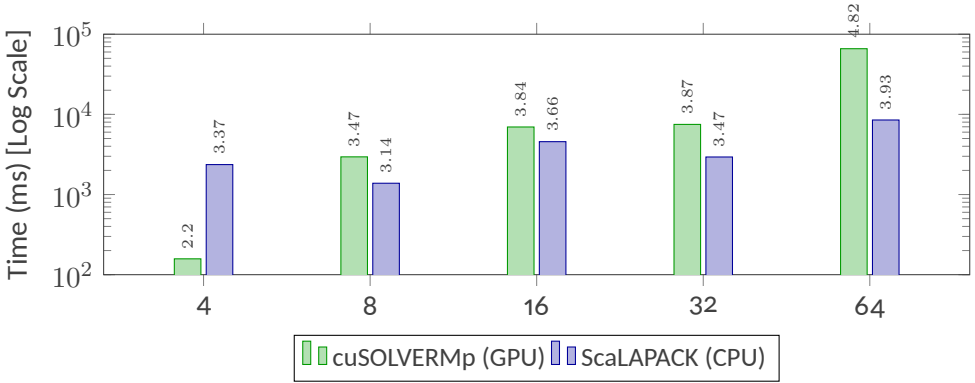
We test cuSOLVERMp against ScaLAPACK (MKL) for the LU factorization of a dense matrix in weak scaling mode.

- We use the routine `cusolverMpGetrf` for cuSOLVERMp and `PDGETRF` for ScaLAPACK.
- Configuration for nodes 1 to 16:
  - 4 Tasks per Node, 1 GPU per Task (on NVIDIA A30 GPUs)
  - 4 Tasks per Node, 16 CPU cores per Task (Intel® Xeon® Gold 6338 CPU @ 2.00GHz)
- Matrix Size per Process: $N_{local} = 4096$
- Matrix Size per Process: $N_{local} = 16384$

# Weak Scaling (Large Matrix): Throughput



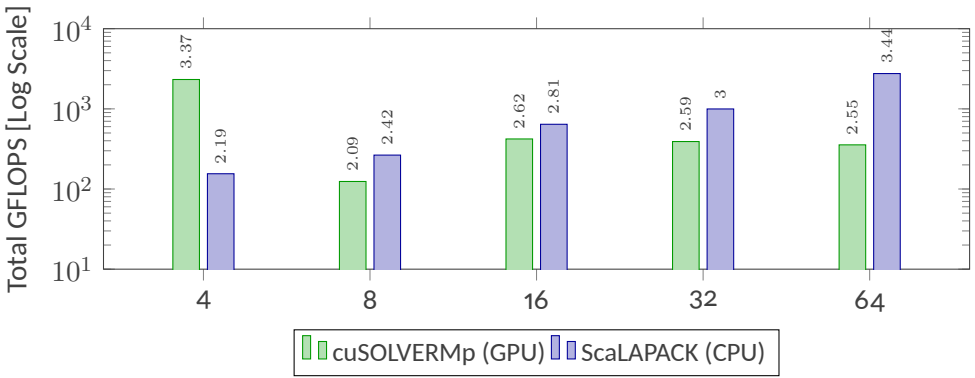
We test cuSOLVERMp against ScaLAPACK (MKL) for the LU factorization of a dense matrix in weak scaling mode.



# Weak Scaling (Large Matrix): Throughput



We test cuSOLVERMp against ScaLAPACK (MKL) for the LU factorization of a dense matrix in weak scaling mode.



# Weak Scaling (Large Matrix): Throughput



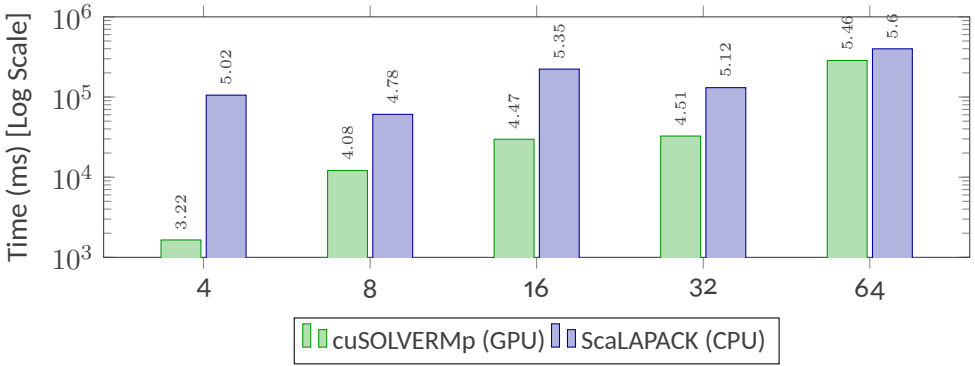
We test cuSOLVERMp against ScaLAPACK (MKL) for the LU factorization of a dense matrix in weak scaling mode.

- We use the routine `cusolverMpGetrf` for cuSOLVERMp and `PDGETRF` for ScaLAPACK.
- Configuration for nodes 1 to 16:
  - 4 Tasks per Node, 1 GPU per Task (on NVIDIA A30 GPUs)
  - 4 Tasks per Node, 16 CPU cores per Task (Intel® Xeon® Gold 6338 CPU @ 2.00GHz)
- Matrix Size per Process: $N_{local} = 4096$
- Matrix Size per Process: $N_{local} = 16384$

# Weak Scaling (Large Matrix): Throughput



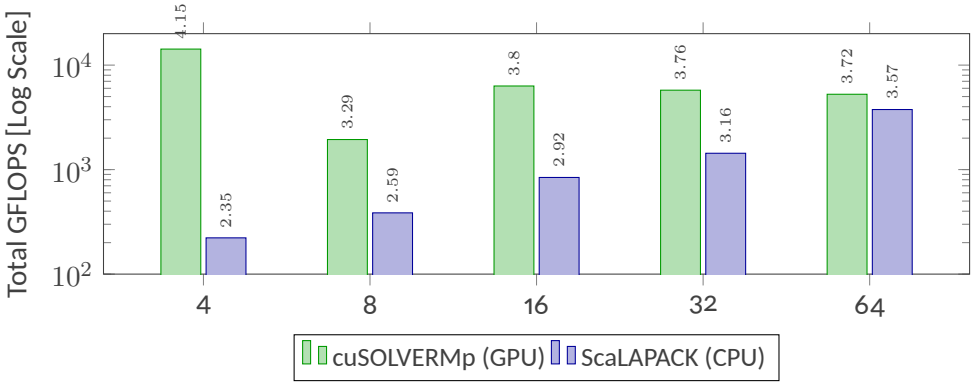
We test cuSOLVERMp against ScaLAPACK (MKL) for the LU factorization of a dense matrix in weak scaling mode.



# Weak Scaling (Large Matrix): Throughput



We test cuSOLVERMp against ScaLAPACK (MKL) for the LU factorization of a dense matrix in weak scaling mode.



# Interpreting the Distributed LU Results



The comparison emphasizes the different sweet spots for CPU and GPU clusters:

- **Massive Throughput on Large Problems**: For $N_{local} = 16384$, cuSOLVERMp is **orders of magnitude faster**. The A30 GPUs exploit Tensor Cores to deliver over $14$ P on small rank counts, crushing the CPU performance. This confirms that for *large-scale dense problems*, GPUs are the superior choice.
- **The Latency penalty on Small Problems**: For $N_{local} = 4096$ distributed across many ranks (e.g., 64), the GPU performance collapses.
  - Computation is too fast to hide communication latency (pivoting, panel exchange).
  - ScaLAPACK on CPUs scales better here because the **slower CPU compute allows for better overlapping with communication**, and CPUs generally handle fine-grained dependencies (like pivoting) with less latency overhead relative to their compute speed.
- **Efficiency**: Weak scaling on GPUs requires **keeping the problem size large** to maintain high efficiency. If the local matrix shrinks or stays constant but relatively small ($< 10$k), the interconnect (PCIe/NVLink) becomes the bottleneck.

# Wrapping up the course

## 3 Conclusions

In this course, we have covered:

- ✔ Some fundamentals of **p**arallel **c**omputing on modern **HPC** systems
  - A quick overview of computer architectures,
  - The difference between Shared Memory and Distributed Memory systems,
  - The main programming models for each of these architectures (**OpenMP** and **MPI**),
  - Programming in modern Fortran.
- ✔ The fundamentals of **BLAS libraries**:
  - in the *Shared Memory* Context via OpenMP and CPU vectorization,
  - in the *Distributed Memory* Context via MPI,
  - in the *GPU* Context via **CUDA**.
- ✔ The fundamentals of **NLA algorithms** and *libraries*:
  - The LAPACK library for Shared Memory systems,
  - The ScaLAPACK library for Distributed Memory systems,
  - The cuSOLVER library for GPU-accelerated systems.

# What remains to do?

3 Conclusions

We have covered a lot of ground, but there is still much more to explore in the world of HPC and numerical linear algebra:

- There is the world of **Sparse Linear Algebra** that we have not touched upon.
- There are many more **advanced algorithms** and **libraries** to explore (e.g., **MAGMA**, **PLASMA**, **PSCToolkit**, **PETSc**, **Trilinos**, etc.).
- Performance tuning and optimization for specific architectures is a deep field in itself.
- Emerging architectures (e.g., **TPUs**, **FPGAs**) and programming models (e.g., **SYCL**, **Kokkos**) are worth exploring.

## The way forward

On a **shorter term**, look for a problem that interests you, and try to implement and optimize a solution using the tools and techniques we have discussed in this course!