Course Notes for the Ph.D. in High Performance Scientific Computing

A Course in Parallel Linear Algebra

Ph.D. in High Performance Scientific Computing

Pasqua D'Ambra* Fabio Durastante† Salvatore Filippone‡

November 5, 2025

University of Pisa

* National Research Council of Italy, Institute for Applied Computing pasqua.dambra@cnr.it
† University of Pisa, Department of Mathematics fabio.durastante@unipi.it
‡ University of Rome Tor Vergata, Department of Civil and Computer Engineering salvatore.filippone@uniroma2.it

A Course in Parallel Linear Algebra

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.
The source code of this book is available at:
`https://github.com/Cirdans-Home/ParallelLinearAlgebra`. You are welcome to contribute by reporting issues, suggesting improvements or even sending pull requests. Please, do not hesitate to contact us if you find any mistakes or if you have any suggestions.

Publisher
This is the version from November 5, 2025 by University of Pisa

All of old. Nothing else ever. Ever tried. Ever failed. No matter. Try again. Fail again. Fail better.

– Samuel Beckett, *Worstward Ho* (1983)

Y así, del poco dormir y del mucho leer, se le secó el cerebro, de manera que vino a perder el juicio.

– Miguel de Cervantes, *Don Quijote de la Mancha* (1605–1615)

42.

– Douglas Adams, *The Hitchhiker's Guide to the Galaxy* (1978–1980)

Preface

These notes serve as an (admittedly brief) introduction to the world of high performance computing software for linear algebra problems. They were prepared for the PhD programme in high performance scientific computing at the University of Pisa: we are grateful for the chance to pass along our accumulated experience.

This specific field of inquiry lies at the intersection between computer science and numerical analysis, but these two fields extend quite a lot in multiple different directions, hence working at the intersection requires getting acquainted with multiple topics.

It is impossible to give a full introduction to these fields in the space available, and many excellent specialist texts are already available on the subject. The present notes should therefore be regarded as a concise overview, developed primarily from the authors' own experience.

We hope nonetheless to share our passion for the challenges and intellectual satisfaction that comes with working in this wonderful field of inquiry, which accounts for a very significant fraction of resources and time in computing centres around the world.

Pasqua D'Ambra
Fabio Durastante
Salvatore Filippone

Contents

List of Figures

List of Tables

Parallel Numerical Linear Algebra: Why? 1

1.1 The Main Ideas

Broadly speaking, Linear Algebra is a branch of mathematics concerned with **vector spaces** and the linear transformations between them. It involves the study of lines, planes, and subspaces, but is also fundamental in understanding systems of **linear equations**, **matrices**, and **vector operations**. Central concepts in Linear Algebra include vectors, matrices, tensors, determinants, eigenvalues, singular values and eigenvectors. While all of these seem to have only a theoretical or mathematical importance, Linear Algebra has widespread applications across science and engineering, including *computer graphics*, *machine learning*, *optimization*, and *physics*, making it a foundational tool in both theoretical and applied mathematics.

Numerical Linear Algebra is the study of how Linear Algebra problems can be solved using numerical methods, particularly on computers; and in the case of this course **parallel computers**. This field is crucial when dealing with *large-scale* problems where exact solutions, when available, are impractical or impossible due to limitations in computational resources and data precision. It focuses on the development and analysis of **efficient**, **stable**, and **accurate** algorithms for matrix computations, such as solving systems of linear equations, computing eigenvalues, performing matrix factorizations, computing matrix functions, and solving matrix equations.

1.1.1 A gallery of problems

Just to give you an idea of the kind of problems we need to face in practice, we will start by going through some examples that routinely appear in applications.

Linear Systems Let us consider the following partial differential equation (PDE) problem:

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega, \end{aligned} \tag{1.1}$$

where Ω is a bounded domain in $\mathbb{R}^d$ and f is a given function $f : \Omega \longrightarrow \mathbb{R}^d$. The solution u is the function we want to compute. The operator Δ is the Laplace operator, which is a second-order differential operator defined as the divergence of the gradient of a function which can be written in cartesian coordinate form as

$$\Delta u = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} + \cdots + \frac{\partial^2 u}{\partial x_d^2}. \tag{1.2}$$

Let us assume that the domain Ω is a d-dimensional unit cube, so that we may discretize it using a finite difference method. This means that we will replace the continuous problem with a discrete one, where we will approximate both the solution u by a vector $\mathbf{u} \in \mathbb{R}^N$ and the function f by a vector $\mathbf{f} \in \mathbb{R}^N$. In a concrete way, we build the grid of points in Ω by dividing it into N equal parts, e.g., for $d = 3$ this means selecting n_1 points in the x_1 direction, n_2 points in the x_2 direction and n_3 points in the x_3 direction. The points are then given by

$$\begin{aligned} x_1 &= \{x_{1,1}, x_{1,2}, \ldots, x_{1,n_1}\}, \quad x_{1,i} = \frac{i}{n_1 - 1}, \quad i = 0, \ldots, n_1 - 1, \\ x_2 &= \{x_{2,1}, x_{2,2}, \ldots, x_{2,n_2}\}, \quad x_{2,i} = \frac{i}{n_2 - 1}, \quad i = 0, \ldots, n_2 - 1, \\ x_3 &= \{x_{3,1}, x_{3,2}, \ldots, x_{3,n_3}\}, \quad x_{3,i} = \frac{i}{n_3 - 1}, \quad i = 0, \ldots, n_3 - 1. \end{aligned}$$

We then approximate the second-order derivatives in (1.2) via a *centered finite difference scheme*, which gives us the following discrete approximation of the Laplacian operator:

$$\begin{aligned} \Delta u_{i,j,k} = & \frac{u_{i+1,j,k} - 2u_{i,j,k} + u_{i-1,j,k}}{h_1^2} \\ & + \frac{u_{i,j+1,k} - 2u_{i,j,k} + u_{i,j-1,k}}{h_2^2} \\ & + \frac{u_{i,j,k+1} - 2u_{i,j,k} + u_{i,j,k-1}}{h_3^2}, \end{aligned}$$

where h_1, h_2 and h_3 are the grid spacings in the x_1, x_2 and x_3 directions, respectively. Similarly, we can consider the discrete values of the function f at the grid points, which we denote by

$$\mathbf{f} = \left(f_{i,j,k}\right)_{i,j,k} \in \mathbb{R}^N,$$

where now $N = n_1 \cdot n_2 \cdot n_3$ is the total number of grid points.

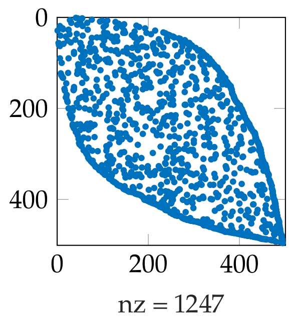

Figure 1.1: Pattern of a sparse matrix, each dot represents a non-zero element of the matrix.

The discretization of the PDE problem leads to a system of linear equations

$$A\mathbf{u} = \mathbf{f},$$

where $A \in \mathbb{R}^{N\times N}$ is what we will call a **sparse matrix**—see Figure 1.1 for a graphical representation of a sparse matrix—that represents the discretized Laplace operator, and $\mathbf{u} \in \mathbb{R}^N$ is the vector of unknowns. Informally, we can say that a sparse matrix is a matrix in which most of the elements are zero, we will came back to specialized algorithms and efficient data structures to store sparse matrices in the second half of the course. For all interesting PDEs the size N of the linear system we have to solve is usually large and we need to use efficient algorithms for it. A good source for reading about finite difference methods is the book [1], other approaches which generate sparse linear systems with features similar to the previous one are the finite element method [2], and the finite volume method [3].

Eigenvalue Problems The second example we want to discuss is the **eigenvalue problem**, which is a fundamental problem in linear algebra. Given a square matrix $P \in \mathbb{R}^{N\times N}$, the eigenvalue problem consists in finding a scalar λ and a non-zero vector $\mathbf{v} \in \mathbb{R}^N$ such that

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

The scalar λ is called an **eigenvalue** of the matrix A, and the vector $\mathbf{v}$ is called an **eigenvector** associated with the eigenvalue λ. An example of application in which we have to compute an eigenvector is the derivation of the **stationary distribution** of a Markov chain [4], which is a stochastic process that undergoes transitions from one state to another within a finite or countable number of possible states. More formally, we can define a stochastic process $\{X_\ell\}_{\ell=0,1,.}$ which takes values in a finite set of states $\mathcal{S} = \{1, 2, \ldots, N\}$, and is defined by a transition matrix $P \in \mathbb{R}^{N\times N}$, where $P_{i,j}$ is the probability of moving from state i to state j; by this construction the sum of the elements in each row of P is equal to 1, the elements of P are nonnegative ($P_{i,j} \geq 0$). The evolution of a probability distribution $\mathbf{p} \in \mathbb{R}^N$ is given by the equation

$$\mathbf{p}_{\ell+1} = P\mathbf{p}_\ell,$$

where $\mathbf{p}_\ell$ is the probability distribution at discrete time ℓ. Under suitable conditions on P, there exists a stationary distribution $\boldsymbol{\pi}$, i.e., a probability distribution at which the process stabilizes, meaning that the distribution does not change over time. This means that we have

$$\boldsymbol{\pi}^\top = \boldsymbol{\pi}^\top P,$$

and $\boldsymbol{\pi}^\top\mathbf{1} = 1$. In many application N is *large*, and we need to find suitable algorithms to compute the eigenvector $\boldsymbol{\pi}$.

Matrix Equations Another important class of problems involves solving **matrix equations**, which arise in various applications such as model reduction in control theory. For instance, consider the **Sylvester equation**, which is a linear matrix equation of the form

$$AX + XB = C,$$

where $A \in \mathbb{R}^{m\times m}$, $B \in \mathbb{R}^{n\times n}$, $C \in \mathbb{R}^{m\times n}$ are given matrices, and $X \in \mathbb{R}^{m\times n}$ is the unknown matrix to be solved for. This equation frequently appears in model reduction techniques, such as balanced truncation, where the goal is to approximate a high-dimensional dynamical system with a lower-dimensional one while preserving key properties. Let us consider a linear time-invariant (LTI) dynamical system described by the following set of ordinary differential equations (ODEs):

$$\dot{\mathbf{x}}(t) = A\mathbf{x}(t) + B\mathbf{u}(t), \tag{1.3}$$

where $\mathbf{x}(t) \in \mathbb{R}^n$ is the state vector, $\mathbf{u}(t) \in \mathbb{R}^m$ is the input vector, $A \in \mathbb{R}^{n\times n}$ is the system matrix, and $B \in \mathbb{R}^{n\times m}$ is the input matrix. Additionally, let the system output be given by:

$$\mathbf{y}(t) = C\mathbf{x}(t), \tag{1.4}$$

where $\mathbf{y}(t) \in \mathbb{R}^p$ is the output vector and $C \in \mathbb{R}^{p\times n}$ is the output matrix. To analyze the system, we are often interested in finding a reduced-order model that approximates the behavior of the original system.

One common approach is to solve the **Sylvester equation**, which arises in model reduction techniques such as balanced truncation. First, we compute the controllability Gramian P and the observability Gramian Q, which satisfy the following Lyapunov equations:

$$AP + PA^\top + BB^\top = 0, \tag{1.5}$$

$$A^\top Q + QA + C^\top C = 0. \tag{1.6}$$

Next, we compute a transformation matrix T that simultaneously diagonalizes P and Q. This involves solving the Sylvester equation:

$$AT + TS = B, \tag{1.7}$$

where S is a diagonal matrix, and T is the transformation matrix that maps the original state space to the reduced-order state space. The Sylvester equation (1.7) is a key step in the model reduction process. Efficient numerical algorithms are used to solve this equation, especially when the dimensions of A, B, and T are *large*.

Machine Learning In recent years, machine learning has become an increasingly important field, with applications in various domains such as image recognition, natural language processing, and recommendation systems. Many machine learning algorithms rely heavily on linear algebra concepts and techniques. For instance, consider the problem of training a linear regression model, which is a fundamental technique in supervised learning. Given a dataset with m samples and n features, we can represent the data as a matrix $X \in \mathbb{R}^{m\times n}$, where each row corresponds to a sample and each column corresponds to a feature. The target variable can be represented as a vector $\mathbf{y} \in \mathbb{R}^m$. The goal of linear regression is to find a vector of coefficients $\boldsymbol{\beta} \in \mathbb{R}^n$ such that

$$\mathbf{y} \approx X\boldsymbol{\beta}.$$

This can be formulated as an optimization problem, where we want to minimize the sum of squared errors between the predicted values and the actual values:

$$\min_{\boldsymbol{\beta}} \|X\boldsymbol{\beta} - \mathbf{y}\|_2^2.$$

Another example is the training of neural networks, which are widely used in deep learning. Neural networks consist of layers of interconnected nodes, where each node performs a linear transformation followed by a non-linear activation function.

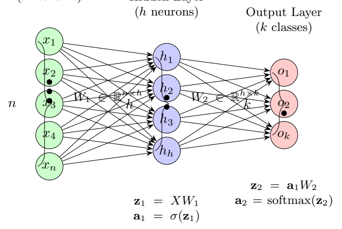
Input Layer
(n features)
Hidden Layer
(h neurons)
Output Layer
(k classes)
x_1
x_2
x_3
x_4
x_n
n
$W_1 \in \mathbb{R}^{n\times h}$
h
h_1
h_2
h_3
h_h
$W_2 \in \mathbb{R}^{h\times k}$
k
o_1
o_2
o_k
$\mathbf{z}_1 = XW_1$
$\mathbf{a}_1 = \sigma(\mathbf{z}_1)$
$\mathbf{z}_2 = \mathbf{a}_1 W_2$
$\mathbf{a}_2 = \mathrm{softmax}(\mathbf{z}_2)$

Figure 1.2: A simple neural network with one hidden layer.

The training process involves optimizing the weights of the connections between the nodes, which can be represented as matrices, e.g., consider a simple feedforward neural network with one hidden layer (Figure 1.2). The input layer has n features, the hidden layer has h neurons, and the output layer has k classes. The weights between the input layer and the hidden layer can be represented as a matrix $W_1 \in \mathbb{R}^{n\times h}$, and the weights between the hidden layer and the output layer can be represented as a matrix $W_2 \in \mathbb{R}^{h\times k}$. The forward pass of the neural network can be expressed as a series of matrix multiplications and non-linear activations:

$$\mathbf{z}_1 = XW_1, \quad \mathbf{a}_1 = \sigma(\mathbf{z}_1), \quad \mathbf{z}_2 = \mathbf{a}_1 W_2, \quad \mathbf{a}_2 = \mathrm{softmax}(\mathbf{z}_2),$$

where σ is a non-linear activation function (e.g., ReLU, sigmoid, see Figure 1.3), and softmax is the softmax function used for multi-class classification. The training of the neural network involves minimizing a loss function, such as cross-entropy loss, using optimization algorithms like stochastic gradient descent. The backpropagation algorithm, which is used to compute the gradients of the loss function with respect to the weights, relies heavily on matrix operations and linear algebra concepts.

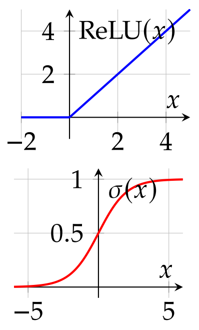

Figure 1.3: ReLU and sigmoid activation functions.

Take home message

Applied mathematics, after the modeling step, is all about solving a suitable combination of linear algebra problems. Nowadays people want to solve ever larger problems, and get reliable results in a reasonable amount of time. This is the reason why we need to use accurate and robust numerical algorithms, and write them to be *efficient*, *scalable* and to run on *parallel computers*.

Sources for Linear Algebra and Numerical Linear Algebra There are many good books on Linear Algebra and Numerical Linear Algebra, and we will not try to give you a complete list of them. However, we will mention a few of them that we found particularly useful. For a general introduction to Numerical Linear Algebra, we recommend the book by Golub and Van Loan [5], which is a classic in the field. It covers a wide range of topics, including matrix factorizations, eigenvalue problems, and singular value decomposition. Other books covering numerical linear algebra with their own perspective include [6] and [7]. A somewhat unusual and refreshing treatment of theoretical topics in Linear Algebra can be found in the book by Axler [8]; for a compehensive treatment of the theory we recommend the books by Horn and Johnson [9, 10]. In the remaining of this note we will focus on the numerical and implementation aspects of Linear Algebra, and we will not go too deep into the theoretical aspects of the subject; nevertheless, we will try to give you some references for the theoretical aspects of the problems we will discuss in the course. If you feel the need to delve more deeply in the theoretical aspects of Linear Algebra, we recommend consulting the above mentioned books [5–10] and the references therein. Some relevant notations and basic facts we use in the following are given in Appendix A.4.

1.2 How *large* is *large*?

In the previous Section Subsection 1.1.1 on page 1 we have seen some examples of problems in numerical linear algebra, and a recurrent theme in all of them is that the size of the problems we are dealing with is *large*. But how *large* is *large*? The answer to this question is "**it depends**". It depends on the problem we are dealing with, the algorithm we are using, the hardware we are using, and the time we have to solve the problem. Furthermore, it is also a matter of when we are asking this question: 20 years ago the answer to "how large is large" would have been different from today, and it will undoubtedly be different yet again 20 years from now.

For instance, if we are dealing with a linear system of equations, the size of the problem could be given in first approximation by the number of unknowns we have to solve for. If we are dealing with a **sparse matrix**, the size of the problem is a combined information given by the number of non-zero elements in the matrix and the overall size of the matrix. If we are dealing with a **dense matrix**, the size of the problem is given by the number of rows and columns in the matrix.

Nowadays, we are are able of solving with relative ease sparse linear systems of equations with several millions of unknowns, and we are pushing towards solving linear systems with hundreds of billions of unknowns. The same applies to the eigenvalue problem, where we are able to compute (few) eigenvalues and eigenvectors of matrices with several millions of rows and columns. The situation for matrix equation is more complicated, and to push towards the solution of large matrix equations we need to be in the case in which the solution of the matrix equation is a low-rank matrix, e.g., in the case of the Sylvester equation (1.7) this means that

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

and $r \ll m, n$. In general, this idea of **exploiting** clever **structures** in the problem we are solving will permit us to solve problem of larger size than the ones we would be able to solve without these structures. For those of you who have a background in computer science, this is akin to the idea of building **data structures** that permit us to store and manipulate large amounts of data in a more efficient way.

1.3 Parallel computers, cluster and supercomputers

To deal with problem which are large in the sense we have just discussed, we need to use **parallel computers**, which are computers that can perform multiple calculations simultaneously. This is achieved by using multiple processors or cores that can work together to solve a problem. Parallel computers can be classified into two main categories:

- **Shared memory systems**: In these systems, all processors share a common memory space. This means that they can access the same data and communicate with each other easily. However, the amount of memory is limited, and the performance can be affected by contention for memory access.
- **Distributed memory systems**: In these systems, each processor has its own local memory. This means that they cannot directly access each other's data, and communication between processors is done through message passing. This allows for larger amounts of memory to be used, but it also requires more complex programming models.

In addition to these two categories, parallel computers can also be classified based on their architecture:

- **Multicore processors**: These are processors that have multiple cores on a single chip. Each core can execute its own thread of instructions, allowing for parallel execution of tasks.
- **Clusters**: These are groups of interconnected computers that work together to solve a problem. Each computer in the cluster is called a node, and they communicate with each other through a network.
- **Supercomputers**: These are extremely powerful computers that are designed to perform complex calculations at high speeds. They often use thousands of processors working in parallel to solve large problems.

To have an idea of what a supercomputer is, let us consider the list[1] of the top 10 supercomputers in the world as of June 2025 and which we have summarized in Table 1.1.

1: You can access the whole list at www.top500.org/lists/top500/2025/06/. The list has been compiled biannually since June 1993 and offers a systematic overview of the most powerful supercomputing systems worldwide, serving as a key reference for tracking progress and trends in high-performance computing.

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*.

Linear algebra problems have been used to benchmark supercomputers for a very long time, and have influenced their design in multiple ways.

The first information we can extract recover from table 1.1 it is that these supercomputers have a huge number of cores; the second is that operating them consumes a lot of power, and the third is that they are all equipped with accelerators, which are specialized hardware components designed to perform specific tasks, namely **graphical processing units** (GPUs). Of course, all of this complexity pays off when we consider the Rmax column, where we can see the *sustained* rate of execution on High-Performance Linpack benchmark (HPL): the number one machine El Capitan[*] is capable of executing 1.7×10^{18} arithmetic operations per second!

We observe that linear algebra is a primary tool for benchmarking supercomputers, since dense linear algebra problems are compute-bound, meaning that their performance depends mainly on the processing capability rather than memory access, and enable the hardware to operate close to its peak performance. Historically, and still today, dense linear algebra has been central to scientific computing, making it a meaningful indicator of raw computational performance and a natural choice for evaluating the capabilities of modern HPC systems. The Linpack benchmark originated from example tests included in the LINPACK User's Guide [11], which measured the performance of solving a dense linear

[*] `https://en.wikipedia.org/wiki/El_Capitan_(supercomputer)`, original meaning `https://en.wikipedia.org/wiki/El_Capitan`

system of size 100 using LU factorization with partial pivoting and the corresponding triangular solvers in double precision. A few years later, this test evolved into a standardized benchmark designed to compare computing systems in terms of floating-point performance. In the current HPL benchmark on which the Top500 list is based, the problem size and software configuration can be chosen by supercomputer vendors to achieve the best performance, although certain rules constrain the operation count — for instance, algorithms such as Strassen's method for matrix–matrix multiplication, which reduce computational complexity below $\mathcal{O}(n^3)$, are not permitted. The continuous interaction between technological advances in supercomputing and the field of linear algebra has driven the evolution of mathematical algorithms and software for linear algebra from the 1980s to the present day. Each major architectural shift — from vector processors to distributed-memory systems, and more recently to hybrid CPU–GPU and heterogeneous exascale platforms — has inspired corresponding innovations in algorithmic design, numerical libraries, and programming models. This co-evolution continues to shape modern high-performance numerical software, ensuring that algorithmic strategies remain aligned with emerging architectures, as we will discuss in the following lectures.

Table 1.1: Top 10 supercomputers from the TOP500 list (June 2025)

Rank	System Description	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	El Capitan - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS, HPE DOE/NNSA/LLNL United States	11,039,616	1,742.00	2,746.38	29,581
2	Frontier - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS, HPE DOE/SC/Oak Ridge National Laboratory United States	9,066,176	1,353.00	2,055.72	24,607
3	Aurora - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
4	JUPITER Booster - BullSequana XH3000, GH Superchip 72C 3GHz, NVIDIA GH200 Superchip, Quad-Rail NVIDIA InfiniBand NDR200, RedHat Enterprise Linux, EVIDEN EuroHPC/FZJ Germany	4,801,344	793.40	930.00	13,088
5	Eagle - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	
6	HPC6 - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, RHEL 8.9, HPE Eni S.p.A. Italy	3,143,520	477.90	606.97	8,461
7	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
8	Alps - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE Cray OS, HPE Swiss National Supercomputing Centre (CSCS) Switzerland	2,121,600	434.90	574.84	7,124
9	LUMI - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.51	7,107
10	Leonardo - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, EVIDEN EuroHPC/CINECA Italy	1,824,768	241.20	306.31	7,494

1.4 What tools are we going to use?

In the context of this course, we will focus on distributed memory systems, which are the most common type of parallel computers used in High-Performance Computing (HPC) environments. These systems are typically composed of many nodes, each with its own local memory, and they communicate with each other using message-passing libraries such as MPI (Message Passing Interface). But before delving into the details of the programming model we need to use, let us first discuss the tools we will be using to write our code. We will be using the following tools:

- **Modern Fortran**: Fortran is a language that has been used for scientific computing since many years. It is particularly well-suited for numerical computations and is still widely used in many scientific applications.
- **Software Version Control: git**: We will be using git as our version control system. This will allow us to keep track of changes to our code and collaborate with others more effectively.
- **MPI, OpenMP, OpenACC, CUDA and other enemies**: We will be using MPI for parallel programming, nevertheless all of the nodes we will deal with will also be equipped by many-core processors, this will drive uso into looking for OpenMP for shared memory parallelism. Furthermore, as you have seen in Table 1.1, the supercomputers we will be using are equipped with accelerators, namely GPUs, and we will look into OpenACC and CUDA for accelerator/GPU programming.
- **Queue manager: Slurm**: We will be using Slurm as our job scheduler to manage the execution of our jobs on the cluster.

1.4.1 Fortran

To keep the number of hours of this course under control we are going to discuss Fortran and its usage together with the implementative and algorithmical aspects of the Parallel Linear Algebra we set out to study. Most of what we discuss *mutatis mutandis* can be ported to C/C++ or other compiled languages. If you want to discover more about Fortran a good sources are the website fortran-lang.org and the book [12].

Fortran, short for "Formula Translation", is one of the oldest high-level programming languages, originally developed in the 1950s by IBM for scientific and engineering applications. It was designed to allow easy translation of mathematical formulas into code, making it particularly well-suited for numerical and computational tasks. Over the decades, Fortran has evolved significantly, with modern versions such as Fortran 90, Fortran 95, Fortran 2003, Fortran 2008, Fortran 2018, and Fortran 2023 introducing features like modular programming, array operations, object-oriented programming, and parallel computing capabilities.

One of the key strengths of Fortran is its performance in numerical computations. It is highly optimized for array and matrix operations, which are fundamental in scientific computing. Fortran compilers are known for their ability to generate efficient machine code, making it a preferred choice for high-performance computing (HPC) applications.

Modern Fortran supports a variety of programming paradigms, including procedural, modular, and object-oriented programming. It also includes features for parallel programming, such as *coarrays* and integration with MPI and OpenMP, enabling developers to write scalable code for distributed and shared memory systems. Despite its age, Fortran remains widely used in fields like climate modeling, computational fluid dynamics, and numerical linear algebra, where performance and reliability are critical.

In the course, we will be using the GNU Fortran compiler (`gfortran`), which is part of the GNU Compiler Collection (GCC). Other, viable options are the Intel Fortran compiler (`ifort`), the Cray Fortran compiler (`ftn`), the LLVM Fortran compiler (`flang`), the PGI Fortran compiler (`pgfortran`), or the NAG compiler (`nagfor`). In reality, the choice of the compiler should also be influenced by the machine and the feature of the language we want to use. For instance, the Cray compiler is particularly well-suited for Cray supercomputers, while the Intel compiler is optimized for Intel architectures. We decided to use `gfortran` as it is widely available and is the default compiler on many systems, furthermore its development is always up to date, or nearly up to date, with the latest Fortran standards.

How to get `gfortran`

To check if `gfortran` is installed on your system, you can run the following command in your terminal:

```
gfortran --version
```

If it is not installed, you can install it using your package manager. For example, on Ubuntu or Debian, you can use:

```
sudo apt-get install gfortran
```

On macOS, you can install it using Homebrew:

```
brew install gcc
```

Another viable option is to install it using `Spack`, which is a package manager for HPC systems. Spack can be downloaded from spack.io, or from their GitHub repository.

Information on how to install and use `Spack` can be found in Appendix B.

Now that we have installed `gfortran`, we can start using it to compile our Fortran code. The basic syntax for compiling a Fortran program is as follows:

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

where `output_file` is the name of the executable file you want to create—selected with the `-o` option, and `source_file.f90` is the name of your Fortran source code file; see Table 1.2 for a list of basic `gfortran` options.

Let us write a simple Fortran program to test our installation. Create a file called `hello.f90`, e.g., by doing

```
touch hello.f90
```

and open it with your favorite text editor. Then, copy and paste the following code into the file:

```fortran
program hello
        use iso_fortran_env, only: output_unit
        implicit none
```

Table 1.2: Basic `gfortran` options.

Option	Description
`-o output_file`	Specify the name of the output executable file.
`-Wall`	Enable all compiler warnings.
`-g`	Generate debug information for debugging.
`-O0`	Disable optimization (default).
`-O1`	Enable basic optimization.
`-O2`	Enable more aggressive optimization.
`-O3`	Enable even more aggressive optimization.
`-fcheck=all`	Enable runtime checks for array bounds and other errors.
`-frecursive`	Enable recursion for subroutines and functions.
`-fPIC`	Enable position-independent code (PIC) for shared libraries.

```
        write(output_unit,'("Hello, World!")')
end program hello
```

This program uses the `iso_fortran_env` module to write the string

```
"Hello, World!"
```

to the standard output. The `implicit none` statement is used to enforce explicit declaration of all variables, which is a good programming practice in Fortran. Now, we can compile the program by running the following command in your terminal:

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

This will create an executable file called `hello`. To run the program, simply execute the following command:

```
./hello
```

You should see the output:

```
Hello, World!
```

Parallel Fortran: coarrays

Fortran is also a parallel language in its own right. A parallel version `hello_par.f90` might look like:

```
program hello
    use iso_fortran_env, only: output_unit
    implicit none
    write(output_unit,'("Hello world from image ",I0," out
    ↪  of ",I0)') this_image(), num_images()
end program hello
```

The coarray program can be compiled by doing[2]

```
caf hello_par.f90 -o hello_par
```

and run with

```
cafrun -np 4 hello_par
```

which will print out something equivalent[3] to

2: The `caf` compiler is a wrapper around the `gfortran` compiler. It is developed by the OpenCoarrays open-source software project [13] which produces an application binary interface (ABI) used by the GNU Compiler Collection (GCC) Fortran front-end to build executable programs leveraging the parallel programming features of Fortran 2018. We will discuss the coarray programming model in more detail in Section 5.2.

3: Try to run this piece of code multiple times, what do you osserve?

```
Hello world from image           2  out of           4
Hello world from image           3  out of           4
Hello world from image           4  out of           4
Hello world from image           1  out of           4
```

We will look at various tools for parallel programming, and weight their advantages and disadvantages. Specifically, the details explaining what is happening in this example code are given in Section 5.2.

1.4.2 Software Version Control: git

Software version control is a system that records changes to files over time so that you can recall specific versions later. It is an essential tool for software development, allowing multiple developers to work on the same project simultaneously without overwriting each other's changes. Version control systems track changes to files, enabling you to revert to previous versions, compare changes, and collaborate with others more effectively. There are two main types of version control systems:

- **Centralized Version Control Systems (CVCS)**: In a CVCS, there is a central server that stores the repository, and developers check out files from this central repository. Examples include Subversion (SVN) and CVS (Concurrent Versions System). The main drawback of CVCS is that if the central server goes down, no one can work on the project.
- **Distributed Version Control Systems (DVCS)**: In a DVCS, every developer has a complete copy of the repository on their local machine. This allows for offline work and better collaboration. Examples include Git, Mercurial, and Bazaar. Git is the most widely used DVCS and is known for its speed, flexibility, and powerful branching and merging capabilities.

Git is a distributed version control system that allows multiple developers to work on a project simultaneously. It was created by Linus Torvalds in 2005 for the development of the Linux kernel. Git is designed to handle everything from small to very large projects with speed and efficiency. Git allows you to track changes to files, collaborate with others, and manage different versions of your codebase. Git is widely used in software development, and it has become the *de facto* standard for version control in many projects. It is used by individual developers, small teams, and large organizations alike.

Continuous integration and **continuous deployment** (CI/CD) are software development practices that aim to improve the quality and speed of software delivery. Continuous integration involves automatically building and testing code changes as they are made, ensuring that new code does not break existing functionality. Continuous deployment takes this a step further by automatically deploying code changes to production after passing tests, allowing for rapid and reliable software releases. It is a key practice in modern software development, enabling teams to deliver new features by being relatively sure that the changes do not break the codebase.

Git is a powerful tool that allows you to:

- Track changes to files and directories.
- Collaborate with others on the same project.
- Create branches to work on different features or bug fixes.
- Merge changes from different branches.
- Revert to previous versions of files.
- Share your code with others using remote repositories.
- Manage conflicts when multiple developers make changes to the same file.
- Keep a history of all changes made to the codebase.
- Tag specific versions of the codebase for release.
- Work offline and synchronize changes later.

- Use hooks to automate tasks during the development process.
- Integrate with other tools and services, such as continuous integration and deployment (CI/CD) systems.

How to get `git`

To start using it on your system, you can check if it is installed by running the following command in your terminal:

```
git --version
```

If it is not installed, you can install it using your package manager. For example, on Ubuntu or Debian, you can use:

```
sudo apt-get install git
```

On macOS, you can install it using Homebrew:

```
brew install git
```

Git could be ued in a completely local environment, but it is mostly used in a distributed environment, where multiple developers work on the same project. In this case, we need to use a remote repository, which is a version of your project that is hosted on the internet or on a network. Remote repositories allow you to share your code with others and collaborate on projects. There are many platforms that provide hosting for Git repositories, such as:

- **GitHub**: A web-based platform that provides hosting for Git repositories. It is widely used for open-source and private projects, and it offers features like issue tracking, pull requests, and project management tools.
- **GitLab**: A web-based platform that provides hosting for Git repositories, similar to GitHub. It also offers features like continuous integration and deployment (CI/CD), issue tracking, and project management tools.
- **Bitbucket**: A web-based platform that provides hosting for Git repositories, with a focus on team collaboration. It offers features like pull requests, issue tracking, and integration with other Atlassian products like Jira.
- **SourceForge**: A web-based platform that provides hosting for Git repositories, with a focus on open-source projects. It offers features like issue tracking, project management tools, and a community of developers.

Another viable option is to use a **self-hosted** Git repository, which is a Git repository that you host on your own server[git]. In our case, we will assume that we are using a remote repository hosted on GitHub, but the same principles apply to all the other platforms.

git: The people from the PHC at the Mathematics Department of the university of Pisa host their own Git server through an instance of Gitea. It can be reached at the address git.phc.dm.unipi.it.

To use GitHub effectively you also need to generate an `ssh` key to regulate and cypher data transfer with the repository. This can be done by using the command `ssh-keygen -t ed25519 -C "your_email@unipi.it"` and then copying the content of the file `~/.ssh/id_ed25519.pub` to your GitHub account settings. Remember that you also need the key to your system agente to be able to use it. This can be done by running the command `ssh-add ~/.ssh/id_ed25519` (assuming that you have selected the default name and location for the key). If the system complains that the agent is not running, you can start it by running `` eval `ssh-agent -s` ``.

First steps with GitHub

To create a new repository on GitHub, follow these steps:

1. Go to github.com and log in to your account.
2. Click on the **New** button in the upper right corner of the page.

3. Fill in the repository name, description, and choose whether it should be public or private.
4. Click on the **Create repository** button.

After creating the repository, you can clone it to your local machine using the following command:

```
git clone
```

followed by the ssh address of the repository, which can be found on the GitHub page of the repository. For example, if your repository is called `my-repo` and your user name is `user-name` the command would be:

```
git clone git@github.com:user-name/my-repo.git
```

The first time you clone a repository, you will get a copy of the entire repository, including all the files, directories, and history. This is a complete copy of the repository, and you can work on it locally without needing to be connected to the internet.

Once you have cloned the repository, you can start working on it. You can create new files, edit existing files, and delete files as needed. When you are ready to save your changes, you can use the following commands[4]:

- `git add <file>`: This command adds the specified file to the staging area, which is a temporary area where you can prepare files for commit.
- `git commit -m "commit message"`: This command creates a new commit with the changes in the staging area and adds a commit message describing the changes.
- `git push`: This command pushes your local commits to the remote repository on GitHub.

4: The first time you want to make a commit to the repository, you need to set your name and email address. This can be done by running the following commands: `git config --global user.name "Your Name"` and `git config --global user.email "your_email@unipi.it"`.

You can also use the `git status` command to check the status of your repository, which will show you which files have been modified, added, or deleted. You can use the `git log` command to view the commit history of your repository, which will show you a list of all the commits made to the repository, along with their commit messages and timestamps. You can also use the `git pull` command to fetch and merge changes from the remote repository to your local repository. This is useful when you are collaborating with others and want to get the latest changes made by other developers.

Exercise 1.4.1 After having created and cloned your first repository to your local machine, create a new file called `GITCOMMANDS.md` in the repository and use the GitHub Markdown syntax to write a short description of the `git` commands you have learned so far. Then *add*, *commit* and *push* this modification to the remote repository.

We will see more about `git` in the next lectures, while we need it to store and share our code.

These notes

Also these notes are stored in a Git repository, which is hosted on GitHub, and can be found at:

https://github.com/Cirdans-Home/ParallelLinearAlgebra

You can use the feature of Git to propose changes to the notes, by creating a **pull request** to the repository. This is a way to suggest changes to the notes, and it will be reviewed by the author before being merged into the main branch of the repository. You can also use the **issues** feature to report bugs or suggest new features for the notes. This is a way to communicate with the author and other users of the notes, and it will be used to track the progress of the changes.

1.4.3 MPI, OpenMP, OpenACC, CUDA and other enemies

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.

- **MPI (Message Passing Interface)**: MPI is a standardized and portable message-passing system that allows processes to communicate with each other in a parallel computing environment. It is widely used in high-performance computing (HPC) applications and provides a set of functions for point-to-point and collective communication, synchronization, and data distribution.
- **OpenMP**: OpenMP is an API that supports multi-platform shared memory multiprocessing programming. It provides a set of compiler directives, library routines, and environment variables that allow developers to specify parallel regions in their code. OpenMP is primarily used for parallelizing loops and sections of code that can be executed concurrently on multiple threads.
- **OpenACC**: OpenACC is a directive-based programming tool that allows developers to write parallel code for heterogeneous systems, including CPUs and GPUs. It provides a set of directives that enable automatic data movement between the host and device memory, making it easier to offload computations to accelerators.
- **CUDA (Compute Unified Device Architecture)**: CUDA is a parallel computing platform and application programming interface (API) developed by NVIDIA for general-purpose computing on its own GPUs. It allows developers to write programs that can execute on NVIDIA GPUs, providing access to the massive parallel processing power of these devices.

These programming tools are not mutually exclusive, and they can be used together in a single application. For example, you can use MPI for inter-node communication and OpenMP for intra-node parallelism. This framwork is usually described as **MPI+X**, where **X** can be OpenMP, OpenACC, or CUDA. There exist research into the possibility of developing alternatives to the MPI+X framework, and maybe some of you are also involved in this. Here we will not discuss these alternatives, since virtually all the large libraries and applications are based on the MPI+X framework. Nevertheless, porting the ideas and algorithms we will discuss to these alternatives could be an interesting avenue of research.

Further details on these parallel programming tools are discussed in Chapter 3 on page 29, Chapter 4 on page 41, and Chapter 4 on page 41.

1.4.4 Cluster ecosystem: Slurm and Environment Modules

Our code will need to be run on a cluster, and this means that we need to use a job scheduler to manage the execution of our jobs. A job scheduler is a software system that manages the allocation of resources on the cluster and the execution of jobs. It is responsible for scheduling jobs, monitoring their progress, and managing the resources of the cluster. There are many job schedulers available, but we will be using **Slurm** (Simple Linux Utility for Resource Management), which is a free and open-source job scheduler that is widely used in HPC environments. Slurm is designed to be scalable, flexible, and easy to use. It provides a simple command-line interface for submitting jobs, monitoring their progress, and managing resources. Slurm is also highly configurable, allowing you to customize its behavior to suit your needs. Since this is not a course on becoming a system administrator, we will not go into the details of how to install and configure Slurm, we will just focus on how to use it to submit jobs to the cluster and monitor their progress.

The first thing we can do is checking the information on the cluster which are available to Slurm. This can be done via the `sinfo` command, which will show us the status of the nodes in the cluster. To given few examples, let us run in on the **Toeplitz cluster** from the Department of Mathematics of the University of Pisa:

```
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
cl1          up   infinite      1   idle lnx1
cl2*         up   infinite      4   idle lnx[2-5]
all          up   infinite      3    mix gpu[01,03-04]
all          up   infinite      1  alloc gpu02
all          up   infinite      5   idle lnx[1-5]
gpu          up   infinite      3    mix gpu[01,03-04]
gpu          up   infinite      1  alloc gpu02
```

The output of the command shows us the status of the nodes in the cluster, including their availability, the time limit for jobs, the number of nodes in each state, and the list of nodes in each partition[5]. When we have runned this command, the Toeplitz cluster had three partitions:

- **cl1**: This partition has one node, which is currently idle.
- **cl2**: This partition has four nodes, which are all idle.
- **gpu**: This partition has three nodes, which are in a mixed state (some are idle and some are allocated).

Furthermore there is a partition called **all**, which is a virtual partition that includes all the nodes in the cluster.

We can distinguish betweeen two types of jobs: **interactive** and **batch** jobs, they have different purposes and are used in different situations. An **interactive job** is a job that runs in the foreground and allows you to interact with it while it is running, it is usually employed for debugging, testing, compiling or for running interactive data analysis. A **batch job** is a job that runs in the background and does not require user interaction, it is usually employed for running long computations or simulations.

5: **Partition:** in the Slurm language a partition is a set of nodes which can be used together simultaneously. Usually in a cluster there are different partition with different conditions of usage: partitions with few nodes which can run jobs only for a small amount of time and can be used for *debug purposes*, large partition which can run long jobs and have large node counts for *production runs*, and sometimes partition also collects nodes with different architectures or specifications, e.g., nodes equipped with GPUs, nodes specialized in *data transfer* jobs, or containing *fat nodes* which have large RAMs and can be used to do data analysis.

Interactive jobs let us start investigating how to run an interactive job on the cluster. To run an interactive job on the Toeplitz cluster, we can use the `srun` command, which is used to submit jobs to Slurm. The basic syntax for running an interactive job is as follows:

```
srun --partition=cl2 --nodes=1 --ntasks-per-node=1
 ↪  --time=00:10:00 --pty bash
```

After running this command, the bash will write something along the lines of[6]:

6: Each Slurm job is assigned a **unique** job ID, which is used to track the job's progress and status.

```
srun: job 12160 queued and waiting for resources
srun: job 12160 has been allocated resources
```

This means that the job has been submitted to Slurm and is waiting for resources to be allocated, once the resources are allocated, the job will start running, and you will be logged into the node where the job is running, i.e., you will see your shell change to something like:

```
durastante@lnx2:~$
```

Let us look at the options we have used in the command:

- `--partition=cl2`: This option specifies the partition to use for the job. In this case, we are using the **cl2** partition.
- `--nodes=1`: This option specifies the number of nodes to use for the job. In this case, we are using one node.
- `--ntasks-per-node=1`: This option specifies the number of tasks to run per node. In this case, we are using one task[7] per node.
- `--time=00:10:00`: This option specifies the time limit for the job. In this case, we are using a time limit of 10 minutes.
- `--pty bash`: This option specifies that we want to run an interactive shell (bash) in the allocated resources.

7: A *task* is a single instance of a program that is running on a node. In the case of an interactive job, we are running a single task, which is the bash shell. In the case of a batch job, we will see next We can run multiple tasks on a single node, or we can run multiple tasks on multiple nodes. In our setting tasks will be the MPI processes we will be using to run our code.

There are many other options that can be used with the `srun` command, another common one which we can use is `--cpus-per-task=4`, which specifies the number of CPUs to use for each task; this is usefuel when we want to run a multi-threaded program, such as a program which uses OpenMP or OpenACC, or if we want to compile our code using multiple threads.

After we are done with the interactive job, we can exit the shell by running the `exit` command. This will terminate the interactive job and return us to our original shell.

Exercise 1.4.2 Use the `srun` command to run an interactive job on the Toeplitz cluster. Use the `--cpus-per-task=4` option to allocate four CPUs for your job. Once you are logged into the node, run the `top` command to see the list of processes running on the node. Then, run the `exit` command to exit the interactive job.

Batch jobs are used to run long computations or simulations that do not require user interaction. To run a batch job on the Toeplitz cluster, we can use the `sbatch` command, which is used to submit batch jobs to Slurm. The basic syntax for running a batch job is as follows:

```
sbatch runscript.sh
```

where `runscript.sh` is a shell script that contains the commands to run the job. The shell script should contain the following lines:

```
#!/bin/bash
#SBATCH --partition=cl2
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --time=00:10:00
#SBATCH --cpus-per-task=4
#SBATCH --job-name=my_job
#SBATCH --output=my_job.out
#SBATCH --error=my_job.err

# Your commands go here
echo "Hello, World!"
```

The first line of the script is called the *shebang* line, and it tells the system which interpreter to use to run the script. In this case, we are using the `bash` interpreter. The next lines are called *Slurm directives*, and they specify the options for the job. They are analogous to the options we used in the `srun` command. The one which are new are

- `--job-name=my_job`: This option specifies the name of the job. In this case, we are using the name `my_job`.
- `--output=my_job.out`: This option specifies the name of the output file for the job. In this case, we are using the name `my_job.out`.
- `--error=my_job.err`: This option specifies the name of the error file for the job. In this case, we are using the name `my_job.err`.

Monitoring jobs is an important part of using a job scheduler, as it allows you to check the status of your jobs and monitor their progress. Slurm provides several commands for monitoring jobs, including:

- `squeue`: This command shows the status of all jobs in the queue, including their job IDs, user names, partition names, and job states.
- `scontrol`: This command provides detailed information about a specific job, including its job ID, user name, partition name, and job state.
- `scancel`: This command is used to cancel a job in the queue. You can use it to cancel a specific job by specifying its job ID.

As an example, if we run the `squeue` command after submitting our interactive job, we will see something like:

```
JOBID PARTITION     NAME     USER ST       TIME  NODES
 ↪  NODELIST(REASON)
12160        cl2     bash durastan  R       1:02      1 lnx2
```

The output of the command shows us the status of the job, including its job ID, partition name, user name, and job state. The job state can be one of the following:

- **PENDING**: The job is waiting for resources to be allocated.
- **RUNNING**: The job is currently running.
- **COMPLETED**: The job has completed successfully.
- **FAILED**: The job has failed.
- **CANCELLED**: The job has been cancelled by the user.

- **TIMEOUT**: The job has exceeded its time limit.

The `NODELIST` column shows the name of the node where the job is running, and the `REASON` column shows the reason why the job is in its current state.

If we want to cancel on or more jobs, we can use the `scancel` command, followed by the job ID of the job we want to cancel. For example, if we want to cancel the job with ID 12160, we can run:

```
scancel 12160
```

Thi command can be executed both if the job is running or if it is in the queue. In the first case, it will terminate the job, by sending a SIGTERM signal to the process, while in the second case it will remove the job from the queue. If you are not the administrator of the cluster, you can only cancel your own jobs, but if you are the administrator, you can cancel any job in the queue.

1.4.5 Environment Modules

The **Environment Modules** system is a tool that allows users to dynamically modify their environment variables[8], such as the `PATH`, `LD_LIBRARY_PATH`, and `MANPATH` variables. This is useful for managing different software packages and versions on a cluster, as it allows users to load and unload different software packages without having to modify their shell configuration files. The Environment Modules system provides a set of commands for loading and unloading software packages, as well as for displaying the current environment variables.

8: The `PATH` variable contains a list of directories the system checks before running a command. Updating the `PATH` variable enables running any executables found in the directories mentioned in `PATH` from anywhere else on the system without typing the absolute file path. `LD_LIBRARY_PATH` tells the dynamic link loader `ld` where to search for the dynamic shared libraries an application was linked against. The `MANPATH` environment variable specifies where the man command looks for reference manual (`man`) pages.

The classical use cases for the Environment Modules system are when you have more than one version of a software package installed on the cluster, or when you have different software packages that depend on different versions of the same library. For example, if you have two versions of the `gcc` compiler installed on the cluster, you can use the Environment Modules system to load the version you want to use, without having to modify your shell configuration files. Consequently, you can have different version of libraries compiled with the different version of the compiler, and you can use the Environment Modules system to load the correct combination of software packages and libraries for your job.

The Environment Modules system provides several commands for managing your environment variables:

- `module avail`: This command shows the list of available software packages on the cluster.
- `module load <package>`: This command loads the specified software package and modifies the environment variables accordingly.
- `module unload <package>`: This command unloads the specified software package and restores the environment variables to their previous state.
- `module list`: This command shows the list of currently loaded software packages.
- `module show <package>`: This command shows detailed information about the specified software package, including its version and dependencies.

- `module purge`: This command unloads all currently loaded software packages and restores the environment variables to their default state.

Again, since this is not a course on becoming a system administrator, we will not go into the details of how to install, configure and wrirte Environment Modules, we will just mention them when we need to change the environment for our jobs. Some information on how to use them in conjunction with the Spack package manager is discussed in Appendix B.

Before closing this chapter, let us see how to used the Environment Modules system to load different version of the `gcc` compiler on the Toeplitz cluster. First of all let us see which version of the `gcc` compiler are available on the cluster. This can be done by running the `module avail` command, which will show us the list of available software packages on the cluster. The output of the command will be something like:

```
--- /software/spack/share/spack/modules/linux-ubuntu22.04-broadwell ----
armadillo/12.8.1-openmpi-4.1.6-gcc-13.2.0
cmake/3.27.9-gcc-11.4.0
cmake/3.27.9-gcc-12.3.0
cmake/3.27.9-gcc-13.2.0
gcc/12.3.0
gcc/13.2.0
intel-oneapi-compilers/2025.0.4-gcc-11.4.0
intel-oneapi-mkl/2024.0.0-openmpi-4.1.6-gcc-13.2.0
intel-oneapi-mkl/2024.2.2-intel-oneapi-mpi-2021.14.1-oneapi-2025.0.4
intel-oneapi-mpi/2021.14.1-oneapi-2025.0.4
likwid/5.3.0-gcc-13.2.0
metis/5.1.0-gcc-12.3.0
metis/5.1.0-gcc-13.2.0
metis/5.1.0-oneapi-2025.0.4
mumps/5.7.3-intel-oneapi-mpi-2021.14.1-oneapi-2025.0.4
mumps/5.7.3-openmpi-4.1.6-gcc-13.2.0
netlib-scalapack/2.2.0-openmpi-4.1.6-gcc-13.2.0
openblas/0.3.26-gcc-12.3.0
openblas/0.3.26-gcc-13.2.0
openblas/0.3.29-oneapi-2025.0.4
openmpi/4.1.6-gcc-12.3.0
openmpi/4.1.6-gcc-13.2.0
parmetis/4.0.3-openmpi-4.1.6-gcc-13.2.0
plasma/23.8.2-gcc-13.2.0
suite-sparse/7.3.1-gcc-13.2.0
suite-sparse/7.8.3-oneapi-2025.0.4
superlu-dist/8.2.1-openmpi-4.1.6-gcc-13.2.0
superlu/5.3.0-gcc-13.2.0

----------------------- /data/software/modules -----------------------
advanpix/4.8.0     cocoa/5.4    julia/1.10.4  julia/gpu-compiled
anaconda3/2024.02  julia/1.8.1  julia/1.11.3  matlab/R2021a

-- /data/software/spackgpu/share/spack/modules/linux-ubuntu22.04-zen3 --
gpu-cmake/3.27.9-gcc-11.4.0
gpu-cmake/3.27.9-gcc-12.3.0
gpu-cmake/3.27.9-gcc-13.2.0
gpu-cmake/3.30.5-gcc-14.2.0
gpu-cuda/12.3.1-gcc-12.2.0
gpu-cuda/12.4.0-gcc-13.2.0
gpu-cuda/12.6.2-gcc-14.2.0
gpu-cuda/12.8.0-gcc-14.2.0
gpu-ddd/3.3.12-gcc-12.2.0
gpu-doxygen/1.9.8-gcc-12.2.0
gpu-eigen/3.4.0-gcc-12.3.0
gpu-gcc/12.2.0
gpu-gcc/12.3.0
gpu-gcc/13.2.0
gpu-gcc/14.2.0
gpu-gdb/14.1-gcc-12.2.0
gpu-gmp/6.2.1-gcc-12.3.0
gpu-gnuplot/6.0.0-gcc-12.2.0
gpu-gsl/2.7.1-gcc-13.2.0
gpu-hdf5/1.14.5-openmpi-4.1.6-gcc-14.2.0
gpu-hpcg/3.1-openmpi-4.1.6-gcc-12.2.0
```

```
gpu-kokkos/4.3.00-cuda-12.4.0-gcc-13.2.0
gpu-metis/5.1.0-gcc-12.2.0
gpu-metis/5.1.0-gcc-13.2.0
gpu-metis/5.1.0-gcc-14.2.0
gpu-mpfr/4.2.1-gcc-12.3.0
gpu-mumps/5.7.3-openmpi-4.1.6-gcc-14.2.0
gpu-openblas/0.3.26-gcc-12.2.0
gpu-openblas/0.3.26-gcc-13.2.0
gpu-openblas/0.3.28-gcc-14.2.0
gpu-openmpi/4.1.6-cuda-12.2.0-gcc-14.2.0
gpu-openmpi/4.1.6-cuda-12.3.1-gcc-12.2.0
gpu-openmpi/4.1.6-cuda-12.4.0-gcc-13.2.0
gpu-openmpi/4.1.6-gcc-12.3.0
gpu-openmpi/4.1.6-gcc-13.2.0
gpu-openmpi/4.1.8-cuda-12.8.0-gcc-14.2.0
gpu-valgrind/3.20.0-openmpi-4.1.6-gcc-12.2.0

Key:
modulepath
```

As you can see from the output, there are many software packages available on the cluster, including different versions of the `gcc` compiler. Observe also that the modules are divided into two blocks. The first block contains the software packages which can be used on the partitions `cl1` and `cl2`, while the second block contains the software packages which can be used on the `gpu` partition[9]. Let us suppose that we want to work on the `cl1/cl2` partition. We can first use `srun` to move on a node there:

9: This is due to the fact that the nodes in the `cl1` and `cl2` partitions are Intel nodes, while the nodes in the `gpu` partition are AMD nodes, hence compilers target different system architectures. You can only compile and execute the software by the correct combination of compiler and architectures. Truth be told, you *could crosscompile* but this is a difficult thing to make work, and why should you bother?

```
srun --partition=cl2 --nodes=1 --ntasks-per-node=1
 ↪  --time=00:10:00 --pty bash
```

Then we can load the `gcc` module we want to use, for example:

```
module load gcc/12.3.0
```

and verif y that the module has been loaded correctly by running:

```
gcc --version
```

The output of the command will show us the version of the `gcc` compiler that is currently loaded in our environment.

If now we want to load a different version of the `gcc` compiler, we can use the `module unload` Command to unload the current version of the `gcc` compiler, and then load the new version we want to use. For example:

```
module unload gcc/12.3.0
module load gcc/13.2.0
```

We can then verify that the new version of the `gcc` compiler has been loaded correctly by running:

```
gcc --version
```

The output of the command will show us the version of the `gcc` compiler that is currently loaded in our environment.

Finally, if we want to unload all the modules we have loaded, we can use the `module purge` command, which will unload all the modules we have loaded and restore the environment variables to their default state.

In the following chapters we will sometime select the modules we want to use to compile and run our code.

Parallel Programming Techniques for Linear Algebra

Programming in action 2

We have seen some of the tools that we are going to use, and we have glanced (in Table 1.1) at the kind of performance we can obtain on those machines: the most powerul machines in the Top500 can achieve a computation rate of $O(10^{18})$ arithmetic operations per second, providing an amount of power we could only dream of in past years*.

But, what is a parallel computer, and how do you *actually* program it? That is what we are going to discuss in the sequel; in particular, we are going to define parallelism in Chapter 3 on page 29.

Extracting the best possible performance from a parallel computer requires three ingredients:

1. Having the best possible implementation for the "serial" (local) parts of the computation;
2. Having an optimal communication strategy;
3. Having a balanced workload distributed across processes.

Therefore after this chapter we will delve into a more detailed discussion about the first topic in Chapter 6 on page 61 and Chapter 7 on page 87, how to get good performance from a "normal" computer, whereas the actual parallelization will be examined later: we will be building from the ground up.

2.1 The evolution of programming

Before delving into the techniques for "serial" programming, let us pause briefly to consider the evolution of programming languages and compilers.

In fact, if we spend a minute for reflection, we realize that the very existence of compilers is absolutely not to be taken for granted: indeed, in the beginning of the history of computer science, the possibility of writing a program capable of translating into machine language with a satisfactory efficiency was very much in doubt, and everybody did programming at machine or assembly language level.

The very first "high level" language with its accompanying compiler was the Fortran programming language in 1954 (see [14] for a historical perspective); the Fortran language is still being actively developed, although it has changed very substantially from its original form. Still, the intellectual enterprise of writing a translator from a language closer to humans into machine language, and producing a final result competitive from a performance point of view, was a formidable challenge, and it has led to many developments in computer science.

* When the first Top500 list was announced in 1993, the most powerful machine in the world was capable of 59 GFLOPS, which is well within reach of a modern workstation even without an accelerator, and the laptops we are using to write these notes would have easily made into the list around position 100.

Programming languages are much more constrained than natural languages; their definition is usually geared towards a certain application area, and is managed by committees that may be "de facto" standardization bodies, or official ones.

Among the languages whose definition falls within the scope of the International Standards Organisation (ISO `https://www.iso.org`) we find:

- Ada
- Algol
- APL
- BASIC
- C
- C++
- COBOL
- Fortran
- Pascal
- PL/I
- Prolog
- Ruby
- SQL
- UML

There are a few other popular languages, for instance Java, Julia, Matlab and Python, which are not standardized by a formal institution.

In any case, each programming language has a "Language Standard" definition. A language standard constraints both the application programmer and the compiler developer:

- The application programmer has to submit a source code respecting the rules laid out in the language document;
- The compiler developer has to produce a compiler that, given a source code that is "legal" in the above sense, produces an executable program that behaves according to the specifications.

Note that the precise rules of the language standard may be much more involved than our intuition tells us, and in some cases thay may give rise to surprising results; this is the main reason why reporting a bug in a compiler[†] is also a nontrivial proposition.

From the above discussion, it follows that for the compiler developer, the first priority is always the correctness of the translated program: efficiency in its intended domain must come after correctness. When we get to efficiency, every language is designed to be convenient in writing certain applications, therefore any language compiler will be optimized with certain criteria and applications in mind.

Among the languages mentioned above, three stand out for our purposes: Fortran, Julia and Matlab. All three languages have been designed for the development of numerical software; among them, Matlab is designed for maximal convenience in prototyping, with its interactive environment, whereas Fortran and Julia are designed to extract maximum performance. Many choices in the language standard definiton have been made to allow for the compilers to make decisions favouring fast programs.

[†] Compilers are, after all, programs, therefore they too exhibit bugs.

Of the *other* languages mentioned in the lists above, the ones that are used the most to achieve the level of performance we are interested in are C and C++; in particular, the C language was designed to write operating systems, and as such provides facilities to precisely control the behaviour of the application program, including the ability to closely match the machine code that is going to be generated. Thus, even if C was never designed specifically for high performance computing, it is possible to generate high performance code. This is also true of C++: again, the language was never designed specifically for HPC, but it can be used effectively.

Over the years, the quality of the available compilers, and of the code they are able to generate, has evolved quite a lot: many things that were done "by hand" with tricky source code constructs in the 90s, can now be safely delegated to the compiler. Nevertheless, writing a program that fully exploits the hardware capabilities will always involve a significant amount of work and ingenuity in finding the best way to express what the programmer's intentions are.

General Parallel Programming Issues 3

We call *parallelism* the ability to have multiple operations completing their execution at the same time. This definition leaves open what we mean by operation; depending on the context we might mean a single machine instruction of any kind, a floating-point operation or something else. In a scientific or engineering application context, the most important parameter is likely the number of floating point operations, because this is usually the limiting factor for the execution speed.

It is possible to have parallelism at multiple levels in a computing system, such as:

- Within a single machine instruction specifying multiple operations; examples of this are the SSE instructions available in the Ix86 Intel processors, or the fused floating-point multiply-and-add available on many modern architectures;
- Within a single processor capable of completing more than one instruction per clock cycle; most modern RISC processors have multiple execution units and are called *superscalar*;
- Within a single silicon chip hosting multiple CPUs; these architectures are known as *multicore* processors*, a core being each complete CPU;
- Within a single computer containing multiple processors;
- Using multiple computers connected through some sort of communication device.

From the application programmer's point of view the first two kinds of parallelism are mostly handled through the compiler and in the libraries implementing the heaviest computational kernels, and are "almost" transparent; in the following we will concentrate on the last three kinds of parallel computing systems.

In classifying the high performance parallel computers currently in common use the discriminating factor is the memory subsystem configuration; thus we distinguish two main kinds of parallel systems:

1. *shared memory* systems;
2. *distributed memory* systems.

3.1 Parallelism: basic concepts

Since its introduction in the early Seventies, the **Flynn taxonomy** [15, 16] has been broadly used to classify computer architectures. According to this scheme, computers belong to one these four categories:

SISD (**S**ingle **I**nstruction **S**ingle **D**ata): this category includes sequential computers where a single stream of data is processed by a single stream of instructions.

* This usage is slightly confusing, since we are calling *processor* both a single CPU and a chip hosting multiple cores; hopefully the context will be sufficient to avoid any major trouble.

SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata): this category mostly includes vector processors capable of handling, with a single instruction, multiple data presented in the form of a vector. The probably most notable example of such machines is the Cray-1 computer equipped with vector registers of length 64.

MISD (**M**ultiple **I**nstruction **S**ingle **D**ata): no significant example of such architectures has ever been built. Flynn places in this category ancient plug-board machines; according to other authors, examples of these architectures are represented by some embedded devices where the same instruction is redundantly executed in multiple streams on the same data in order to achieve fault tolerance by verifying accordance of the results.

MIMD (**M**ultiple **I**nstruction **M**ultiple **D**ata): here, multiple instruction streams concurrently operate on different (sub)sets of data. To this category belongs the vast majority of the parallel computers built in the last 30 years including both shared and distributed memory multiprocessors.

This simple taxonomy can still be used to roughly categorize modern architectures; however the current landscape of high performance computers is much more complex and modern parallel supercomputers can commonly be described as hybrid combinations of different types of architectures. A typical modern supercomputer is composed of several nodes communicating through a network interconnect; nodes include several processors eventually arranged in a shared-memory NUMA (Non-Uniform Memory Access) fashion; each processor is a multicore; each core is equipped with SIMD units like the SSE units in x86 processors or the AltiVec in Power processors capable of executing floating-point vector instructions of size two or four; finally, nodes can be equipped with multiple accelerators such as GPU devices that have their own memory system and are connected to the other processor(s) (commonly referred to as *hosts*) through a PCI link or a high-speed interconnect which is typically hardware specific, e.g., the NVLink in NVIDIA GPUs. Exploiting the considerable computational power of such computers clearly demands programming models and algorithms that are capable of matching their hierarchical structure and heterogeneity. In the following Chapter 4 on page 41 and Chapter 5 on page 47 we will discuss the parallel programming models that can be used to exploit the computational power of modern supercomputers. In this chapter we will discuss the basic concepts of parallel programming and the performance metrics that can be used to evaluate the performance of parallel programs.

3.2 Parallelism: Performance metrics

Since there are countless alternatives in parallel computing, in terms of hardware architectures, programming paradigms and applications, it is necessary to define some metrics to evaluate the performance of a parallel system.

There exists *no single criterion that would be meaningful to all users*. For instance a computer scientist might be interested in the pure **algorithmic speed-up**, while a computational scientist would be more interested in the **time to completion** and in the **maximum size of the problem that**

can be analyzed, and a system administrator just wants to **maximize system utilization**.

We add a word of caution against any published performance measures: there is no real substitute for an actual test with the workload we are interested in. All *benchmarks* are indicators that are only as good as their relation to our intended usage of a parallel computer; procurement of a machine that will be used to run a single critical application day in, day out, is very different from procurement of a machine that will be installed in a computing center serving a wide community of researchers.

3.2.1 Scalability of a parallel system

A *parallel system* may be defined as *the implementation of a parallel algorithm on a given parallel architecture*. With the *scalability theory* we want to organize the evaluation of performance of parallel systems, taking into account all possible usage aspects. Specifically we need to define the appropriate metric to use when tackling the following questions:

- 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?

Given the variety of questions to be answered, we may only give a general criterion, and not a single, precisely defined measurement procedure.

We say that a *parallel system* is *scalable* if it gives the same performance per processor while growing the number of processors involved and/or the size of the problem to be solved. We also say that a *program* is scalable if we improve its performance when increasing the number of processors employed from $p - 1$ to p.

3.2.2 Speed-up and efficiency

Let us first define the *size of a problem* W as the number of basic operations necessary for the best sequential algorithm known to solve the problem.

A given problem of size W may be solved by a program on one or more processors, running in parallel; we may thus define:

- Serial execution time T_s: the time between the start and the end of the program execution on one processor;
- Parallel execution time on p processors T_p: the time between the start of the execution and the completion of execution on the last processor.

We note explicitly the influence of I/O operations on the total execution time; most of the I/O is normally executed once at the start of the simulation, and later operations are typically rather infrequent; depending on our main modeling objective and measurement procedures, we may include them or we may want to look at the scalability of the *computational kernel* in isolation. However if the I/O is an essential part of the application, then it is advisable to look at schemes for parallelizing the I/O operation themselves[1].

1: This a sector quite distinct from parallel linear algebra, there exists several libraries that permits handling Parallel I/O operations and this is an active area of research see, e.g., the ROOT library from CERN root.cern or the CAPIO framework High-Performance-IO/capio.

Given the previous considerations, T_s will be essentially a function of the size of the problem and of the processor speed, while T_p will additionally depend on the number of processors, on the parallel system architecture and on the balance between the communication speed and the processor speed. Thus:

- $T_s = f(W)$
- $T_p = f(W, p, arch)$

It is not possible to define a parallel system performance in an absolute sense, without referring to a specific kind of application. An example is the Top500 list of the most powerful computers in the world from Table **??**: its rules are very specific on the way to measure and report computing power using a predeterimined application, i.e., the factorization of a large dense matrix.

We now define a widely used performance index: the **speed-up** on a problem $S(W, p)$, which is the ratio between:

- $T_s(W)$: Sequential execution time for the *best* algorithm known to solve the problem;
- $T_p(W, p)$: Parallel execution time on p processors.

Thus we have:

$$S(W, p) = \frac{T_s(W)}{T_p(W, p)} \tag{3.1}$$

According to the value attained by $S(W, p)$ we distinguish the situations:

1. $S(W, p) = p$: *linear* speed-up
2. $S(W, p) < p$: *sub-linear* speed-up
3. $S(W, p) > p$: *super-linear* speed-up

A linear speed-up is usually the goal to aim for: having a speed-up of p while using p processors means that all parts of the application have been perfectly parallelized with no penalization from the necessary communication.

There exist a few codes for which it is possible to partition the computation in substantial tasks that can proceed with little or no communication among them, and they are often called "embarassingly" parallel; however in most cases an increase in the number of processors entails an increase of:

- startup times,
- data communication times,
- synchronization overhead.

We may therefore expect that with growing p we will see a growing distance from the ideal linear speedup.

In a few cases it is possible to have a super-linear speed-up, i.e. $S(W, p) > p$; this may happen for two reasons:

1. A serial program may be confronted with a problem instance so large that it cannot exploit efficiently the memory hierarchy (see Section Subsection 3.3.3 on page 37 for some details on this aspect), in terms of cache memory and/or central memory; in this case a partition of the problem on multiple processors may well bring the memory load on each one of them within the hardware resources, thus improving the computational speed;
2. In some graph search algorithms the partition of the problem in multiple subproblems may drastically alter the search sequence, so that the solution may be found much earlier; in these cases, if there is enough memory available, we might even experience a speed-up while running a parallel program on a single processor!

The possible speed-up behaviours vs. number of processors are shown in Figure Figure 3.1; we show a realistic situation for the case of sub-linear speed-up, in that the distance from the linear case grows with p, because with a fixed problem size W the communication overhead increases in percentage.

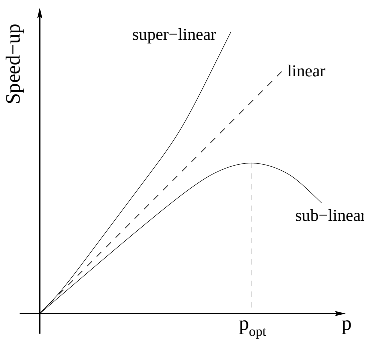

Figure 3.1: Speed-up vs. number of processors

Since the speed-up $S(W, p)$ is a function of both number of processors p as well as problem size W, we often want to look at the behaviour of S varying W with a fixed machine size p; the behaviour in such a case is shown in Figure Figure 3.2.

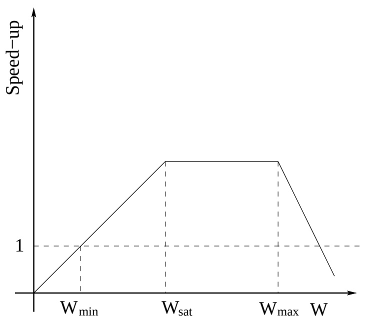

Figure 3.2: Speed-up vs. problem size

From the figure we may see that we do have a benefit from a parallel run only in a certain range of problem sizes $[W_{\text{min}}, W_{\text{max}}]$. For small problem sizes we have that the communication overhead is so large that $S < 1$; the actual crossing point is dependent on the ratio between the speed of the processor and the speed of the network, and on the specific algorithm we are considering. The speedup reaches a peak at W_{sat}, levels off, and then after W_{max} it drops rapidly; this is a typical behaviour if the memory per node grows with W, because the underlying computing node performance degrades rapidly as soon as the available node memory is overflowed.

A closely related metrics is the *efficiency* of a parallel system, defined as the ratio between the speed-up and the number of processors:

$$E(W, p) = \frac{S(W, p)}{p} \tag{3.2}$$

or, substituting the definition (3.1):

$$E(W, p) = \frac{S(W, p)}{p} = \frac{T_s(W)}{T_p(W, p) \cdot p} \tag{3.3}$$

If we rule out the odd cases of superlinear speed-up, we normally have that $E(W, p) \in (1/p, 1)$.

3.2.3 Amdhal's law

Let us consider the following definitions based on the serial execution time $T_s(W)$:

The serial fraction f_s of a program is the ratio between the time spent in code sections that are intrinsically serial and the total time $T_s(W)$.

The parallel fraction f_p of a program is the ratio between the time spent in code sections that are *parallelizable* and $T_s(W)$.

Obviously we have $f_s = (1 - f_p)$; we can now state the *Amdhal's law*:

$$T_p(W,p) = T_s(W) \cdot f_s + \frac{T_s(W)}{p} \cdot f_p \tag{3.4}$$

that is, the parallel executin time $T_p(W,p)$ is an average between $T_s(W)$ e $\frac{T_s(W)}{p}$, weighted with the serial and parallel fractions, respectively.

Using this relation we may derive a simple relation between the speed-up $S(W,p)$, the number of processors p and the serial fraction f_s. We have:

$$\begin{aligned} S(W,p) &= \frac{T_s(W)}{T_p(W,p)} \\ &= \frac{T_s(W)}{T_s(W) \cdot f_s + \frac{T_s(W)}{p} \cdot (1 - f_s)} \\ &= \frac{p}{1 + (p-1) \cdot f_s} \end{aligned} \tag{3.5}$$

Considering the limit for $p \longrightarrow +\infty$ we obtain:

$$S(W,p) = \lim_{p \to +\infty} \frac{p}{1 + (p-1) \cdot f_s} = \frac{1}{f_s} \tag{3.6}$$

This result is quite important, because it gives a hard limit for the results that can be obtained by parallelizing an application. If, for instance, a code has a sequential fraction of 5%, then no matter how much effort we put int, it is impossible to get a speed-up larger than 20.

If we want to have absolute performance, it is then clear that we have to leave no stone unturned in attacking an application code; unfortunately in many cases, especially at fixed problem sizes, the overhead added in terms of synchronization and data exchange is such that the speed-up is not really growing even if we parallelize every line of code.

It is however important not to overestimate the negative conclusions that could be drawn from Amdhal's law. First of all, only a few application context require to get performance at all costs; these kind of applications (e.g. weather forecasts) are among the driving forces in the evolution of computing techniques. On the other hand, in most application areas we are really interested in scaling up the size of the machine when we want to handle much larger problem instances: we don't (usually) fire 100 processors at a linear system of size 1000. Since the linear algebra applications are usually of this kind, the limits defined by Amdhal's law are not as constraining as they seem.

3.2.4 Gustafson's law

As noted, the outlook from Amdhal's law is too pessimistic, and the reason should (by now) be clear: normally we use parallelism to solve big problems, not small ones, and each problem will have a "natural" range beyond which it does not make sense to add processors. To account for this we may use Gustafson's law [17]; if a serial program has a sequential fraction α and a parallel fraction $(1 - \alpha)$ then, as we grow the number of processor, we scale the parallel part of the workload to have $W(n) = \alpha W + (1 - \alpha)nW$, thus obtaining

$$S'_n = \frac{W(n) = \alpha W + (1 - \alpha)nW}{W} = \alpha + (1 - \alpha)n \tag{3.7}$$

Even if this is a much better measure of the possibilities of parallel computers, it still has some problems; the reason is that we are assuming that the serial fraction α remains the same as we scale the program.

3.2.5 Closure

What happens in practice is that the workload is composed of:

- A serial part;
- A parallel part;
- A parallelization/communication overhead.

To estimate precisely the speedup we need to account for all these parts, and this is something very application dependent.

3.3 Paradigms, models and tools for parallel programming

Having given a brief overview of the architectural features important to parallel computing, let us turn our attention to the problems to be tackled when we actually try to implement a parallel application. First of all let us note that the use of a parallel machine does not necessarily mean the use of parallel application; it is quite possible that the machine is needed to run multiple independent instances of a given serial application. In this case the machine is used to maximize the *throughput*, i.e. the number of user requests completed per unit time; this scenario has some analogy with the *work pool* paradigm discussed later, but is nonetheless outside our main interests.

At this point we introduce a distinction between paradigms, models and tools for parallel programming, following [17]:

Paradigm: the logical structure imposed on a parallel algorithm;
Model: the mechanism by which parallelism is expressed in the code;
Tool: the software instrument employed to implement the program code (compiler, library, etc.).

We thus have three different levels related to the problem we are facing[2]. Choices made at these three levels may combine in various ways, although some programming tools "encourage" certain strategies more than others; moreover, any choice at the software level has still to take into account the hardware architecture on which the application will run.

2: In an interview ("Dr. Dobb's Journal", April 1996, www.ddj.com/184409858) Donald E. Knuth, one of the most influential computer scientists in the world, states: ". . . the psychological profiling [of a computer scientist] is mostly the ability to shift levels of abstraction, from low level to high level. To see something in the small *and* to see something in the large."

In this section we give an overview of the alternatives that have been experimented in the last few decades; in the current practice two programming tools, and the models they embody, dominate the field, and will be detailed in Chapter 4 on page 41 and Chapter 5 on page 47.

3.3.1 Algorithmic paradigms

In Figure Figure 3.3 we illustrate some important parallel programming paradigms:

Phase parallel: the computation is organized in a succession of phases in which the tasks alternate between doing their own independent computations and synchronization phases during which the necessary interactions take place;

Divide and conquer: Each task divides its own problem in two or more subproblems, and assigns them to "children" tasks, who in turn will do the same recursively; when all "children" tasks have completed, their "parent" will collect their solutions and combine them to build the solution to its own problem to be sent to its parent, and so on;

Owner computes: there exists a "natural" partitioning of data, with each subset of the partition assigned to a task, such that each task will execute (mostly) the same operations on the subset of data it owns;

Master-worker: one of the tasks takes on the role of controller, distributes parts of the job to the controlled tasks and collects the partial solutions combining them to build the problem solution;

Work pool: there is a shared data structure containing a queue of jobs to be executed; each free task accesses the queue and takes charge of one of the jobs; when the job is completed, it may happen that, given the results, the task has to add more jobs to the queue before taking charge of a new one. This process goes on until all tasks are free and the queue is empty.

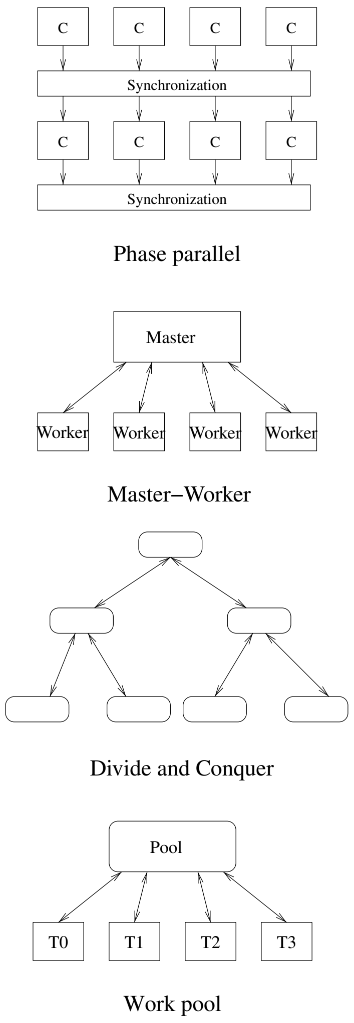

Figure 3.3: Some parallel programming paradigms

The algorithms typical of CSE, and specifically those for fluid dynamics problems, usually lend themselves to a phase parallel structure, with the partitioning of the computing load driven by a partitioning of the simulation domain according to the "owner computes" scheme. This is the natural approach for most applications based on finite difference, volumes or elements discretizations of partial differential equations.

It is however possible to have hybrid situations; consider for instance an application aiming at the optimization of the design of some device. In principle this is a maximization problem for a certain function in the space of all possible design points; the evaluation of the function on a given point is a (potentially very) complex simulation of the device. In this case it may be appropriate to employ a master-worker scheme, where the master task organizes the search in the configuration space, delegating each function evaluation to the worker tasks.

3.3.2 Programming models

We will concentrate on the parallel programming models most relevant to our application domains:

Implicit parallelism: in this model the programmer delegates to the compiler the exploitation of the available parallelism. This is a very difficult task for the compiler, consequently the efficiency attained is usually quite low.

Data parallel: the parallel program contains one control flux (i.e. a single stream of high-level instructions), and the parallelism is available because those instruction are applied to data sets that may be partitioned and operated upon independently. It implies a logically shared memory; perhaps the best example of this model is the use of the HPF (High Performance Fortran) language. The main difference with respect to the previous model is that there are language constructs (e.g.: FORALL) and directives guiding the compiler in the process of parallelizing the source code.

Message passing: The parallel application is built out of a set of processes that may only interact through the exchange of *messages*, i.e. packets of data, under the explicit programmer control. In principle each process may execute a different program, thus guaranteeing the maximal algorithmic flexibility. In practice we see the common usage of the SPMD (Single Program Multiple Data) technique, in which all processes execute a copy of the same program; this is a natural implementation of the *owner computes* paradigm, similar to the data parallel, but having a greater flexibility because different tasks may be executing different sections of code;

Shared variable: in this model we assume a logically shared memory, just as in the data parallel programming, but we may have multiple control fluxes and private data areas as in the message passing model.

The **data parallel** approach and HPF have generated a strong interest in the past, because of the compiler based approach and because of the upward compatibility with respect to serial Fortran applications; however HPF has not been very successful in the marketplace and its usage is declining to the point of disappearing.

The **message-passing model** is the most popular today for applications needing scalability to a large number of processors; the main disadvantage is the need for the programmer to explicitly insert all necessary communications. If the user does not apply a coherent programming discipline, it may lead to applications written at too “low” a level, hard to reuse and maintain. Message passing is implemented with the usage of subroutine libraries.

The **shared-variable** approach may be programmed explicitly by the user, but becomes really interesting when it can be implemented through a compiler; similarly to the data-parallel case, the programmer then inserts directives into the source code to guide the compiler in the desired parallelization.

3.3.3 Roofline model for multicore architectures

Modern computer architectures are organized around a memory hierarchy designed to balance speed, capacity, and cost. At the top of this hierarchy are the registers and cache memories (L1, L2, and L3), which provide extremely fast access to the data most frequently used by a processor. Below them lies the main memory (RAM), followed by secondary storage such as solid-state drives (SSD) or hard disk drives (HDD), and finally tertiary storage for long-term or archival data.

A critical performance parameter across this hierarchy is the memory bandwidth, which represents the rate at which data can be transferred between memory and the processor. As processors have become faster and more parallel, improvements in computational speed have far outpaced increases in memory bandwidth. This imbalance has led to what is known as the *memory wall* — the point at which the latency and limited bandwidth of memory become the primary bottleneck to overall system performance. To better understand this limitation, the *roofline model* [18] is often employed as a visual framework that relates a system's computational throughput to its memory bandwidth. It is a visual performance model that provides an insightful way to bound the performance of a kernel on a given architecture. The roofline model is based on two key hardware characteristics: the peak floating-point performance of the processor (Perf, in FLOP/s) and the peak memory bandwidth (BW, in Bytes/s). It also requires the *operational intensity* (OI) of the kernel, defined as the ratio of the number of floating-point operations to the number of bytes accessed from memory (in FLOP/Byte). The model is represented as a log-log plot of the attainable performance as a function of the operational intensity; see Figure 3.4. Note that we have:

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

The plot consists of two regions: a *memory-bound region* and a *compute-*

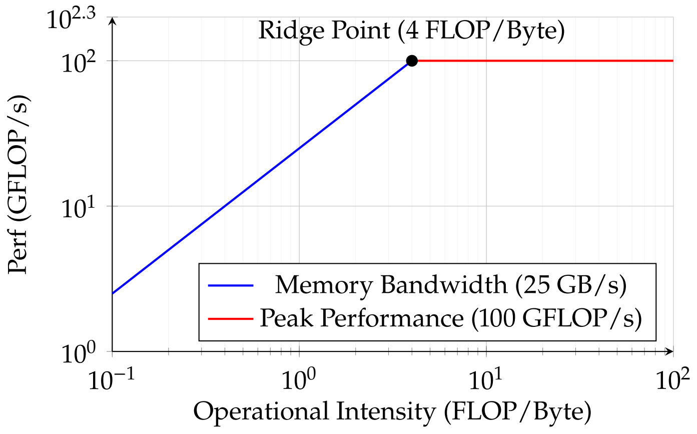

Figure 3.4: Roofline model for a hypothetical architecture with a peak performance of 100 GFLOP/s and a memory bandwidth of 25 GB/s. The ridge point is at an operational intensity of 4 FLOP/Byte.

bound region. The memory-bound region is characterized by a linear increase in performance with increasing operational intensity, limited by the memory bandwidth. The compute-bound region is characterized by a horizontal line at the peak floating-point performance, indicating that the performance is limited by the processor's computational capabilities. The intersection of the two regions is called the *ridge point*, which represents the minimum operational intensity required to achieve peak performance. The roofline model can be used to analyze the performance of different kernels on a given architecture, identify performance bottlenecks, and guide optimization efforts. By comparing the operational intensity of a kernel to the ridge point, one can determine whether the kernel is memory-bound or compute-bound, and focus optimization efforts accordingly. More specifically, since the achievable performance of an application depends linearly on both its operational intensity (OI) and the available memory bandwidth (BW), algorithmic optimization techniques that improve operational intensity and data locality are crucial for enhancing performance. These principles have guided the evolution of optimization strategies in dense linear algebra programming, as exemplified by the progression of the Basic Linear Algebra Subprograms (BLAS) from Level 1 to Level 3 routines (see Chapter 6 on page 61 for details). Higher-level BLAS operations, such as matrix–matrix multiplication, achieve much higher operational intensity by reusing data in fast memory, thereby significantly reducing memory traffic and approaching the compute-bound regime of the roofline model.

To obtain a measure of the bandwidth available to a given application we may use the `STREAM`[3] benchmark [19, 20], which measures the sustainable memory bandwidth (in GB/s) and the corresponding computation rate for simple vector kernels. The benchmark is composed of four simple vector kernels: `COPY`, `SCALE`, `SUM`, and `TRIAD`. The `COPY` kernel copies a vector from one location to another, the `SCALE` kernel scales a vector by a constant factor, the `SUM` kernel adds two vectors together, and the `TRIAD` kernel performs a scaled vector addition. Each kernel is executed multiple times to obtain a reliable measure of the memory bandwidth. The benchmark is designed to be simple and easy to understand, while still providing a good measure of the memory bandwidth available to real-world applications. The `STREAM` benchmark is widely used in the high-performance computing community to evaluate the memory performance of different architectures and to compare the performance of different systems.

3: The STREAM benchmark can be obtained from `http://www.cs.virginia.edu/stream/` or installed via Spack with the command `spack install stream stream_type=double`, usually we are interested in an *multicore* architecture and on using an OpenMP drive, this can be installed with the command `spack install stream_type=double +openmp`.

To obtain a measure of the peak floating-point performance of a given architecture this is more difficult, since it depends on the specific instructions used, the vectorization capabilities of the processor, the number of cores, the clock frequency, and other factors. A simple way to estimate the peak floating-point performance is to use the following formula:

$$\text{FLOP/s} = \text{Number of Cores} \times \text{Clock Frequency (GHz)} \times \text{FLOP per Cycle}$$

where FLOP per Cycle is the number of floating-point operations that can be performed in a single clock cycle. This value depends on the specific architecture and can be obtained from the processor's documentation. For example, a modern x86 processor with AVX2 instructions can perform 8 double-precision floating-point operations per cycle. This means that a processor with 4 cores running at 3 GHz can achieve a peak performance of:

$$\text{Peak FLOP/s} = 4 \times 3 \times 8 = 96\ \text{GFLOP/s}$$

However, this is just an estimate, and the actual peak performance may be lower due to various factors such as memory bandwidth limitations, cache misses, and other overheads. Vendors usually provide a theoretical peak performance value for their processors, which can be used as a reference point for performance analysis.

Intra-node Parallelism 4

During the last decades, the performance of computer microprocessors managed to keep the pace with Moore's law[1] mostly thanks to higher and higher clock frequencies—the result of the deep exploitation of micro-architectural techniques such as pipelining, out-of-order, speculative or superscalar execution or branch prediction that are commonly gathered under the generic name of Instruction Level Parallelism (ILP) techniques. At the beginning of the last decade, this trend has reached the point of diminishing returns mostly due to two hard limits:

1: **Moore's law**, named after Gordon E. Moore, co-founder of Intel, is the observation that the number of transistors on a microchip doubles approximately every two years, leading to a corresponding increase in computational power and a decrease in relative cost. This empirical trend, first articulated in 1965, has driven the exponential growth of the semiconductor industry and has been a key enabler of technological advancements in computing, from personal computers to modern supercomputers. However, as transistor sizes approach physical limits, maintaining this pace of progress has become increasingly challenging, prompting innovations in alternative computing paradigms and architectures: which is why we are now discussing parallelism.

- **Concurrence limit**: despite the fact that ILP techniques can be very sophisticated and complex, there is a limit to the level of concurrence that can be achieved through them: even the most advanced and modern microprocessors cannot issue more than four or five instructions per clock cycle whereas the concurrence available in a wide range of operations is much larger.
- **Power limit**: reducing the microprocessors power consumption and heat dissipation is an issue of considerable importance for the computer industry; for example, it is crucial for extending the battery life of mobile devices and for limiting the operational costs of large scale supercomputers (as a reference, consider that the highest-ranked computers in the Top500 list consume around 30 MW—the equivalent of a small town, or at least, the equivalent of a small town which is not housing a *supercomputing center*). Increasing the processors clock speed brought the processors power consumption and heat dissipation to unsustainable limits due to the fact that power depends on the cube of the frequency.

In order to work around these issues, the microprocessors industry abruptly steered towards Thread Level Parallelism (TLP) techniques which gave birth to a new generation of computer processors, commonly known as *multicores* or *Chip Multi-Processors* (CMP). Conceptually, these processors simply pack onto the same die multiple independent Processing Units (i.e., cores) capable of handling different instructions and data streams. As a result, they are capable of sustaining much higher levels of concurrence. At the same time, considerable performance improvements can be achieved at a much smaller cost in terms of power consumption and heat dissipation. As a matter of fact, increasing the numbers of cores on a chip only increases the power consumption by the same factor due to its linear dependence on the number of transistors. Likewise, by slightly reducing the operational frequency and increasing the number of cores, performance can be improved and power consumption reduced at the same time. A few years from this technological revolution, multicore processors are nowadays ubiquitous and the evolution of computers is driven by a run towards higher and higher numbers of cores per chip.

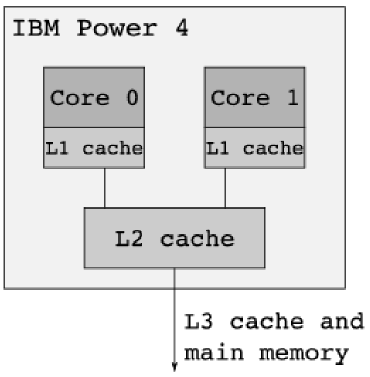

Figure 4.1: The architecture of the IBM POWER4 processor.

4.1 Intra-node parallelism: advanced architectures

The first general-purpose, multicore processors was the POWER4, released by IBM in 2001. This chip, whose architecture is sketched in Figure Figure 4.1 on the preceding page, shipped two cores, each with its own L1 cache memory; L2 cache is shared among the two cores as well as the L3 one which lies off the chip. The two cores access the main memory through a shared bus.

To obtain the analogous of Figure 4.2 for your processor you can use the following command: `lstopo --no-attrs --no-factorize --no-collapse --no-cpukinds --no-legend topology.pdf` The `lstopo` command is part of the `hwloc` package which is available on most Linux distributions, e.g., on Ubuntu you can install it with the command: `sudo apt-get install hwloc`.

The POWER4 chip can be roughly described as two processors glued together on the same die. This idea is at the base of most of the multicore processors developed so far. Last generation multicore processors commercialized by the major chip producers (e.g., AMD or Intel) can pack up to larger numbers of cores on the same die. For example, the AMD EPYC 9655P processor, released in 2023, is a processor with 96 cores and 192 threads; while Intel Xeon w9-3595X processor, released in 2024, is a processor with 60 cores and 120 threads. On a lower scale, the are processors with different kind of cores like the Intel i9-14900HX processor packing 24 Cores and 32 Threads divided into 8 performance cores (16 Threads, 2.2 GHz) and 16 efficient cores (16 Threads, 1.6 GHz) which is depicted in Figure 4.2

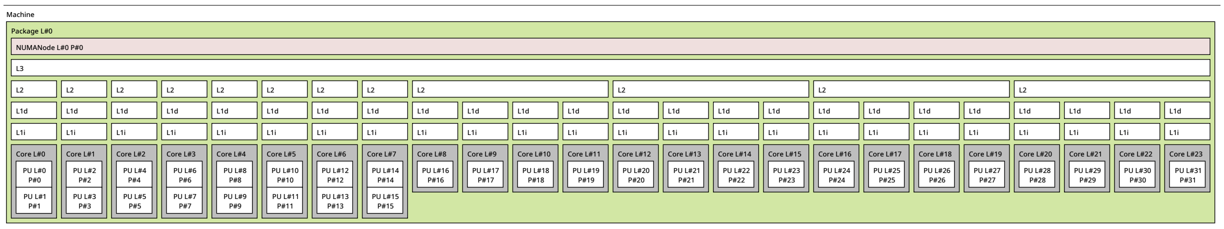

Figure 4.2: The architecture of the Intel i9-14900HX processor, on the left part of the figure we see the eight performance cores, each with is dedicated L2 cache, while on the right part of the figure we see the sixteen efficient cores which shares one L2 cache every four cores. The L3 cache is shared among all the cores, while the L1 cache is private to each core.

Modern CPUs are equipped with a hierarchy of special-purpose cache memories designed to mitigate the performance bottlenecks associated with accessing main memory. These caches are built using static random-access memory (SRAM), a type of memory that is significantly faster than the dynamic random-access memory (DRAM) used in main system memory. Although SRAM is faster and more power-efficient for frequent accesses, it is also more expensive and occupies more physical space per bit than DRAM. As a result, caches are relatively small in capacity but extremely fast, enabling quick access to the most frequently used data. To balance speed, cost, and capacity, CPU caches are typically structured into multiple levels, commonly referred to as L1, L2, and L3. The L1 cache is the smallest and fastest, located closest to the processor core, and usually split into separate instruction and data caches.

A careful analysis of the architecture of the modern processor architectures reveals a detail that shows how limited the scalability can be on certain operations. In light of the EPYC 9655P's simultaneous capabilities—710 GFLOP/s of peak double-precision throughput versus a fixed 614 GB/s of socket memory bandwidth—it becomes clear why workloads naturally bifurcate into compute-bound and memory-bound regimes:

- *memory-bound*: for these are the operations the ratio between number of computations and number of data transfers from the main memory is close to one or smaller. Because no data reuse is possible (i.e., data brought into cache memories are never reused), these operations cannot run any faster than the speed at which data is transferred from the main memory. Since, as shown before, one or a few cores are typically sufficient to saturate the available memory bandwidth, parallelizing these operations for multicore processors only provides a marginal benefit (mostly due to a better utilization of the memory bus) if any at all. One notable operations in this family is the sparse matrix-vector product that performs $\mathcal{O}(nnz)$ floating-point operations on $\mathcal{O}(nnz)$ data, nnz being the number of nonzeroes in the matrix; this operation is the computational kernel of iterative methods for the solution of linear sparse systems. Level1 (e.g., the sum of two vectors) and Level 2 (e.g., the dense matrix-vector product) BLAS operations perform, respectively, $\mathcal{O}(n)$ floating-point operations on $\mathcal{O}(n)$ data and $\mathcal{O}(n^2)$ floating-point operations on $\mathcal{O}(n^2)$ data and are, therefore, another example of operations belonging to this family.

- *computation-bound*: for these operations, the amount of computations is much higher than the amount of data transfer from memory. This property provides a very high temporal locality of data, i.e., data that are read from main memory and brought into caches can be reused in multiple instructions over time. Because each core is equipped with one or more levels of exclusive cache, most of the memory traffic happens in parallel without using the main memory bus. In this case performance scalability can be very good up to relatively high number of cores. To this family belong, for example, the Level 3 BLAS routines (e.g., the dense matrix-matrix product) that perform $\mathcal{O}(n^3)$ floating-point operations on $\mathcal{O}(n^2)$ coefficients. Some very common algorithms, such as the dense LU factorization described in Section Subsection 6.2.3 on page 80, rely on these routines and therefore benefit from the same property.

If the processor needs a piece of data, it first checks the L1 cache. If the data is not found there—a situation known as a cache miss—it proceeds to check the L2 cache, which is larger but slightly slower. If the data is still not found, the search continues to the L3 cache, which is larger still but with higher latency. If none of the caches contain the required data, the CPU finally accesses main memory, which is much slower but has much greater capacity. This hierarchical approach provides an effective compromise: it allows the processor to access frequently used data with minimal delay, while still supporting access to the full range of memory available in the system. By organizing caches into multiple levels, the CPU can take advantage of fast, low-capacity storage for immediate needs, while relying on larger, slower caches and eventually main memory for less frequently accessed data. This structure significantly reduces the average time required to access memory, thereby improving overall system performance. The rationale behind this layered design stems from the trade-offs inherent in memory technology. A single, large cache made entirely of fast SRAM would be prohibitively expensive and consume excessive power and space. Conversely, using only DRAM would result in unacceptable delays for many applications. Multi-level caches enable the CPU to navigate these trade-offs effectively, ensuring that performance remains high even as data sets and workloads grow. Thus, the cache hierarchy is a cornerstone of modern processor architecture, essential for bridging the gap between processor speed and memory latency.

This memory bottleneck (sometimes referred to as *the memory wall* in literature) which hinders the performance of memory-bound operations is clearly a hard constraint towards the scalability of multicore processor, indeed in our AMD EPYC 9655P when an algorithm's operational intensity (FLOPs per byte transferred) exceeds roughly 1.16 FLOP/byte, the processor's arithmetic units throttle performance (compute-bound), whereas below that knee, data-movement across the memory subsystem saturates first (memory-bound). This divergence underpins the "memory-wall": as core counts and aggregate compute grow, the relatively static bandwidth imposes diminishing returns on memory-heavy kernels, making cache reuse or higher bandwidth essential for scaling beyond today's multicore ceilings.

One technological approach for improving on this limitation consists in adding an on-chip interconnect to multicore processors: because cores can directly and efficiently exchange data through this interconnect, the traffic towards main memory can be reduced and the memory bottleneck relieved. Historically, one notable example of this approach has been the Cell Broadband Engine (CBE) processor released by the STI (Sony Toshiba IBM) consortium in November 2006. The CBE was equipped with one PowerPC core and eight SIMD vector cores connected through a ring interconnect with an aggregated bandwidth of 204.8 GB/s. Another distinctive feature of the CBE processor was that cores are not equipped with cache memories but with local scratchpad memories that are explicitly handled by the programmer. Because of its substantial computing power (204.8 Gflop/s for single-precision computations) the CBE processor gained a considerable popularity in the scientific computing community and was chosen as the main computational engine of the RoadRunner supercomputer, the first to cross the 1 Petaflop/s barrier, ranked number 1 on the June 2008 Top500 list[2]. Because of its difficult programming model, much more similar to the one used for distributed memory parallel computers rather than shared memory ones,

2: See the list on the TOP500 website top500.org/lists/top500/2008/06/.

the interest around the CBE processor for scientific computing has rapidly decreased.

Another approach to overcome the memory wall consists in using 3D stacked memory and has become the object of an hectic research activity. Currently used memory layouts can be roughly described as the processor and the memory modules being side-by-side and connected through a wire (the memory bus) in a 2D configuration; this novel approach, instead, disposes memory cells into arrays stacked one on top of the other and then all together on top of the multicore chip in a 3D layout.

4.2 Intra-node parallelism: tools

We are now at a crucial point in our discussion: how do we harness the available computing capabilities of the advanced architectures we have seen in the previous sections? One of the most common answers relies on the concept of a *thread*, which in turn requires the introduction of the concept of a *process*.

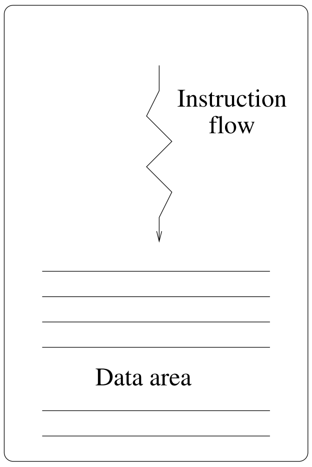

Figure 4.3: A process.

All modern computers support *multiprogramming*, that is, the ability of the system of having multiple programs in execution at the same time. From a *microscopic* point of view you obviously cannot have more programs executing than there are available execution cores, but from a *macroscopic* (that is, on a human time scale) point of view there can be many programs executing at the same time. In this context, the *process* is a basic unit of execution consisting of an instance of a program being run, together with the data it operates upon; thus, it is a *dynamic* entity, as opposed to a *static* entity such as a program (and you can have multiple processes running the same program). The data part is *private* to the process owning it; we will return to the process concept in Chapter 5.

What happens when you have multiple execution cores? One possibility is to have multiple independent fluxes of instruction, each one of them executing part of the program, each one of them with a certain private data area, but all of them sharing the overall program data. These units of execution/instruction fluxes are called *threads*; for scientific applications, we typically aim at having as many active threads as there are available processing cores*.

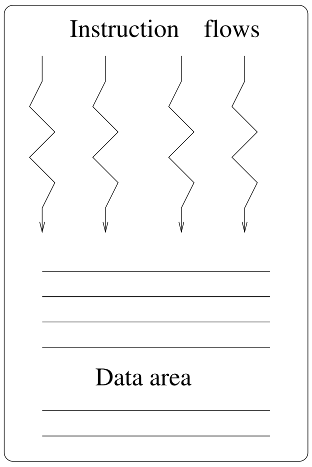

Figure 4.4: A set of threads within a process.

From a programmer's point of view, handling threads requires some software support; on most modern systems you can use *POSIX* threads (see e.g. [21]) to implement what you need, but they are a very low-level tool, and not particularly attuned to the needs of computational scientists.

The most popular programming tool used for scientific applications in conjunction with languages such as Fortran and C is *OpenMP*.

4.2.1 OpenMP

OpenMP is a de-facto standard API (Application Program Interface) for writing shared memory parallel scientific applications in Fortran,

* Other classes of applications, such as databases and web servers, also use threads, but their usage constraints and the programming and tuning techniques are quite different.

C and C++; it was first conceived in 1997, and its specification is maintained by the OpenMP Architecture Review Board (`www.openmp.org`), an organization to which anybody can contributed.

An OpenMP compilation system requires a *compiler* that supports it, and consists of:

- Compiler directives,
- Run time routines,
- Environment variables.

A *compiler directive* is a statement that can be interpreted in two ways:

1. As an instruction to the compiler to do something, or a notification that a compiler is allowed to do something under the responsibility of the programmer;
2. As a comment, for all compilers that do not support the directive system that is being used.

Thus, it is broadly possible to insert OpenMP parallelization statements in the source code whilst at the same time maintaining the original behaviour of the program. The instructions to the compiler can be supplemented by environment variables and/or specific function calls.

In its original conception, OpenMP was concerned mostly with splitting the workload of **do** loops, based on the concept of *fork-join*, shown in Figure 4.5:

1. Upon entering execution of a certain block of code, for instance a **do** loop, multiple threads are activated, and the workload for that block of code is split among them;
2. The threads may interact among them during the execution by using shared memory areas;
3. Upon completion of the block of code, all threads but one are deactivated, and execution proceeds as in a serial program.

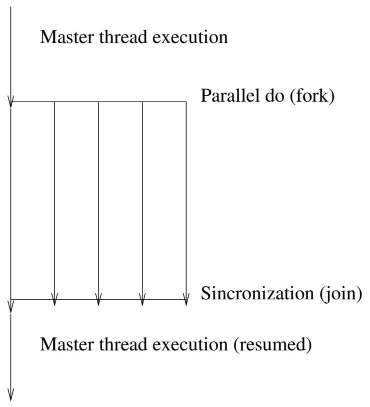

Figure 4.5: The fork-join execution model.

We have described programming with OpenMP referring to threads, meaning that with a logical sharing of memory we have a natural match in shared memory systems, even though this is not mandated by the OpenMP standard. Indeed, there have been attempts at providing a shared memory logical programming view of distributed memory systems, but no such attempt has been particularly successful in practice.

The OpenMP standard has been updated multiple times; at the time of this writing, version 6.0 has been recently released, and version 5.2 is broadly supported by many compilers; in particular, it has been extended with:

- Support for irregular and data-driven dispatching of workload;
- Source-to-source transformations to improve memory hierachy handling and workload splitting among threads;
- Support for novel architectural features such as SIMD extensions and accelerators.

We will show some OpenMP code in the sequel; for a complete presentation see [22–25]. OpenMP is also frequently used in conjunction with MPI, which we will see in Section 5.1, in what is called *nested* parallelism.

4.3 Intra-node parallelism: Accelerators

In relatively recent years the computing world has seen the widespread adoption of *GPUs*, with the most common ones being the devices built by NVIDIA corporation. These devices were originally designed to handle graphics computations, but were soon coopted to perform general purpose tasks, which are commonly described as GPGPU (general purpose programming on graphics programming units); currently there are multiple companies building similar devices, and because of the way they are used they are normally referred to as *accelerators*.

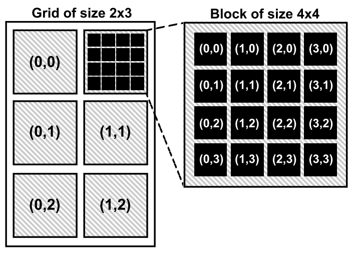

Figure 4.6: A 2D grid of threads

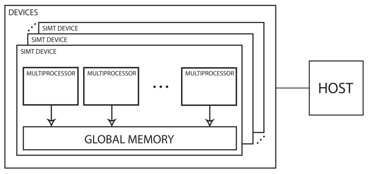

Figure 4.7: SIMT model: host and device

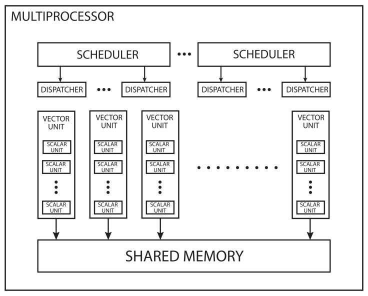

Figure 4.8: SIMT model: a multiprocessor

We now very briefly describe some of the features of the NVIDIA devices, especially from a programmer's point of view. The NVIDIA GPGPU architectural model is based on a scalable array of multi-threaded, streaming multi-processors, each composed of a fixed number of scalar processors, one or more instruction fetch units, on-chip fast memory, which is split between shared memory and cache, plus additional special-function hardware.

CUDA is the programming model provided by NVIDIA for its GPGPUs; a CUDA program consists of a *host* program that runs on the CPU host, and a *kernel* program that executes on the GPU *device*.

The computation is carried on by threads grouped into blocks. More than one block can execute on the same multiprocessor, and each block executes concurrently. During the invocation (also called *grid*) of a kernel, the host program defines the execution configuration, that is:

- how many blocks of threads should be executed;
- the number of threads per block.

Each thread has an identifier within the block and an identifier of its block within the grid (see Figure 4.6). All threads share the same entry point in the kernel; the thread ID can then be used to specialize the thread action and coordinate with that of the other threads.

Figures 4.7 and 4.8 describe the underlying SIMT architecture. Note that a single host may be connected with multiple devices. Each GPU device is made up by an array of multiprocessors and a global memory. The host is connected to devices using a bus, often having a much smaller bandwidth than that of the device global memory. Multiprocessors execute only vector instructions; a vector instruction specifies the execution on a set of threads (called *warp*) with contiguous identifiers inside the block. The warp size is a characteristic constant of the architecture; its value is currently 32 for NVIDIA's GPUs. Programming GPUs is a very specialized activity, and many research efforts are currently devoted to improving their usage. Besides the support available in the latest version of the OpenMP standard [22], there is also the OpenACC programming interface standard `https://www.openacc.org/specification`, and a number of efforts such as SYCL and Kokkos.

Inter-node Parallelism | 5

The most powerful computers in the world listed in the Top500[*] are all *clusters*: a number of computer nodes, most of them equipped with multicore processors and possibily with accelerators, connected through a high-speed network.

The main question from a user's point of view is then: how do you program such a machine? This is what we will (relatively briefly) describe here.

5.1 Inter-node parallelism: MPI

When dealing with clusters, the most common programming tool is the "Message Passing Interface", or MPI.

This project started in the early '90s to consolidate the experience gathered in developing distributed memory applications, and to provide with a unified programming interface to simplify developers' life. It is a de-facto standard, and anybody can participate in its development (see `https://www.mpi-forum.org/`); there exist multiple implementations, from vendors as well as open-source from research projects such as

- 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?

From the MPI 4.1 standard document

> MPI (Message-Passing Interface) is a message-passing library interface specification.

All parts of this definition are significant.

- MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process.
- MPI is a specification, not an implementation; there are multiple implementations of MPI.
- This specification is for a library interface; MPI is not a language, and all MPI operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings which, for C and Fortran, are part of the MPI standard.

[*] You can access the whole list at www.top500.org/lists/top500/2024/11/.

- Extensions to the “classical” message-passing model are provided in collective operations, remote-memory access operations, dynamic process creation, and parallel I/O.

The first item is very important: message passing programming is about having multiple cooperating processes. By definition each process has its own private address space, therefore other processes cannot access data directly; hence the need for messages, in which the two processes cooperate to exchange data.

Every message passing environment thus has a core that comprises at least the following basic functionalities:

- Environment start and stop;
- Identification of participating processes;
- Send and receive data.

In principle, anything you need to cover in your programming can be implemented through these simple functionalities; in practice, as already mentioned, many extensions have been included in the MPI interface specification, because:

- Even if many functionalities can be covered with simple send/receives, robust and efficient implementations require much more sophisticated data structures and algorithms that require specialized knowledge, and it is much better to delegate them to the MPI library implementors behind a unified interface;
- Some issues are purposedly left partially or wholly undefined, such as how to start multiple processes, often interacting with the management system of the cluster we are running on; in some versions of MPI it is even possible to have a number of processes that changes dynamically at runtime;
- Access to some hardware features, such as I/O and direct remote memory access, require specialized interface and/or extensions beyond the programming model defined above.

What does an MPI program look like? An MPI version of “Hello, world” `hello_mpi.f90` might look like:

```fortran
program hello
  use iso_fortran_env, only: output_unit
  use mpi_mod
  implicit none
  character(len=40) :: message
  integer :: myrank, np, ierror, i
  integer :: status(MPI_STATUS_SIZE)
  integer, parameter :: tag=123
  call MPI_Init(ierror)
  call MPI_Comm_rank(MPI_COMM_WORLD,myrank,ierror)
  call MPI_Comm_size(MPI_COMM_WORLD,np,ierror)
  write(message,&
       &'("Hello world from image ",I0," out of ",I0)')
        ↪  myrank,np
  if (myrank == 0) then
    write(output_unit,'(a)') message
    do i=1,np-1
      call MPI_Recv(message,40,MPI_CHARACTER,i,tag,&
           & MPI_COMM_WORLD,status,ierror)
      write(output_unit,'(a)') message
    end do
  else
    call MPI_Send(message,40,MPI_CHARACTER,0,tag,&
         & MPI_COMM_WORLD,ierror)
  end if
  call MPI_Barrier(MPI_COMM_WORLD,ierror)
  call MPI_Finalize(ierror)
end program hello
```

A few observations are in order:

- The type of the data being sent is specified by one of the arguments; in most clusters where the nodes are homogeneous, the data is transferred directly, but in principle it would possible, indeed required, for the MPI implementation to translate the data representation if we are running on a heterogeneous system;
- Every message has an *envelope* consisting of the following:
  1. Source process;
  2. Destination process;
  3. Tag;
  4. Communicator.

  The standard mandates that two messages with the same envelope must be delivered in the order they were sent;
- The `tag` is an arbitrary user-defined attribute; it allows different kinds of messages to be exchanged; in particular, a message with a certain tag might be received before a message with a different tag even if it was sent later and has an otherwise identical envelope;
- The communicator is an extremely important argument because it allows for a clean separation of the messaging space in different portions, each one corresponding to a different communicator object:
  - If the program uses a library, you clearly do not want to the messages defined and exchanged within the library to be erroneously matched to the messages in the user's program; to this purpose, libraries will (very likely) create a new private communicator upon initialization;
  - You may define communicators for groups of processes, to limit the exchange of data; for instance, if you have a 2-dimensional arrangement of processes in rows and columns, it is possible to define communicators for each row/column.

  In our "hello world" example we are using the `MPI_COMM_WORLD` communicator, which is predefined by the MPI implementation at the start of every application, but a realistic application would immediately duplicate it into its own private communicator, even for the user part, via the appropriate environment functions.
- Send and receive functions appearing in the above example are the default ones, but there exist much more sophisticated versions allowing for fine-grained control of the application behaviour.
- We have seen one example of *collective operation* in the call to `MPI_Barrier`, but many other collectives exist and they are very often essential to achieve good performance.

As you can see, there are a number of issues that the user has to take care of when handling the communications among different processes. MPI gives very fine-grained control over the operations of your parallel program, but that control come with the responsibility of properly using the facilities from the library.

5.1.1 Point-to-point operations

Point-to-point operations are the simplest form of data exchange: one process sends a packet of data to another one. Not surprisingly the interfaces to the relevant library functions contain the words *send/receive*.

The description above is deceptively simple: many, many details must be considered. For instance, what happens if a process wants to send a message to another that is unaware (perhaps because it has not yet reached a certain point in the computation)? Do we want the data to flow anyway and be handled by the operating system, perhaps up to a limited size? When is the sender process free to reuse the data area holding the contents to be sent? Do we want to wait until the receiving process becomes ready and prepares a data area sufficient for storing the incoming message? Do we want maximum performance or ease of use? These and many other questions are handled by the MPI standard in a general way defining different *modes* for communicating data, thereby allowing a great flexibility in tailoring the communication to the application needs.

In our PSCToolkit library [26] (and in most of our presentation) we will use a subset of modes:

1. The SEND operation is "locally blocking": as soon as the subroutine returns the application code may freely reuse the data buffer, but this does not necessarily mean that the message has even started its trip, let alone arrived to destination. This is a useful feature when calling the message passing environment from Fortran, where we might want to send data using a "natural" expression which implies that it is held in a compiler-generated temporary variable: it would be highly undesirable to depend on the contents of a temporary not to change until some unrelated event (i.e. the actual message delivery) has happened. On the other hand it would be equally undesirable to only use transmission modes in which the sender waits for the receiver to actually handle the incoming message: not only this provides lower performance, it also makes it harder to write code that does not go into deadlock. In practice the library will perform a copy to an internal buffer and will start an asynchronous send on the copy, returning as soon as the send operation has been requested to the communication subsystem (but not necessarily performed). Given this behaviour, a timing of a send routine is much less than the time it takes for the message to travel to its destination[†];
2. The RECV operation performed on the receiver side is "globally blocking": the call completes only when the data has actually been received and put in the user designated data buffer;

[†] To measure the actual performance of a point-to-point operation the common method is to time a *ping–pong* between two processes: the first sends, the second receives, then sends back to the first, who in its turn receives; the measured time is then assumed to be twice the amount it takes for a one-way trip.

3. The call to the RECV operation is not tightly synchronized; it may be placed before or after the matching SEND. In practice this means that we rely, at least to some extent, to the MPI implementation to provide buffer space for incoming messages for which no RECV has been posted yet;
4. If a task i sends two successive messages to task j, we are guaranteed that delivery will happen in exactly the same order, and thus the order of the RECV should match that of the SENDs; we are giving up the availability of the explicit tag that enables MPI messages to be received out of order. However it is still possible for task j to receive a message sent after i's message by some other task $k \neq i$.

We will also make heavy use of the *overloading* capabilities of the Fortran language, trying to simplify the calling sequences; for each kind of operation we will define a single *generic interface* that accounts for all specific versions. The correct underlying implementation is chosen automatically by the compiler according to the data type (integer, real, complex, character, logical), and to the rank (0 =scalar, 1 =vector, 2 =matrix) of the variable being used. Thus we are guaranteed that it is impossible to incur in a mismatch between the subroutine being called and the data type of the variable.

Other modes available in MPI for point-to-point SEND operations include:

Standard: the MPI library decides whether to use internal buffering or wait for a matching receive;
Buffered: the message is stored in an outgoing buffer, and the operation returns independently of a receive;
Synchronous: the send operation may be started, but it will be completed only after a matching receive has been posted;
Ready: a send may only be issued if a matching receive has already been issued.

Moreover, the send/receive operations may also be issued in their *non-blocking* version: the calling process initiates the operation, and receives a handle to test for successful completion at a later point. This is quite useful, and indeed we provide this facility for the halo data exchange of Sec. 8.3.6.

MPI and OpenMP: which one to choose?

The MPI model makes very minimal assumptions on the underlying hardware architecture, especially for what concerns the memory model; thus it can be used on parallel machines up to thousands of processors, and has been implemented on practically all architectures in common use today.

Even though the initial cost of the parallelization is quite higher than with OpenMP, the scaling up of the application is usually less problematic from a software point of view.

Thus MPI would be the tool of choice when:

- The architecture is a multicomputer (e.g. a Linux cluster);

- the problem potentially requires a massive amount of computing power;
- we want to have a highly scalable and flexible application;
- we have enough time and resources to write the parallel code.

To conclude our comparison between OpenMP and MPI, we state on the basis of our experience an 80/20 heuristics: the total investment needed to have a fully optimized application is generally the same, but is distributed in a different way, 80 % initial parallelization and 20 % tuning for MPI, exactly the opposite for OpenMP.

In parallel programming, just like in many other human enterprises, the devil is in the details.

5.1.2 Collective operations

Collective operations by definition involve (and must be called by) *all* processes participating to a given context, i.e. all tasks in a certain virtual parallel machine; actually we have already seen one example with the synchronization operator `mpi_barrier`.

All functionalities provided with collective operations may be described, and might be implemented, with a sequence of *send/receive* to/from a root process; examples of these naive implementations will be given below. Even in this case there would be an incentive to define a specific calling sequence to avoid code duplications for the most common operations.

However the main reason for defining specific routines is performance: the optimal implementation of a collective operation may be *much faster* (and complex) than the *naive* one. Finding the optimal implementation is an extremely complex enterprise, and we cannot expect the general user to do it; without entering into details [27, 28], we explicitly note that the optimal algorithm for a given operation will depend not only on the number of tasks involved, but also on the performance of the network, on its topology and on the size of the data to be processed, and a quality implementation will automatically choose among multiple algorithms the best one for any given call.

Note also that collective operations introduce new possibilities for deadlocks, such as only a subset of the processes entering a call, or passing to the subroutine arguments that are inconsistent across processes.

Broadcast

The operation of *broadcast* propagates the value of a certain data item from a *root* task to the set of all those participating in the same *context*. The PSCToolkit interface is as follows:

```
call psb_bcast(icontxt, a, root=root)
```

where the optional argument `root` specifies the source task, with a default of 0. After a successful completion the contents of `A` will be the same for all tasks, and will be equal to the contents that were available on task `root` before the call. Thus the functionality of the operation may be achieved by the following naive implementation:

```
if (iam == root) then
  do i=0,np-1
    if (i /= iam) then
      call psb_snd(icontxt,a,i)
    end if
  end do
else
  call psb_rcv(icontxt,a,root)
end if
```

which is, of course, not optimal.

The value for the `root` argument must be the same for all participating processes; otherwise results are unpredictable, with deadlock a very likely outcome. As in the send/receive case the data `A` may be:

- A scalar of type integer, real, complex, logical, character; for character data there is an additional optional argument `length` to override the size of the string to be exchanged;
- A rank-1 or rank-2 array of type integer, real or complex.

A simple and immediate example of the use of a broadcast is a typical application structure in which the initial data input for a simulation is performed on just one processor, which then propagates the relevant values to all others. This is common practice, and is mandatory if data is not available on a shared file system.

Combine

The *combine* operations (in MPI jargon *reduce*) perform a distributed arithmetic or logical operation which combines values present on different tasks to give a single instance of the data which is the result, available on a single destination task. A naive implementation would be all tasks sending their data to the destination task, which applies locally the required arithmetic operation; in some sense this is the opposite of a *broadcast*. In our environment we provide the operations of sum, maximum, minimum, maximum absolute value and minimum absolute value; the underlying MPI layer allows very general versions and can be used with any associative arithmetic operation.

The PSCToolkit calling interfaces are:

```
  call psb_sum(icontxt, a, root=root)
  call psb_amx(icontxt, a, root=root, ia=ia)
  call psb_amn(icontxt, a, root=root, ia=ia)
```

where `A` is the integer, real or complex data as before, whereas `root` is the identifier of the destination task. We also have:

```
  call psb_max(icontxt, a, root=root, ia=ia)
  call psb_min(icontxt, a, root=root, ia=ia)
```

similar to the previous, but available only for integer and real data. If `A` is an array, the implied operation is performed element by element, consistently with the Fortran array semantics. The default value is $root = -1$, which is interpreted as meaning that the result should be available to all tasks; this is functionally equivalent to performing a broadcast immediately after the combine operation, although (as noted before) there may exist a faster implementation. Thus the functionality of the sum might be achieved by the following naive implementation:

```
if (root==-1) then
  root_=0
else
  root_=root
end if
if (iam == root_) then
  do i=0,np-1
    if (i /= iam) then
      call psb_rcv(icontxt,tmp,i)
      a = a + tmp
    end if
  end do
else
  call psb_snd(icontxt,a,root_)
end if
if (root==-1) then
  call psb_bcast(icontxt,a,root_)
end if
```

which is, of course, not optimal.

Notice that depending on the data type this collective function will often be based on floating-point operations; since floating-point addition is not associative (see appendix A), a change in the sequence of sends/receive will change the result. This has the implication that it is not reasonable to expect bit-identical results between the naive implementation and an optimized library implementation; another implication is that an operation involving a global sum, such as a dot product, will produce different results when run on a different number of processors on otherwise identical input data. However we should be careful to avoid an unfortunate situation which might exist in some implementations; if a given process issues a receive for the first available message, regardless of the sender process, fluctuations in timings across the network may cause the messages to arrive in a different relative order, thus forcing the floating point operations to be executed in a different order between two successive runs on an otherwise identical machine configuration. While we may accept numerical results (and convergence histories) to vary with the number of processors, since we are effectively using a different machine, it is highly desirable that they should be reproducible once we fix the configuration and the input data: any collective communication implementation ought, at the very least, to provide the means for the user to force a communication pattern that gives reproducible results, if not make it the default.

In the maximum/minimum absolute value case it is possible to specify an optional argument `ia`, an integer of the same rank and size of the data `A`; it will keep track, for each entry of the result `A`, of the identifier of the task the result comes from.

An example of use may be given as follows: find an extravagantly expensive way to compute the sum of the series

$$\sum_{k=1}^{n} k^2.$$

A possible solution consists in launching a parallel job with n tasks, with each one computing one term of the series then delegating the sum to a combine[‡]:

```
call psb_init(icontxt)
call psb_info(icontxt,iam,np)

temp = dble(iam) * dble(iam)
call psb_sum(icontxt,temp)
if (iam == 0) then
   write(6,*) 'total sum for n = ',np,' is ',temp
endif
```

A more realistic example would be the search for the maximum element of a distributed vector to check convergence of an iterative method. If the vector is partitioned among tasks, we can first search for the maximum in the local part on each task, then we apply a combine to get the global maximum; if `root=-1` (as is the default), all tasks will have the (same) result, and will be able to take a coherent decision on whether to proceed or stop the iterations.

5.1.3 Multiple interacting processes: what could possibly go wrong?

How do errors manifest themselves in a message-passing program?

First of all, there are errors detected and reported by the MPI library itself; in most cases these are related to the underlying network, or the operating environment, and there is very little, if anything, to be done at the application level to improve the situation.

Probably more interesting for our readers are programming mistakes in the message-passing application itself; these obviously include the errors that are possible with a serial application. Whether related to the serial part or the message passing statements, you may think of the dreaded `segmentation fault` error message as being a very unfortunate event; however, in some sense it is actually a good thing, in that it is giving a very clear alert that something is deeply wrong.

In practice, there are two things that are more insidious during execution:

1. Race conditions, where the outcome of a code is not fully determined and might change from one run to the next;
2. *Deadlocks*: (a subset of) the processes in the application are waiting for a certain event, but because of the structure of the code this event will never materialize, and no progress is possible.

[‡] Actually the code shown, albeit working, is not a completely correct answer to the problem as stated; are you able to spot the error?

To illustrate a deadlock, let us go back to the example of a simple sequence of send/receive operations. The order of *send/receive* pairs in the source code must take into account the blocking semantics of the receive operations, discussed above; this feature provides an implicit synchronization among processors, but may also cause a *deadlock*. Consider the very simple case in which task 0 must send an integer to task 1, and vice-versa; the most reliable way is the following:

```
if (iam == 0) then
    call psb_snd(icontxt,n,1)
    call psb_rcv(icontxt,k,1)
else if (iam == 1) then
    call psb_rcv(icontxt,n,0)
    call psb_snd(icontxt,k,0)
endif
```

The above code will certainly complete (barring a catastrophic failure in the underlying network or operating system layers).

The following scheme, relying on the “locally blocking” semantics, is also very likely to work:

```
if (iam == 0) then
    call psb_snd(icontxt,n,1)
    call psb_rcv(icontxt,k,1)
else if (iam == 1) then
    call psb_snd(icontxt,k,0)
    call psb_rcv(icontxt,n,0)
endif
```

at least if the operating system does not run out of memory space for pending messages[§].

However if we do write the exchange as follows:

```
if (iam == 0) then
    call psb_rcv(icontxt,k,1)
    call psb_snd(icontxt,n,1)
else if (iam == 1) then
    call psb_rcv(icontxt,n,0)
    call psb_snd(icontxt,k,0)
endif
```

each task will wait for the message coming from the other, and will never reach the call to the send routine(s). We thus have a *deadlock*: we are waiting for an event that cannot possibly happen; the computer is busy, but no useful work is performed.

In CSE applications the most likely error causing this kind of problem is to enter a section of code containing the *send/receive* operations according to a variable whose value is slightly different on different processes: if one process chooses the branch with the receive and the other does *not* choose a branch with a matching send, the first process is blocked indefinitely.

Thus variables that may alter the control flux of a program must be carefully controlled, and preferrably computed through collective operations, so as to make sure that all processes have exactly the same reference value and the selection of different execution paths happens in a coherent fashion.

[§] Admittedly an unlikely event if we are dealing with just a single integer!

5.2 Inter-node parallelism: PGAS

Building Blocks for Linear Algebra Programming

Building Blocks for Dense Linear Algebra 6

6.1 Introduction

To introduce properly the problem at hand, let us consider a simple code for computing the Cholesky factorization $A = U^T U$. It is easy

Algorithm 1: Cholesky factorization

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

to see that this code can be translated into open code in any common programming language. However you may notice that the resulting code has the appearance of "reinventing the wheel", that is, it is but one example of similar code resurfacing over and over again in the implementation of these kind of algorithms.

This observation suggests an immediate strategy:

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

Some languages define some of these as native operators; this is true to a varying extent of Matlab, Fortran and Julia. Writing code with them consists of combining the appropriate sequence of calls to the primitive operators.

It would therefore be appropriate to define a set of functions/subroutines to implement these operators: this would guarantee that the same code is written once but reused multiple times, thereby offering the opportunity to amortize the cost of a quality implementation. We have taken the Cholesky factorization algorithm as a reference for simplicity in presentation; however, the same kind of reasoning applies to many other algorithms, in both dense and sparse linear algebra. Encapsulating the operators inside standardized code and interfaces enables developers to explore alternative implementations while preserving the overall behaviour of the code.

Moreover, restructuring the code in this way may (and does) suggest alternative ways of exploiting the characteristics of computing architectures, such as for example the possibility of using building blocks involving submatrices and not just vectors.

These topics will be explored in the following and in Chapter 7.

6.2 BLAS

The Basic Linear Algebra Subprograms (BLAS) are a set of low-level routines that perform common linear algebra operations such as vector and matrix multiplication. The BLAS is divided into three levels:

- Level 1: Vector operations (e.g., dot product, vector addition)
- Level 2: Matrix-vector operations (e.g., matrix-vector multiplication)
- Level 3: Matrix-matrix operations (e.g., matrix-matrix multiplication)

The BLAS is designed to be efficient and portable, making it a popular choice for high-performance computing applications. The BLAS is often used as a building block for higher-level libraries and applications, such as LAPACK (Linear Algebra PACKage) and ScaLAPACK (Scalable LAPACK). The BLAS is implemented in many programming languages, including C, Fortran, and Python. In this chapter, we will focus on the dense BLAS, which is a set of routines for performing dense linear algebra operations on matrices and vectors.

There are several implementations of the BLAS, including:

OpenBLAS: An open-source implementation of the BLAS and LAPACK libraries.
ATLAS: Automatically Tuned Linear Algebra Software, an open-source implementation of the BLAS and LAPACK libraries.
Intel MKL: A high-performance implementation of the BLAS and LAPACK libraries optimized for Intel processors.
cuBLAS: A GPU-accelerated implementation of the BLAS library for NVIDIA GPUs.
BLIS: A portable and high-performance implementation of the BLAS library.

6.2.1 Level 1 BLAS

Level 1 BLAS routines operate on vectors and include operations such as vector addition, scalar multiplication, and dot products. The following are some of the most commonly used Level 1 BLAS routines:

- `AXPY`: Computes the vector sum $y = \alpha x + y$, where α is a scalar, and x and y are vectors.
- `DOT`: Computes the dot product of two vectors x and y, i.e., $y = x^\top y$.
- `NRM2`: Computes the Euclidean norm of a vector x, i.e., $\|x\|_2 = \sqrt{x^\top x}$.
- `ASUM`: Computes the sum of the absolute values of the elements in a vector x, i.e., $\sum_{i=1}^{n} |x_i|$.

Let us start by writing a simple Fortran program that uses the Level 1 BLAS routines to compute the AXPY operation. The AXPY operation is defined as:

$$y = \alpha x + y$$

where α is a scalar, and x and y are vectors. The following Fortran code demonstrates how to use the AXPY routine from the BLAS library:

```fortran
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
```

The routine which executes the AXPY operation is `daxpy`, where the first argument is the length of the vectors, the second argument is the scalar α, and the third and fourth arguments are the input vectors x and y. The last argument is the increment for the input vectors:

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

Observe also that the AXPY routine is called with the `d` prefix, which indicates that the routine operates on double-precision floating-point numbers. BLAS are strongly typed, and the prefix indicates the type of the data being used[1] To compile this program we need to have acces to a BLAS library. In this case we will use OpenBLAS, which is an open-source implementation of the BLAS and LAPACK libraries[2]. We compile the program using the following command:

1: The other prefixes are:

- s: single precision,
- d: double precision,
- c: complex single precision,
- z: complex double precision.

2: The OpenBLAS library can be installed on most Linux distributions using the package manager, for example on Ubuntu it can be installed using the following command `apt-get install libopenblas-dev`. One can always build OpenBLAS from source, or use Spack to install it (see Appendix B).

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

and we run the program using the following command:

```
./axpy_example
```

The program will output the result of the AXPY operation, which is the vector y after the operation has been performed:

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

AXPY in OpenMP

This is a quite simple routine, and is a good starting point for thinking about exploiting parallelism. Indeed the AXPY operation can be parallelized by splitting the input vectors into chunks and computing the AXPY operation on each chunk in parallel.

In the context of multi-core processors, the AXPY operation can be parallelized using OpenMP. The OpenMP API is a set of **compiler directives**[3], library routines, and environment variables that influence

3: **Compiler directives** are special comments in the source code that are interpreted by the compiler if the parallelization flags are enabled. They are used to specify how the code should be parallelized.

run-time behavior. It is designed for parallel programming in C, C++, and Fortran. OpenMP provides a portable and scalable model for developers of shared memory parallel applications.

Let us first write a simple OpenMP program that computes the AXPY operation in parallel, and then discuss what is happening in the code.

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

This sample program can be compiled using the following command:

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

and then run using the following command:

```
./axpy_omp_example
```

You can compare the ouput of this program with the previous one and observe that the result is the same.

So, *what is happening in the code?* The first thing to notice is that we have included the OpenMP header file

```
use omp_lib
```

This header file contains the definitions of the OpenMP routines and directives. The next thing to notice is the `!$omp parallel do` directive, which tells the compiler to parallelize the following loop. But how things are being parallelized? The `!$omp parallel do` directive tells the compiler to spawn a team of *threads*[4] and distribute the loop iterations among them. of the loop among them.

4: In OpenMP, a **thread** is an independent execution unit within a program that can run concurrently with other threads. Each thread has its own execution context, including a program counter and machine registers, but shares the process's address space and resources. Threads also have execution-specific attributes, such as the cores they can utilize and performance metrics like CPU time usage.

But how many threads are created? The number of threads created is determined by the `OMP_NUM_THREADS` environment variable, which can be set before running the program, for example to set the number of threads to 4, we can run the following command:

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

The number of threads can also be set in the code using the OpenMP function: `omp_set_num_threads`, observe that for some task we may have a limitation of the number of usable threads, hence it may be useful to set the number of threads in the code instead of using the environment variable.

A good idea would be for our test program of printing in input the number of threads being used so that we can see how many threads are being used. We can do this by adding the following code to our program:

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

This code uses the `omp_get_num_threads` routine to get the number of threads, observe that we need to use the *`!$omp parallel`* directive to create a parallel region and then use the *`!$omp end parallel`* directive to end the parallel region to ask for the number of threads. Moreover, it uses the *`!$omp single`* directive since there is no need for all threads to repeatedly update the variable `nthreads`.

There are two important questions that we need to answer:

1. How the threads are scheduled?
2. Who owns what data?

The OpenMP API provides several scheduling policies to control how the iterations of the loop are distributed among the threads. The scheduling policies are specified using the `schedule` clause on the parallel directive:

- `static`: Iterations are divided into chunks of equal size, and each thread is assigned a chunk. This is the default scheduling policy. It is predictable and easy to understand, but may not be the most efficient for all loops.
- `static, chunk_size`: Similar to `static` scheduling, but the size of the chunks can be specified. This provides more control over the distribution of iterations.
- `dynamic`: Iterations are assigned to threads as they become available. This allows for load balancing among threads and can improve performance for irregular loops, but it may introduce overhead due to scheduling and make the code less predictable.
- `guided`: Similar to `dynamic` scheduling, but the size of the chunks decreases over time. This improves load balancing while reducing scheduling overhead.

- `runtime`: The scheduling policy is determined at runtime based on the value of the `OMP_SCHEDULE` environment variable. This provides flexibility in choosing the scheduling policy without modifying the code.
- `auto`: The scheduling policy is determined by the compiler. This allows the compiler to optimize the scheduling based on the specific loop and system characteristics.

As an example, consider the following scheduling policy for the AXPY operation:

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

where `chunk_size` is the size of the chunks assigned to each thread.

The OpenMP API provides several clauses to control the visibility of data between threads, which can be specified on a parallel directive:

- **shared:** A variable declared as `shared` is visible to all threads in the team. There is a single instance of the variable that each thread can access.
- **private:** A variable declared as `private` is only visible to a single thread. Each thread creates its own uninitialized instance of the variable, and only the creating thread has that instance in scope.
- **firstprivate:** Similar to `private`, but the private instance is initialized with the value of the original variable from the thread that created the parallel region.
- **lastprivate:** Each thread has a private instance of the variable. At the end of the parallel region, the variable in the thread that created the parallel region is updated with the value from the private instance of the thread that executed the last iteration of the loop or the lexically last section.

In our example program, the variables which are shared between the threads are the input vectors x and y, and the scalar α, while the i variable has a single value for each thread. Hence we can be more precise and rewrite the OMP directive as:

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

If we compile and run this program, we will see that the result is the same as before.

Let us try to write a version of these codes that we can use to **measure performance**. As a firt time measuring step we will use the `cpu_time` function, which returns the CPU time in seconds. To get a more reliable measure of the time taken we need to do the following:

- Run the code several times and take the average time.
- Use a large enough problem size to get a reliable measure of the time taken.

Let us start by modifying the code using BLAS to read the size of the problem from the command line and add the time measurement code. The following code does this:

```
program axpy_blas_time
    use iso_fortran_env, only: int64, real64, output_unit, &
        error_unit
    implicit none
    integer(kind=int64) :: n
    real(kind=real64), dimension(:), allocatable :: x, y
    character(len=100) :: arg
    real(kind=real64) :: alpha, t1, t2, elapsed_time, &
        average_time
    integer(kind=int64) :: i,info

    ! Read the size of the vector from command line arguments
    if (command_argument_count() < 1) then
        write(error_unit, '("Usage: ./axpy_blas_time <size>")')
        stop
    end if
    ! Read the size of the vector
    call get_command_argument(1, arg)
    read(arg, *) n
    ! Allocate the vectors
    allocate(x(n), y(n), stat=info)
    if (info /= 0) then
        write(error_unit, '("Error allocating memory")')
        stop
    end if
    ! Initialize the vectors and scalar
    x = [(real(i), i=1,n)]
    y = [(sqrt(real(i)), i=1,n)]
    alpha = 2.0
    ! Call the AXPY routine many times
    do i = 1, 1000
        elapsed_time = 0.0
        call cpu_time(t1) ! Start the timer
        call daxpy(n, alpha, x, 1, y, 1)
        call cpu_time(t2) ! Stop the timer
        elapsed_time = elapsed_time + (t2 - t1)
    end do
    ! Calculate the average time
    average_time = elapsed_time / 1000.0
    ! Print the elapsed and average time:
    write(output_unit, '("Elapsed time: ", 1PE12.6, " s")') &
        elapsed_time
    write(output_unit, '("Average time: ", 1PE12.6, " s")') &
        average_time

    ! Deallocate the vectors
    deallocate(x, y, stat=info)
    if (info /= 0) then
        write(error_unit, '("Error deallocating memory")')
        stop
    end if
    return
end program axpy_blas_time
```

The **`allocate`** routine is used to allocate memory for the **`allocatable`** arrays. An allocatable array is an object that can be dynamically associated with memory, but such that it always has a well-defined status (either allocated or not); together with some other semantics features, it is guaranteed never to generate a *memory leak*. The size of the array is determined by the user-provided input `n`, which is read from the command-line arguments. The `stat` argument in the **`allocate`** statement is used to capture any errors during memory allocation. If the allocation fails, the program writes an error message to the `error_unit` and stops execution. The **`deallocate`** routine is used to release the memory allocated for the arrays when they are no longer needed. Similar to allocation, the `stat` argument is used to check for errors during deallocation. If deallocation fails, an error message is written, and the program stops execution. The program initializes the `x` and `y` arrays with specific values and performs the AXPY operation multiple times to measure its execution time. The `cpu_time` intrinsic is used to measure the time taken for the operation. The average execution time is calculated and printed to the output.

To compile the code we use the following command:

```
gfortran -O3 -march=native -mtune=alderlake -o axpy_blas
 ↪  axpy_blas_time.f90 -lopenblas
```

differently from before, since we want to try and measure the performance of the code, we activate the optimization flags. We are going to run this example on a Laptop which has a Intel® Core™ i9-14900HX processor. To set the *tuning* flags for the compiler to the right architecture we run the command:

```
gcc -mtune=native -Q --help=target|grep mtune
```

and we get the following output:

```
-mgather                          -mtune-ctrl=use_gather
-mscatter                         -mtune-ctrl=use_scatter
-mtune-ctrl=
-mtune=                           alderlake
```

The `-mtune=alderlake` flag is the one we are looking for, and we can use it to set the tuning flags for the compiler, different processors will have different tuning flags, and you can find the right one for your processor using a combination of the command above and a quick search on the internet.

Having done this, we can now adapt the version using OpenMP to measure also the relevant performance metrics. To this end it suffices to `use omp_lib` code, and the OpenMP directives we have seen before:

```fortran
do i = 1, 1000
    elapsed_time = 0.0
    t1 = omp_get_wtime() ! Start the timer
    ! Write the OpenMP directive to parallelize the for loop
    !$omp parallel do shared(x, y, alpha) private(j)
    ↪  schedule(static)
    do j = 1, n
        y(j) = alpha * x(j) + y(j)
    end do
    !$omp end parallel do
    t2 = omp_get_wtime() ! Stop the timer
    elapsed_time = elapsed_time + (t2 - t1)
end do
```

As before, we can compile the code by doing

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

We are now ready to run the code and measure the performance. To do this a good idea is to write a small script that runs the code for different thread counts and problem sizes and writes the results to a file. The following script does this:

```bash
#!/bin/bash
# This script runs the AXPY BLAS benchmark with different
# sizes and threads. The axpy_blas runs the BLAS (OpenBLAS)
# implementation of the AXPY operation. The axpy_omp
# runs the OpenMP implementation of the AXPY operation.
module load openblas
SIZE=1000000
# Run the BLAS version
./axpy_blas ${SIZE} >> axpy_blas.log 2>&1
```

```
# Run the OpenMP version with different thread counts
for threads in 1 2 4 8 16 32; do
    export OMP_NUM_THREADS=$threads
    ./axpy_omp ${SIZE} >> axpy_blas.log 2>&1
done
```

We can run it using the following command:

```
chmod +x axpy_blas.sh
./axpy_blas.sh
```

The script will create a file called `axpy_blas.log` with the results of the runs. To analyze the results we can write a small Python script that reads the file and plots the results, the following script does this:

```
import re
import matplotlib.pyplot as plt
import numpy as np

filename = "code/axpy_blas.log"
# Lists to hold extracted values
elapsed_times = []
average_times = []
# Regular expression pattern for scientific notation floats
pattern = r"([-+]?\d*\.\d+E[-+]\d+)"
# Read and parse the file
with open(filename, 'r') as file:
    for line in file:
        match = re.search(pattern, line)
        if match:
            value = float(match.group(1))
            if "Elapsed time" in line:
                elapsed_times.append(value)
            elif "Average time" in line:
                average_times.append(value)

# Custom x-axis labels
x_labels = ["BLAS", "1", "2", "4", "8", "16", "32"]
indices = np.arange(len(x_labels))
# Create side-by-side bar plots
fig, axes = plt.subplots(2, 1, figsize=(14, 5))
# Elapsed time bar plot
axes[0].bar(indices, elapsed_times, color='skyblue')
axes[0].set_title("Bar Plot of Elapsed Times")
axes[0].set_xlabel("Thread Count")
axes[0].set_ylabel("Time (s)")
axes[0].set_xticks(indices)
axes[0].set_xticklabels(x_labels)
# Average time bar plot
axes[1].bar(indices, average_times, color='salmon')
axes[1].set_title("Bar Plot of Average Times")
axes[1].set_xlabel("Thread Count")
axes[1].set_ylabel("Time (s)")
axes[1].set_xticks(indices)
axes[1].set_xticklabels(x_labels)

plt.tight_layout()
plt.show()
# Save the figure to a file as EPS
fig.savefig("axpy_times.eps", format='eps')
```

We can run it using the following command:

```
python3 axpy_time.py
```

The script will create a file called `axpy_time.eps`, which we have included in Figure 6.1. The plot shows the performance of the AXPY operation

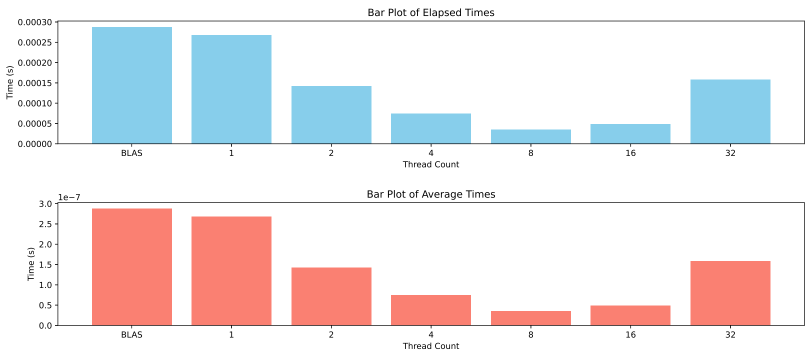

Figure 6.1: Performance of the AXPY operation using OpenMP and BLAS

using BLAS and OpenMP with different number of threads. The x-axis shows if the problem has been solved by BLAS or by the OpenMP implementation with a given number of threads, while the y-axis shows the time taken to compute the AXPY operation in seconds.

From this experiment we observe that the performance of the AXPY operation using 8 threads is the one that achieves better performance for this problem size. In all cases the performance of the OpenMP implementation is better than the BLAS implementation. Although for this example we cheated a little, since we used a version of OpenBLAS installed via Spack and which does not have multithreading enabled.

Exercise 6.2.1 Adapt the code we have already written to run on the **Toeplitz cluster**. This means selecting the right compiler and the right flags to compile the code from the available nodes. You can use the `module` command to load the right compiler and the right flags. You can use the `module avail` command to see the available modules. You can use the `module load` command to load the right module.

Then to run the code you have to produce a job script that will run the code on the cluster. You can use the `sbatch` command to submit the job script to the cluster. The job script will have to be a job script to run on a single node of the cluster, and a single task, with a number of cpus per task equal to the number of threads you want to use.

Adapt the Python script to read the output of the job script and plot the results.

Exercise 6.2.2 Explore the different scheduling policies available in OpenMP and how they affect the performance of the AXPY operation. You can do this by modifying the scheduling policy in the OpenMP directive and measuring the performance of the AXPY operation.

The dot and the nrm2 operations

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, and it is defined as the sum of the products of the corresponding elements of the two vectors. The dot product can be computed using the BLAS routine `ddot`, which computes the dot product of two vectors x and y of size n:

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

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

What kind of parallelism can we exploit in this case? The dot product is a what is called a **reduction** operation, which means that we can compute the dot product in parallel by splitting the input vectors into chunks and computing the dot product on each chunk in parallel. The results of the dot products on each chunk can then be added together to get the final result. This is a good example of a reduction operation that can be parallelized using OpenMP. The following code shows how to do this[5]:

5: Observe that in the code we have added a declaration for the variable `ddot` as a `real(real64)` variable, which is the type of the result of the `d` variant of the dot product routine. This is due to the fact that `ddot` is a function that returns a value. We need to let the compiler know what to expect when it sees the `ddot` function. This is also due to the fact that BLAS and their interfaces comes from before the Fortran 90 standard, and hence they do not have a module we can use to import the routines. The same will apply in the following for the `dnrm2` routine, which is the double precision version of the 2-norm routine. Try to run the code without the declaration of the variable `ddot` and see what happens. You will get a compilation error, since the compiler does not know what to expect when it sees the `ddot` function.

```
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()
    !$omp end single
    !$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()
    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
```

The code uses again the `!$omp parallel do` directive to create a parallel region and distribute the iterations of the loop among the threads, the *reduction clause* is used to specify that the variable `c` is a reduction variable, which means that each thread will have its own private copy of the variable `c`, and at the end of the parallel region the values of the private copies will be added together to get the final result. We have used again the `!shared(x,y)` and `!private(i)` clauses to specify the visibility of the data between the threads.

The code for the computation of the 2-norm of a vector which is defined as:

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

is similar, we can use the `dnrm2` routine to compute the 2-norm of a vector x of size n:

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

where `incx` is as usual the increment for the input vector x. Also the implementation using OpenMP is similar to the one we have seen before, it suffices to change the **do** loop as

```
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)
```

Exercise 6.2.3 Write a program analogous to the one computing the dot product to compute instead the 2-norm of a vector using OpenMP and the BLAS routine `dnrm2` to verify the results. Measure the performance of the code and compare it with the performance of the BLAS implementation. You can use the same script we have used to measure the performance of the AXPY operation to measure the performance of the 2-norm operation. You can again plain around with the scheduling policies to see how they affect the performance of the code.

All the other Level 1 BLAS operations

In addition to the BLAS we have discussed, the 1st level BLAS operations include the operations reported in Table 6.1. The operations are all vector operations, and they are all defined in the BLAS standard.

Table 6.1: Level 1 BLAS 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	$x^\top y$	$2n$	$2n$
c, z	`cdotu(n, x, incx, y, incy)`	(complex)	$x^\top y$	$2n$	$2n$
c, z	`cdotc(n, x, incx, y, incy)`	(complex conj)	$x^H y$	$2n$	$2n$
sds, ds	`dot(n, x, incx, y, incy)`	(internally double precision)	$x^\top y$	$2n$	$2n$
s, d, sc, dz	`nrm2(n, x, incx)`	2-norm	$\|x\|_2$	$2n$	n
s, d, sc, dz	`asum(n, x, incx)`	1-norm	$\|x\|_1$	n	n
s, d, c, z	`iamax(n, x, incx)`	∞-norm	$\|x\|_\infty$	n	n
s, d, c, z	`rotg(a, b, c, s)`	generate plane (Givens') rotation (c real, s complex)		$O(1)$	$O(1)$
s, d, c, z + t	`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		$6n$	$2n$

6.2.2 Level 2 BLAS

The level 2 BLAS define operators that involve vectors and matrices, such as

GEMV : Computes the vector sum $y = \alpha Ax + \beta y$, where α, β are scalars, x and y are vectors. and A is a two-dimensional matrix;

GER : Computes the rank-1 update $A = \alpha xy^\top + A$, where α is a scalar, x, y are vectors and A is a two-dimensional matrix;

TPSV and [TRSV]: Solve a triangular system of equations $Ax = b$, where A is a triangular matrix and b is a vector;

If the vectors involved are of size n and the matrices of size $n \times n$, then the level 2 BLAS operators involve $O(n^2)$ arithmetic operations, hence the name.

Let us try to write a simple Fortran program that uses the Level 2 BLAS routines to compute the GEMV operation. The GEMV operation, in its general form, is defined as:

$$y = \alpha Ax + \beta y \tag{6.1}$$

where α and β are scalars, and A is a matrix of size $m \times n$, x is a vector of size n, and y is a vector of size m.

The following Fortran code demonstrates how to use the GEMV routine from the BLAS library:

```
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
    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)
    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
        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
    end if
end program gemv_blas
```

The routine which executes the GEMV operation is `dgemv`, where the first argument is the transposition flag, the second argument is the size of the matrix, the third argument is the size of the vector, the fourth argument is the scalar α, and the fifth and sixth arguments are the input matrix A and vector x. The seventh argument is the increment for the input vector x, and the eighth argument is the scalar β. The ninth argument is the input vector y, and the last argument is the increment for the input vector y:

```fortran
call dgemv('N', n, m, alpha, A, lda, x, incx, beta, y,
↪  incy)
```

Observe also that the GEMV routine is called with the d prefix, which indicates that the routine operates on double-precision floating-point numbers. The lda argument is the leading dimension of the matrix A, which is the size of the first dimension of the array that stores the matrix A. The leading dimension is used to specify the memory layout of the matrix, and it is used to access the elements of the matrix in memory, i.e., it needs to be the actual size of the first dimension of the array that stores the matrix A.

Let us look at how the GEMV operation can be parallelized employing OpenMP directives[6].

6: The next example of codes are different implementations of the GEMV routine, it could be a good idea to creaet a Fotran **module** which contains them, and then use the **use** statement to import the module in the code that will run the performance (and correctness) tests. This will allow us to have a cleaner code, and to reuse the code in different programs. The module can be created using the **module** statement and the **end module** statement.

Listing 6.1: Example of Fortran module to house the GEMV operation.

```fortran
module gemvmod
    use iso_fortran_env
    use omp_lib
    implicit none
    private
    public :: ! subroutine names
contains
    ! Here we define the
    ! subroutine that will
    ! compute the GEMV
    ! operation and whose
    ! name will be put
    ! after the public
    ! statement
end module gemvmod
```

First we need to write (6.1) in a form that is more suitable for parallelization:

$$y_i = \alpha \sum_{j=1}^{n} A_{ij} x_j + \beta y_i \tag{6.2}$$

where $i = 1, \dots, m$ and $j = 1, \dots, n$. The first idea we may have is to parallelize the outer loop, i.e., the loop over i. This seems a good idea, since the computation of y_i does not depend on the computation of y_j for $j \neq i$, using the OpenMP directive we have alread seen we can write a subroutine:

```fortran
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, *)
    real(real64), intent(in) :: x(*)
    real(real64), intent(inout) :: y(*)
    real(real64) :: ddot
    integer :: i
    real(real64) :: temp
    !$omp parallel do private(i,temp) shared(m, n, A, x, y, alpha,
    ↪  beta)
    do i = 1, m
        temp = ddot(n, A(i,1:n), 1, x, 1)
        y(i) = alpha * temp + beta * y(i)
    end do
    !$omp end parallel do
end subroutine gemv_openmp_n
```

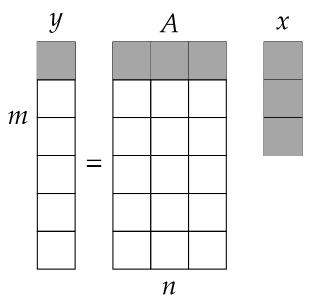

Figure 6.2: The GEMV operation. The matrix A is of size $m \times n$, the vector x is of size n, and the vector y is of size m. The shaded cells are the ones that are accessed in the computation of the GEMV operation.

The question is: how are we accessing the memory? We are processing the memory in n-blocks with a stride of m—see Figure Figure 6.2—which is suboptimal as it leads to inefficient memory bandwidth utilization. The vector y is accessed sequentially, allowing it to be stored in registers. If n

is small, the vector x can be reused efficiently at the cache level. However, due to the irregular access pattern of the matrix A, this approach is not well-suited for column-major matrices, which are the default storage format in Fortran. To improve performance, we can start by considering a different scheduling and blocking policy while first maintaining the same looping structure:

```fortran
subroutine gemv_openmp_n_block(m, n, alpha, A, lda, x, beta, y)
    use, intrinsic :: iso_fortran_env, only: real64
    use omp_lib
    implicit none
    integer, intent(in)          :: m, n
    real(real64), intent(in)     :: alpha, beta
    real(real64), intent(in)     :: A(lda, *)
    integer, intent(in)          :: lda
    real(real64), intent(in)     :: x(*)
    real(real64), intent(inout)  :: y(*)
    ! Local variables
    integer :: i, j
    real(real64) :: temp
    real(real64) :: ddot
    !$omp parallel do schedule(dynamic,32) private(i,j,temp)
    ↪   shared(m,n,A,x,y,alpha,beta)
    do i = 1, m
        temp = ddot(n, A(i,1:n), 1, x, 1)
        y(i) = alpha * temp + beta * y(i) ! scale-add update
    end do
    !$omp end parallel do
end subroutine gemv_openmp_n_block
```

The code above uses the `!$omp schedule` directive to specify that the iterations of the loop over i should be scheduled dynamically in chunks of 32 iterations. This means that each thread will process a chunk of 32 iterations at a time, and when it finishes processing the chunk, it will request another chunk of 32 iterations to process. This allows the threads to balance the workload dynamically, and it can improve the performance of the code. The choice of the chunk size is important, and it can affect the performance of the code. A chunk size that is too small can lead to overhead due to the scheduling of the threads, while a chunk size that is too large can lead to load imbalance among the threads due to caching effects. The optimal chunk size depends on the size of the problem and the size of the cache. One can experiment with different chunk sizes to find the optimal one for a given problem size/machine configuration.

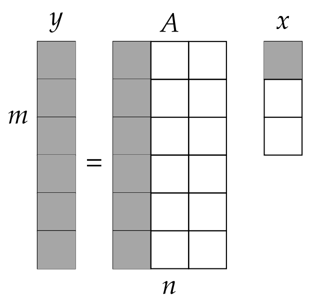

Figure 6.3: The GEMV operation. The matrix A is of size $m \times n$, the vector x is of size n, and the vector y is of size m. The shaded cells are the ones that are accessed in the computation of the GEMV operation.

An alternative approach involves swapping the order of the loops, effectively computing the entries by m-blocking—see Figure Figure 6.3. Since 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. This approach is particularly well-suited for GEMV operations with small m, as it takes advantage of the memory hierarchy for better performance, this can be coded as:

```fortran
subroutine gemv_openmp_m(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, *)
    real(real64), intent(in) :: x(n)
    real(real64), intent(inout) :: y(m)
    integer :: i
    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
end subroutine gemv_openmp_m
```

The third approach would be to tile the matrix A into blocks of size $n_x \times n_y$, and then compute the GEMV operation on each block; see Figure Figure 6.4. 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 for this approach is similar to the one we have seen before, but we need to add an outer loop that iterates over the blocks of the matrix A. The code for this approach is a simple example of divide-and-conquer approach, where we divide the problem into smaller subproblems and solve each subproblem by a call to the sequetial GEMV operation. An example of this approach can be coded as[7]:

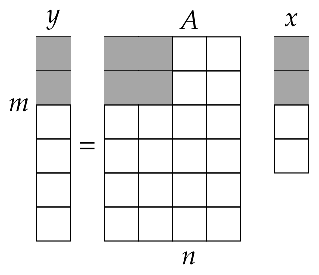

Figure 6.4: The GEMV operation. The matrix A is of size $m \times n$, the vector x is of size n, and the vector y is of size m. The shaded cells are the ones that are accessed in the computation of the GEMV operation.

7: Observe that in general the tile sizes n_x and n_y may not divide exactly the matrix size m and n, hence we need to handle the case where the last tile is smaller than n_x and n_y. This can be done by using the min function to compute the size of the last tile.

```
subroutine gemv_openmp_tiled(m, n, alpha, A, lda, x, beta, y, n_x,
↪  n_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, *)
    real(real64), intent(in) :: x(n)
    real(real64), intent(inout) :: y(m)
    integer, intent(in), optional :: n_x, n_y
    real(real64), allocatable :: yloc(:)
    ! local variables
    integer :: n_x_, n_y_
    integer :: i, j, ti, mb, nb
    ! set tile sizes or defaults
    if (.not. present(n_x)) then
        n_x_ = 32
    else
        n_x_ = n_x
    end if
    if (.not. present(n_y)) then
        n_y_ = 32
    else
        n_y_ = n_y
    end if
    ! 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,tj,mb,nb,yloc)
        allocate(yloc(m))
```

We have used in the code the **optional** keyword in the subroutine argument list, which allows us to specify that the arguments n_x and n_y are optional. This means that we can call the subroutine without specifying these arguments, and the default values will be used. To check if the arguments are present, we can use the `present` function, which returns `.true.` if the argument is present, and `.false.` otherwise. Another important point we had alread remarked is that the BLAS DGEMV routine expects its LDA argument to be the first dimension of the original array `A(lda,*)`, not the local tile height, since this is used internally to ensure that each tile addresses memory correctly.

```
        yloc = 0.0_real64
        ! Tile the i-j loops; collapse for better load balance
        !$omp do collapse(2) schedule(static)
        do i = 1, m, n_x_
            do j = 1, n, n_y_
                mb = min(n_x_, m - i + 1) ! handle edge tiles
                nb = min(n_y_, n - j + 1)
        ! perform the small GEMV into the thread-local yloc
                call dgemv('N', mb, nb, alpha, &
                            A(i, j), lda, &
                            x(j), 1, &
                            1.0_real64, yloc(i), 1)
            end do
        end do
        !$omp end do
        ! Safely accumulate thread-local yloc into global y
        do ti = 1, m
            !$omp atomic
            y(ti) = y(ti) + yloc(ti)
        end do
        deallocate(yloc)
    !$omp end parallel
end subroutine gemv_openmp_tiled
```

With respect to the previous example, we have exploited new OpenMP directives, first we have used the `!collapse(2)` directive, which allows us to collapse the two loops into a single loop, this allows us to have a better load balance between the threads, and to use a better scheduling policy. We have also used the `!omp atomic` directive, which allows us to safely update the global vector `y` with the local vector `yloc`. This is important since we are updating the global vector `y` in parallel, and we need to ensure that the updates are done in a safe manner. The `!omp atomic` directive ensures that the update is done in a safe manner, and that the updates are done in the correct order.

Observe also that for the default tile sizes we have used the values 32 for both n_x and n_y, the way in which this value should be set is not trivial, and follows consideration about the cache size and the memory hierarchy of the machine. The idea is to use a tile size that is small enough to fit in the cache, but large enough to allow for good memory access patterns. The tile size should be chosen based on the size of the cache, the size of the matrix, and the size of the vector. As a first approximation, 32 is usually a good guess, but if your objective is to squeeze the last drop of performance out of your machine, you should experiment with different tile sizes to find the optimal one for your machine.

Listing 6.2: Forward substitution algorithm.

```
subroutine fwd_subs(A, b, x)
implicit none
integer :: i, j
real(real64), intent(in) ::
↪  A(:,:)
real(real64), intent(in) :: b(:)
real(real64), intent(out) ::
↪  x(size(b))
integer :: n
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
```

Exercise 6.2.4 Write a program that uses the BLAS GEMV routine to compute the GEMV operation, and compare the performance of the BLAS implementation with the performance of the OpenMP implementations. You can adapt the same Python script we have used to measure the performance of the AXPY operation to measure the performance of the GEMV. It is also a good idea to play around with the scheduling policies to see how they affect the performance of the code.

Not all the Level 2 BLAS operations are amenable to parallelization, for example the `trsv` operation, which solves the system of equations

$$Ax = b, \quad A = \begin{bmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ a_{21} & a_{22} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix} \text{ or}$$

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a_{22} & 0 & \cdots & 0 \\ 0 & 0 & a_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

where A is a triangular matrix, x is the solution vector, and b is the right-hand side vector. The `trsv` operation is not parallelizable since the solution of the system of equations is done in a sequential manner, i.e., we need to solve the system of equations for x_i before we can solve the system of equations for x_{i+1}—if the matrix A is lower triangular Listing 6.2—-or we need to solve the system of equations for x_i before we can solve the system of equations for x_{i-1}—if the matrix A is upper triangular Listing 6.3. The are options for treating the case of sparse (lower or upper) triangular matrices via iterative methods, or other specialized algorithms. We will come back to this in Chapter 7 on page 87.

In general, when we have to design an algorithm which we expect to be parallelizable, it is a good idea to avoid as much as possible solution of triangular systems since this is a classical example of bottleneck in parallel algorithms.

All the other Level 2 BLAS operations

As we have seen, expressing parallelism at the level of the Level 2 BLAS is already more challenging than at the level of the Level 1 BLAS. The reason is that the Level 2 BLAS operations are all matrix-vector operations, and the dependencies between the elements of the matrix and the vector are more complex than the dependencies between the elements of the vector in the Level 1 BLAS operations, i.e., this gave us the opportunity to discuss the treatment of different data layouts, and the impact of the memory hierarchy on the performance of the code.

The level 2 BLAS operations include more operations than the one we have discussed, as we have done for the Level 1 BLAS, we report in Table Table 6.3 on the following page the full list of operations. As you can see, the level 2 BLAS operations are all matrix-vector operations and are all of quadratic complexity.

We also need to mention the existence of a special case of the level 2 BLAS which involves the use of band storage. This is a special case of the level 2 BLAS which is used to store banded matrices. In BLAS, packed storage formats are used for *banded* and *triangular banded* matrices to improve efficiency by **storing only the nonzero elements**. A banded matrix has nonzero entries confined to a diagonal band around the main diagonal. If a matrix has kl sub-diagonals (below the main diagonal) and ku super-diagonals (above the main diagonal), it is stored in a two-dimensional array of size $(kl + ku + 1) \times n$, where n is the number of columns. The

Table 6.3: Level 2 BLAS Operations

types	name (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 + t	`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.	$x = A^T x$	n^2	$n^2/2$
s, d, c, z	`trsv(uplo, trans, diag, n, A, lda, x, incx)`	triangular solve	$x = A^{-1} x$	n^2	$n^2/2$
s, d, c	`ger(m, n, alpha, x, incx, y, incy, A, lda)`	general rank-1 update	$A = A + \alpha x y^T$	$2mn$	mn
s, d	`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 + t	`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	$A = A + \alpha x y^H + y(x^H)$	$2n^2$	$n^2/2$

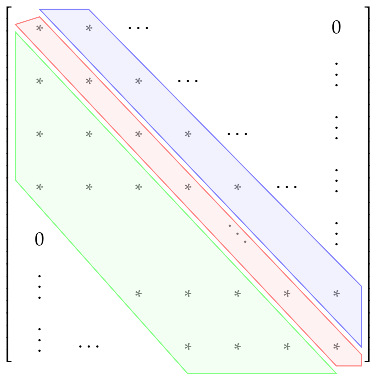

Figure 6.5: Banded matrix with kl sub-diagonals and ku super-diagonals.

main diagonal is stored in row ku of the packed array. Super-diagonals are stored above row ku, and sub-diagonals below. For a matrix element $A(i, j)$, the corresponding element in the packed array AB is located at AB$(ku + i - j, j)$.

A triangular banded matrix is either upper or lower triangular with a specified bandwidth k. For an upper triangular banded matrix, the main diagonal and up to k super-diagonals are stored; for a lower triangular banded matrix, the main diagonal and up to k sub-diagonals are stored. The storage array has dimensions $(k + 1) \times n$. In the upper triangular case, the packed mapping is AB$(k + i - j, j)$ for $i \leq j$ and $j - i \leq k$; in the lower triangular case, it is AB$(i - j, j)$ for $i \geq j$ and $i - j \leq k$. In **??** we show the list of the level 2 BLAS operations that are defined for banded matrices and require the use of packed storage.

6.2.3 Level 3 BLAS

The level 3 BLAS define operators that involve matrices, such as

GEMM : Computes the matrix $C = \alpha AB + \beta C$, where α, β are scalars, and A, B, C are matrices;

SYR2K : Computes the symmetrix rank-2 update $C = \alpha AB^\top + \alpha BA^\top + \beta C$, where α is a scalars and A, B, C are matrices;

If the matrices involved are of size $n \times n$, then the level 3 BLAS operators involve $O(n^3)$ arithmetic operations (given $O(n^2)$ accesses to data), hence the name.

We are going to look into the GEMM operation in more detail, since it is the one which is most amenable to parallelization, and it is one of the most used in practice. As we have done for the level 2 BLAS GEMV operation, let us start from the mathematical formulation of the GEMM operation, which is given by

$\beta = 0.0$, and the result is stored in C. The following code shows how to call the GEMM operation in Fortran:

```fortran
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
    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
    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
```

The code is very simple, and it is similar to the one we have seen for the GEMV operation, it also uses the `dgemm` routine, which is the one that implements the GEMM operation in the BLAS library and is called as follows:

```fortran
call dgemm(transa, transb, m, n, k, alpha, A, lda, B, ldb, beta,
↪  C, ldc)
```

where `transa` and `transb` are the transposition options for the matrices A and B, respectively, `m`, `n`, and `k` are the dimensions of the matrices, `alpha` and `beta` are the scalars, and `lda`, `ldb`, and `ldc` are the leading dimensions of the matrices A, B, and C, respectively. The leading dimension is the first dimension of the array that stores the matrix in memory.

To see how the GEMM operation can be parallelized, we need to look at the mathematical formulation of the GEMM operation, which is given by

$$C_{ij} = \alpha \sum_{l=1}^{k} A_{il} B_{lj} + \beta C_{ij}, \quad i = 1, \ldots, m, j = 1, \ldots, n. \tag{6.4}$$

First of all, let us express the GEMM operation as a looped operation plainly in Fortran:

```fortran
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
real(real64), intent(in) :: A(n,k)
real(real64), intent(in) :: B(k,m)
real(real64), intent(in) :: beta
real(real64), intent(inout) :: C(n,m)
! Local variables
integer :: i, j, l
real(real64) :: sum
! Matrix multiplication
! C = alpha * A * B + beta * C
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
```

The above code is a straight formula-to-code implementation of the GEMM operation as we have writte in (6.4). If we look at the code, we can observe that we can swap the order of the outer two loops, i.e., we can loop over the columns of the matrix C first, and then over the rows of the matrix C, i.e.,

```fortran
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
real(real64), intent(in) :: A(n,k)
real(real64), intent(in) :: B(k,m)
real(real64), intent(in) :: beta
real(real64), intent(inout) :: C(n,m)
! Local variables
integer :: i, j, l
real(real64) :: sum
! Matrix multiplication
! C = alpha * A * B + beta * C
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
```

Do you think this is a good idea? Let us try and measure the performance of the two variants of the code. We can write a simple wrapper calling the two subroutines on three randomly generated matrices, and measure the time over 100 repetitions:

```
    program matmul_sequential
    use iso_fortran_env, only: real64, output_unit
    implicit none
    integer, parameter :: n = 1000
    real(real64) :: A(n,n), B(n,n), C(n,n)
    real(real64) :: alpha, beta
    real(real64) :: start_time, end_time, elapsed_time
    integer :: rep
    ! Initialize matrices A and B
    call random_number(A)
    call random_number(B)
    call random_number(C)
    ! Set alpha and beta
    alpha = 1.0d0
    beta = 1.0d0
    ! Perform matrix multiplication with matmul
    elapsed_time = 0.0d0
    do rep = 1, 100
        call cpu_time(start_time)
        call matmul_ijl(n,n,n,alpha,A,B,beta,C)
        call cpu_time(end_time)
        elapsed_time = elapsed_time + (end_time - start_time)
    end do
    write(output_unit,*) 'Average time for matmul_ijl:',
    ↪  elapsed_time/100.0d0
    elapsed_time = 0.0d0
    do rep = 1, 100
        call cpu_time(start_time)
        call matmul_jil(n,n,n,alpha,A,B,beta,C)
        call cpu_time(end_time)
        elapsed_time = elapsed_time + (end_time - start_time)
    end do
    write(output_unit,*) 'Average time for matmul_jil:',
    ↪  elapsed_time/100.0d0
    ! Exit the program with a success status
    stop
end program matmul_sequential
```

Since we are looking for performance, we compile the code with the `-O3` optimization flag, and the `-mtune=native` and `-march=alderlake` flags, which are used to optimize the code for the specific architecture of the machine we are using. After running the code, we get the following output:

```
Average time for matmul_ijl:  0.37526105999999992
Average time for matmul_jil:  0.15405741999999961
```

As you can see, the second version of the code is about 2.6 times faster than the first one. Indeed, in the second version of the code, we have improved the memory access pattern by accessing the elements of the matrix C in a column-major order, which is the same order in which the elements are stored in memory. As we have discussed in Section 4.1, modern CPUs fetch memory in cache lines and benefit when loops stride through memory sequentially. In this regard, the second version of the code is better than the first one, nevertheless, we can still improve the performance of the code by using a yet better memory access pattern. If we select the (j, l, i)-ordering of the loops:

- The innermost loop is i, so each iteration reads $A(i, l)$ and $C(i, j)$ contiguously (first index varies);
- $B(l, j)$ is constant inside the inner loop (with fixed l and j), so it can be held in a register (or broadcast) while sweeping through a whole column of A and C;
- Over l (middle loop), $A(:, l)$ (column of A) and $B(:, j)$ (column of B) are accessed sequentially; both are contiguous in memory for fixed j;
- Over j (outer loop), each column of C is computed in turn.

Namely, we can write the code as follows:

```fortran
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
    real(real64), intent(in) :: A(n,k)
    real(real64), intent(in) :: B(k,m)
    real(real64), intent(in) :: beta
    real(real64), intent(inout) :: C(n,m)
    ! Local variables
    integer :: i, j, l
    ! Matrix multiplication
    C = beta * C
    ! Compute C += alpha * A * B
    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
```

This order achieves unit-stride access on all arrays by using memory in cache-line order, in Fortran terms, the first index (row) is looped innermost, which maximizes locality. Indeed if repeat the previous experiment by adding the measure for the new code, we get:

```
Average time for matmul_ijl:  0.37526105999999992
Average time for matmul_jil:  0.15405741999999961
Average time for matmul_jli:   9.1884539999999792E-002
```

The new code is about 1.5 times faster than the previous one, and about 4 times faster than the first one; see Figure 6.6 for a graphical representation of the performance of the three versions of the code and the time measured for the OpenBLAS implementation of the DGEMM.

Exercise 6.2.5 Reimplement this experiment on your own machine, and measure the performance of the three versions of the code and version of the BLAS you have installed. You can use the `-O3` optimization flag, and the `-mtune=native` and `-march=<---->` flags, which are used to optimize the code for the specific architecture of the machine you are using. Try also other *computing loads* by varying the size of the matrices, and see how the performance of the code changes.

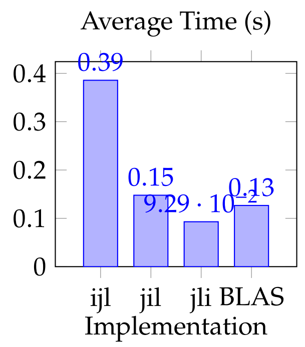

Figure 6.6: Performance of the three versions of the GEMM operation for the three different ordering of the `do` loops together with the DGEMM implementation of the OpenBLAS.

Please note that with these examples we have just begun to explore the coding techniques that are needed to achieve peak performance: a very substantial amount of work would still be necessary to arrive at a level of performance comparable to that of most BLAS implementations.

6.2.4 Performance consideration for the BLAS

The BLAS were designed to solve a major problem: different computer architectures typically require different coding techniques to achieve optimal performance. This entails the need to recode algorithms for every new major computing platform, which is clearly unsustainable in the long term given the complexity of coding algorithms in an efficient and stable manner.

Modern compilers are very sophisticated, and some of the techniques that once were necessarily implemented "by hand" can now be delegated to the compiler, possibly under the control of directives as in OpenMP

The solution to this major problem revolves arount the idea of the BLAS:

1. Design and code algorithms based on the BLAS operators;
2. Reimplement "only" the BLAS operators for new computer architectures.

By and large, the above strategy has worked effectively, indeed the success of LAPACK is based precisely on the peformance portability of the BLAS.

Note that most modern computer architectures are based on processors connected to a memory hierarchy, designed to alleviate the speed differential beween the processor and the main memory access; these hierarchies are based on the idea of reusing data as much as possible whenever it has been accessed. This concept of data reuse naturally matches the design of the level 3 BLAS, where we access $O(n^2)$ data to perform $O(n^3)$ operations, thus providing the potential to reuse each data item n times.

It is also possible to build the entire BLAS library around an efficient implementation of GEMM plus a few additional support routines, see [29].

Sparse Matrices and Iterative Solvers 7

7.1 Introduction

What is a sparse matrix? The most famous definition of "sparse matrix" is attributed to James Wilkinson, one of the founding fathers of numerical linear algebra [30]:

> Any matrix with enough zeros that it pays to take advantage of them.

Sparse linear systems (and sparse eigenvalue problems) often arise in the solution of Partial Differential Equations, and in the sequel we will almost always assume that this is the case; other sources exist, but they often exhibit somewhat different features, and we will not discuss them in detail.

Solving PDEs in a closed, analytic form is almost always impossible, and even when it is possible, it is by no means guaranteed that this will lead to the most accurate solution. Therefore, when faced with PDEs, we typically have to discretize and linearize them; there are multiple techniques to perform these steps, but for the most part they share the same general appearance:

- The original PDE domain is partitioned, and only $O(1)$ variables and equations are associated with each partition;
- Each equation only contains a limited number of nonzero coefficients k; moreover, k is bounded independently of the size n of the domain (and of the linear system), and is determined by local topological features.

These facts tend to push towards a different way of organizing the data layout, as well as handling the interaction with the discretization mesh these problems come from.

Another extremely important point about sparse problems is that they almost always require *iterative* solution strategies. To see why this is the case, we need to consider two major factors.

The first one is that when an $n \times n$ matrix A is sparse, its storage requires much less than n^2 memory; indeed, if the number of nonzeros per row is $O(1)$, that is, it is bounded independently of n, it is possible to store A using only $O(n)$ memory by storing explicitly only the non-zero coefficients together with their indices, and this is very often the case in a PDE context.

The fact that matrix A only takes up $O(n)$ memory is extremely important in that n is typically related to the degree of refinement of the mathematical model; thus, a linear storage cost will encourage the use of much more refined models (given the same amount of memory available), entailing much larger values for n.

The second factor emerges from the factorization strategies we describe in Section 6.2 and Section 8.1: the update steps in Algorithm 7 tend to introduce non-zeros into positions that in the original matrix where zeros (and thus not stored explicitly). This phenomenon is called *fill-in* and ultimately limits the effectiveness of factorization strategies when applied to sparse problems: since a cheap storage footprint $O(n)$ encourages to use much larger values of n, fill-in leads to excessive memory pressure by growing the number of nonzeros.

The ideal situation would be to require that the storage associated with A remains the same throughout the solution process: this is exactly what happens with *iterative solvers*, where the matrix is only employed to perform operations that do not alter its structure such as matrix-vector multiplications.

7.1.1 A simple iterative solver

Let us consider the following iteration (Richardson):

$$x_{k+1} = x_k + \omega(b - Ax_k). \tag{7.1}$$

If the scalar ω is such that $\|I - \omega A\| < 1$, then the iteration converges and the *residual* $(b - Ax_k)$ becomes negligible, or

$$\bar{x} = \lim_{k\to\infty} x_k, \qquad A\bar{x} = b,$$

with the obvious caveat that in practice we stop the iteration as soon as x_k is "good enough".

The notion of "good enough" is usually implemented by computing the norm of the residual, typically scaled against the norm of the RHS, and comparing it with a tolerance:

$$\frac{\|b - Ax_k\|}{\|b\|} \leq \tau. \tag{7.2}$$

Thus the basic ingredients of any simple iterative solver can be listed as follows:

1. Matrix-vector products;
2. Vector sums;
3. Vector norms;
4. Vector dot products.

There are many possible ways to combine these ingredients in defining an iterative solver; the previous method 7.1 is just a very simple example, and much more sophisticated methods have been developed over the years.

7.1.2 Classical iterative solvers

Let us now state a slightly modified version of of 7.1

$$x_{k+1} = x_k + M^{-1}(b - Ax_k), \tag{7.3}$$

where M is a linear operator (preconditioner) that formally transforms $Ax = b$ into $MAx = Mb$, and ideally satisfies the properties:

1. M is easy to compute;
2. M^{-1} is easy to apply;
3. $M^{-1}A \approx I$.

Different choices for M give rise to many iterative methods, including common ones like Jacobi, Gauss-Seidel, Chebychev. From the point of view of the software kernels that we need to develop for a complete implementation we have practically covered all of them (save for the sparse triangular system solution, which however can be easily derived from the matrix-vector code).

7.1.3 Krylov solvers and preconditioners

Unfortunately the methods that can be derived from 7.3 are not efficient enough for large scale problems. The algorithms of choice to solve very large scale problems are currently drawn from the class of Krylov subspace iterations, preconditioned with many different strategies. We will very briefly recall here two of the main Krylov methods, Conjugate Gradients (CG) and Generalized Minimum Residual (GMRES), and give some ideas about preconditioning; for a thorough introduction to the subject see [31].

Conjugate Gradients

The first method we describe is the famous Conjugate Gradients method, which is applicable when the matrix A is symmetric positive definite (SPD). We start from its original derivation by defining an equivalent minimzation problem:

$$\min_x \phi(x) = \frac{1}{2}x^T Ax - x^T b. \tag{7.4}$$

That this is equivalent to our linear system can be easily seen by applying the conditions for the minimizer x_c:

$$\min_x \phi(x) \quad \Rightarrow \quad \nabla\phi(x_c) = 0,$$

and computing the gradient of the function we come back to

$$Ax_c - b = 0,$$

which is our original problem. An obvious idea is then to start from an arbitrary point and move along the gradient, which is, after all, the direction along which the function changes most rapidly:

$$-\nabla\phi(x) = b - Ax = r,$$

since we want to have a reduction in ϕ. If the residual r is nonzero, there exists α such that

$$\phi(x_i + \alpha r_i) < \phi(x_i)$$

and it can be easily seen that

$$\alpha = r_i^T r_i / r_i^T A r_i.$$

This is the steepest descent method, which is however not satisfactory beacuse the gradient at each step is constrained to be orthogonal to the gradient at the previous step

$$r_{k+1}^T r_k = 0,$$

and this may slow down convergence significantly. If we now modify the search direction as in

$$p_k = r_{k-1} + \beta_k p_{k-1},$$

and apply a little bit of algebra, we arrive at the conjugate gradients method, which has the property that

$$p_k^T A p_j = 0 \qquad k \neq j,$$

or, the search directions are orthogonal *in the inner product induced by the matrix A*. Note that by properly sequencing the operations and providing

Algorithm 2: Conjugate Gradients

```
1  Compute $r^{(0)} \leftarrow b - Ax^{(0)}$ ;
2  while $r_i \neq 0$ do
3      $i \leftarrow i + 1$;
4      if $i = 1$ then
5          $p^{(1)} \leftarrow r^{(0)}$ ;
6      else
7          $\beta_i \leftarrow -p_{i-1}^T A r_{i-1} / p_{i-1}^T A p_{i-1}$;
8          $p^{(i)} \leftarrow r^{(i-1)} + \beta_i p^{(i-1)}$;
9      $\alpha_i \leftarrow p_i^T r_{i-1} / p_i^T A p_i$;
10     $x^{(i)} \leftarrow x^{(i-1)} + \alpha_i p^{(i)}$;
11     $r^{(i)} \leftarrow r^{(i-1)} - \alpha_i A p_i$;
```

some additional storage, the method only requires one matrix-vector product per iteration.

GMRES

What should we do when the matrix A is not SPD? One possible method is GMRES, which computes at each step the solution to the minimization problem

$$\min \|b - Ax\|_2,$$

from the vector space V_k built by successive multiplications by A

$$V_k = \operatorname{span}\{r, Ar, A^2 r, \ldots, A^k r\}.$$

GMRES requires keeping track explicitly of V_k and therefore is quite expensive in terms of memory; it is usually implemented in its restarted (RGMRES) varian, that is, builds V_k up to a certain size m then restarts the process.

Algorithm 3: GMRES(m)

```
1  Compute $r_0 \leftarrow b - Ax_0$, $\beta \leftarrow \|r_0\|$ and $v_1 \leftarrow r_0/\beta$;
2  for $j = 1, \dots, m$ do
3      Compute $w_j \leftarrow Av_j$;
4      for $i = 1, \dots, j$ do
5          $h_{ij} \leftarrow (w_j, v_i)$;
6          $w_j \leftarrow w_j - h_{ij}v_i$;
7      $h_{j+1,j} \leftarrow \|w_j\|_2$. If $h_{j+1,j} = 0$ then $m \leftarrow j$ and go to 9;
8      $v_{j+1} \leftarrow w_j/h_{j+1,j}$;
9  Define the $(m+1) \times m$ Hessenberg matrix $H_m = \{h_{ij}\}_{1 \leq y \leq m+1, 1 \leq j \leq m}$ ;
10 Compute $y_m$ the minimizer of $\|\beta e_1 - H_m y\|_2$ and set
   $x_m \leftarrow x_0 + V_m y_m$;
```

Other Krylov methods

There are many other variants of Krylov subspace methods, among which we can list BCG, CGS, BiCGSTAB, BiCGSTAB(l), QMR, TFQMR and so on.

No method is universally better than all the others; the proper choice will require some experimentation and/or an analysis of the spectral properties of the linear system matrix A.

7.1.4 Preconditioners

In the method 7.3 we mentioned the use of a linear transformation M called a preconditioner. An appropriate choice of preconditioner is often essential to achieve convergence in a reasonable amount of effort. A *preconditioner* is a preprocessing for the linear algebraic system

$$Ax = b, \qquad x, b \in \mathbb{R}^n,\ A \in \mathbb{R}^{n \times n}, \tag{7.5}$$

that transforms it into an equivalent one by applying a nonsingular transformation, for example:

$$M^{-1}Ax = M^{-1}b. \tag{7.6}$$

The choice of the transformation M is usually made so that certain spectral properties of the coefficient matrix $M^{-1}A$ are more favourable for the convergence properties of the Krylov methods applied to the system at hand. Equation (7.6) shows a *left* preconditioning approach; it is also possible to apply a *right* preconditioning,

$$AM^{-1}u = b, \quad x = M^{-1}u \tag{7.7}$$

or a split preconditioning

$$M_1^{-1}AM_2^{-1}u = M_1^{-1}b, \quad x = M_2^{-1}u. \tag{7.8}$$

In the case of A sparse, the transformed matrix $M^{-1}A$ is usually not built explicitly, because

1. The inverse of a sparse matrix is in general dense

2. The product of two sparse matrices is denser than either of the factors.

So, what normally happens is that the operator M^{-1} is applied after matrix A at each step in the method requiring an explicit inversion and a product.

The preconditioner matrix M should then satisfy three conflicting requirements:

- ▶ It should be computed cheaply from A (cfr the setup time in equation (7.9));
- ▶ It should be cheap to apply M^{-1};
- ▶ We should have $\|A - M\|$, $\|M^{-1}\|$ and $\|I - M^{-1}A\|$ "small" for some norm depending on the problem at hand.

The number of iterations needed to attain a prescribed tolerance is a very important factor but we also need to consider the cost of each iteration as well as the preprocessing cost. If the choice of M finds a good tradeoff between the quality of the approximation of A and the cost to compute and apply M, then the benefit in the number of iterations j can compensate the setup time and computational overhead per iteration needed to implement the preconditioning strategy. For example we often have that the time to solution T_{slv} is given by:

$$T_{slv} = T_{setup} + N_{it} \times T_{it}, \tag{7.9}$$

where T_{setup} is the time to build the preconditioner, N_{it} the number of iterations and T_{it} the time per iteration.

On the other hand, if either the setup or the application of the M^{-1} operator is too expensive, then the total time may grow.

Taking two extreme examples, if we choose $M = I$, then both setup and application are very fast, but the convergence is no better than before. If, on the other hand, if we choose $M = A$, then the setup is very easy and the convergence is very fast, but at each step we would need to solve the very same problem we started from, and therefore we would end up with an application phase that costs just as much as the full solution.

The art of preconditioning lies therefore in finding a reasonable trade-off among all of the previous factors.

Stationary Iterations as Preconditioners

While stationary iteration methods such as the Richardson iteration 7.1 are not very effective in comparison with Krylov subspace methods, the splittings on which they are based can be used to provide a preconditioner M. The simplest preconditioner is the (point) Jacobi preconditioner in which we invert the diagonal D of matrix A. When using these kind of splittings we usually specify a fixed (and small) number of iterations to be performed.

Note that to implement the Jacobi iteration we need to implement multiplication by a vector element by element, whereas to implement the Gauss-Seidel iteration we need to implement a triangular system solution for a sparse coefficient matrix.

Incomplete Factorizations

For a general sparse matrix A the application of a direct solution method often means computing its LU factorization: if we can afford the computation of the triangular factors, we can also solve (exactly) a linear system having A as a coefficient matrix, and we also have a "preconditioner" that guarantees immediate convergence of iterative methods. The catch is obvious: the triangular factors are too expensive because of *fill-in*, i.e. they contain many more nonzeroes than the original matrix A, and we can rarely afford their exact computation.

However we can trade the exactness of the factorization for the memory space, by defining an *incomplete* factorization as

$$A = \hat{L}\hat{U} - R$$

where we throw the fill-in onto the residual R (which is discarded), to keep an acceptable sparsity structure in the triangular matrices $\hat{L}\hat{U}$. Thus a general incomplete factorization may be stated as Algorithm 4 by referring to a *pattern* P which discriminates the accepted factorization entries. Algorithm 4 is written in the customary form overwriting the

Algorithm 4: General Incomplete LU Factorization

```
for $i = 2, \dots, n$ do
    for $k = 1, \dots, i-1$ do
        if $(i, k) \in P$ then
            $a_{ik} \leftarrow a_{ik}/a_{kk}$ ;
            for $j = k+1, \dots, n$ do
                if $(i, j) \in P$ then
                    $a_{ij} \leftarrow a_{ij} - a_{ik}a_{kj}$ ;
```

entries of A with the entries of $\hat{L}$ and $\hat{U}$; in practice, since the iterative methods will require A for the matrix-vector product, the factors are stored separately.

That a matrix A admits an incomplete factorization and that the factorization is a good preconditioner are entirely non-trivial propositions; however there exists a vast class, the so-called M-matrices, for which it can be proven that such a factorization exists [31, 32], and gives rise to a *convergent* splitting, thus guaranteeing a good quality preconditioner.

Algebraic Multigrid Preconditioners

Multilevel methods are often used to build preconditioners for the matrix A, and are coupled with Krylov methods to solve a linear system $Ax = b$. Generally speaking, a multilevel method provides an approximate inverse of A by suitably combining approximate inverses of a hierarchy of matrices that represent A in increasingly coarser spaces. This is achieved by recursively applying two main processes: *smoothing*, which provides an approximate inverse of a single matrix in the hierarchy, usually by a simple iteration, and *coarse-space correction*, which computes a correction to the approximate inverse by transferring suitable information from the current space to the next coarser one and vice versa, and by computing, through smoothing, an approximate inverse of the coarse matrix (see [33–35]).

Our software framework for multigrid preconditioner is detailed in [36].

Final observations

The literature on preconditioners is immense, and it is impossible to do it justice in these brief notes; some good starting points include [31, 33–35], whilst our own contributions can be found in e.g. [36–38]. We will not discuss preconditioners in any further details here, except to underline that, as we have seen, from a software point of view they require the availability of two additional kernels:

1. Multiplication/division of two vectors element by element;
2. Solution of a triangular linear system with a sparse coefficient matrix.

7.2 Sparse Matrix-Vector product

Let us return to the sparse matrix definition by Wilkinson:

> Any matrix with enough zeros that it pays to take advantage of them.

This definition implicitly refers to some operation in the context of which we are "taking advantage" of the zeros; experience shows that it is impossible to exploit the matrix structure in a way that is uniformly good across multiple operators, let alone multiple computing architectures. The usual two-dimensional array storage is a linear mapping that stores the coefficient $A(I, J)$ of an $M \times N$ matrix at position $(J-1) \times M + I$ of a linear array. This formula assumes column-major ordering used in Fortran, Matlab and Julia; an analogous formula applies for row-major storage used in C and Java. Thus representing a matrix in memory requires just one linear array, and two integer values detailing the size of the matrix.

Now enter sparse matrices: "taking advantage" of the zeros essentially means avoiding their explicit storage. But this means that the simple mapping between the index pair (I, J) and the position of the coefficient in memory is destroyed. Therefore, all sparse matrix storage formats are devised around means of rebuilding this map using auxiliary index information: a pair of dimensions does not suffice any longer. How costly this rebuilding is in the context of the operations we want to perform is the critical issue we need to investigate. Indeed, performance of sparse matrix kernels is typically much less than that of their dense counterparts, precisely because of the need to retrieve index information and the associated memory traffic. Moreover, whereas normal storage formats allow for sequential and/or blocked accesses to memory in the input and output vectors x and y, sparse storage means that coefficients stored in adjacent positions in the sparse matrix may operate on vector entries that are quite far apart, depending on the *pattern* of nonzeros contained in the matrix.

By now it should be clear that the performance of sparse matrix computations depends critically on the specific representation chosen. Multiple factors contribute to determine the overall performance:

- the match between the data structure and the underlying computing architecture, including the possibility of exploiting special hardware instructions;
- the suitability of the data structure to decomposition into independent, load-balanced work units;
- the amount of overhead due to the explicit storage of indices;
- the amount of padding with explicit zeros that may be necessary;
- the interaction between the data structure and the distribution of nonzeros (pattern) within the sparse matrix;
- the relation between the sparsity pattern and the sequence of memory accesses especially into the x vector.

Many storage formats have been invented over the years; a number of attempts have also been directed at standardizing the interface to these data formats for convenient usage (see e.g., [39]).

We will now review two very simple and widely-used data formats: COOrdinate (COO) and Compressed Sparse Rows (CSR). These two formats are probably the closest we can get to a "general purpose" sparse matrix representation. For each format, we will show the representation of the example matrix in Figure 7.1.

Table 7.1: Notation for parameters describing a sparse matrix

Name	Description
M	Number of rows in matrix
N	Number of columns in matrix
NZ	Number of nonzeros in matrix
AVGNZR	Average number of nonzeros per row
MAXNZR	Maximum number of nonzeros per row
NDIAG	Number of nonzero diagonals
AS	Coefficients array
IA	Row indices array
JA	Column indices array
IRP	Row start pointers array
JCP	Column start pointers array
NZR	Number of nonzeros per row array
OFFSET	Offset for diagonals

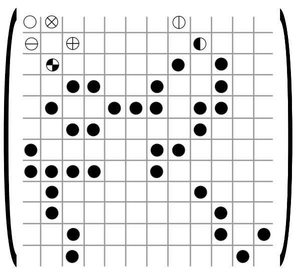
Figure 7.1: Example of sparse matrix

7.2.1 COOrdinate

The COO format is a particularly simple storage scheme, defined by the three scalars `M`, `N`, `NZ` and the three arrays `IA`, `JA` and `AS`. By definition of number of rows we have $1 \leq IA(i) \leq M$, and likewise for the columns; a graphical description is given in Figure 7.2.

Algorithm 5: Matrix-Vector product in COO format

```
do i=1,nz
  ir = ia(i)
  jc = ja(i)
  y(ir) = y(ir) + as(i)*x(jc)
end do
```

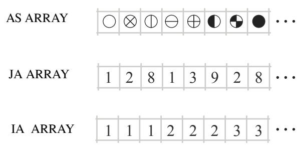

Figure 7.2: COO compression of matrix in Figure 7.1

The code to compute the matrix-vector product $y = Ax$ is shown in Alg. 5; it costs five memory reads, one memory write and two floating-point operations per iteration, that is, per nonzero coefficient. Note that the code will produce the result y (to within floating-point rounding tolerance) even if the coefficients and their indices appear in an arbitrary order inside the COO data structure.

7.2.2 Compressed Sparse Rows

The CSR format is perhaps the most popular sparse matrix representation. It explicitly stores column indices and nonzero values in two arrays `JA` and `AS` and uses a third array of row pointers `IRP`, to mark the boundaries of each row. The name is based on the fact that the row index information is compressed with respect to the COO format, after having sorted the coefficients in row-major order. Figure 7.3 illustrates the CSR representation of the example matrix shown in Figure 7.1.

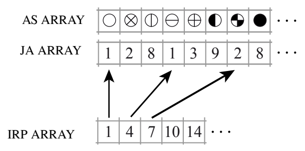

Figure 7.3: CSR compression of matrix in Figure 7.1

Algorithm 6: Matrix-Vector product in CSR format

```
do i=1,m
  t=0
  do j=irp(i),irp(i+1)-1
     t = t + as(j)*x(ja(j))
  end do
  y(i) = t
end do
```

The code to compute the matrix-vector product $y = Ax$ is shown in Alg. 6; it requires three memory reads and two floating-point operations per iteration of the inner loop, i.e., per nonzero coefficient; the cost of the access to `x(ja(j))` is highly dependent on the matrix pattern and on its interaction with the memory hierarchy and cache structure. In addition, each iteration of the outer loop, i.e., each row of the matrix, requires reading the pointer values `irp(i)` and `irp(i+1)`, with one of them available from the previous iteration, and one memory write for the result.

7.2.3 Sparse Matrix-Vector Product considerations

The previous discussion of COO and CSR has barely scratched the surface of the possible implementations of the kernels for sparse matrices; as an example, the article [40] lists 67 different variations, just for usage on GPUs, and all of them published in just the four preceding years.

From a user point of view, the variability is therefore bewildering; it is also quite problematic to hardwire a data storage format into your software, since this will inevitably make the end product inflexible, hard to evolve and to adapt to new computing architectures and usage conditions.

Moreover, it is a fact that even on a single, given computing architecture, no individual storage format is likely to be uniformly better than all others when considering different operations involving the matrix itself. Encapsulating storage format variations under a uniform outer shell allows for having a single entry point with no need for conditional compilation in the case in which the user is actually running in serial mode.

The usefulness of having an object with the ability to switch among different types was recognized long ago; as early as 1983 we find the following statement in [41], Section 4.12:

> Often a seemingly simple representation problem for a set or mapping presents a difficult problem of data structure choice. Picking one data structure for the set makes certain operations easy, but others take too much time and it seems that there is no one data structure that makes all the operations easy. In that case the solution often turns out to be the use of two or more different structures for the same set or mapping.

It is therefore desirable to have a flexible framework that allows to switch among different formats as needed, even at runtime.

7.2.4 Design Patterns: the "State" Pattern

In the Object Oriented Design of software, the term "Design Pattern" denotes an accepted best practice in the form of a common solution to a software design problem that recurs in multiple contexts [42, 43].

The State pattern in Object Oriented Design is a behavioral pattern that involves encapsulating an object's state behind an interface in order to facilitate varying the object's type at runtime. Figure 7.4 shows a UML class diagram of the State pattern, including the class relationships and the public *methods*. The methods described in an OOD typically map to the type-bound procedures in Fortran OOP.

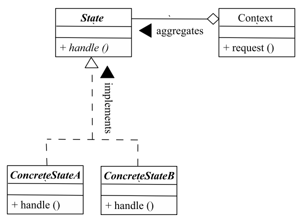

Figure 7.4: State design pattern

Let us consider the problem of switching among different storage formats for a given object. Before the dawn of OOP, a common solution involved defining a data structure containing integer values that drive the interpretation and dispatching of the various operations on the data. This older route complicates software maintenance by requiring the rewriting and recompiling of previously working code every time one incorporates a new storage format. A more modern, object-oriented strategy builds the dispatching information into a type system, thereby enabling the compiler to perform the switching. However, most OOP languages do not allow for a given object to change its type dynamically (there do exist dynamically typed languages, but they are not in common use). This poses the dilemma of how to reference the object and yet allow for the type being referenced to vary.

The solution lies in adding a layer of indirection by encapsulating the object inside another object serving only as an interface that provides a handle to the object in a given context. All code in the desired context references the handle but never directly references the actual object. This solution enables the compiler to delay until runtime the determination of the actual object type (what Fortran calls the "dynamic type"). The sample code in Figure 7.5 demonstrates the State pattern in a sparse-matrix context, wherein a `base_sparse_mat` type plays the role of "State" from Figure 7.4 and `spmat_type` serves as the "Context" also depicted in

```
module base_mod
  ! The base class for STATE objects
  type :: base_sparse_mat
    ! data components here
  contains
    procedure, pass(a) :: foo => base_foo
  end type base_sparse_mat
contains
  subroutine base_foo(a)
    class(base_sparse_mat) :: a
    ! Actual implementation
    write(*,*) 'This the FOOing of a base sparse matrix'
  end subroutine base_foo
end module base_mod

module coo_mod
  ! A derived class for STATE objects in COO
  use base_mod
  type, extends(base_sparse_mat) :: coo_sparse_mat
    integer :: nnz=0               !> Number of nonzeros.
    integer, allocatable :: ia(:)  !> Row indices.
    integer, allocatable :: ja(:)  !> Column indices.
    real, allocatable :: val(:)    !> Coefficient values.

  contains
    procedure, pass(a) :: foo => coo_foo
  end type coo_sparse_mat
contains
  subroutine coo_foo(a)
    class(coo_sparse_mat) :: a
    ! Actual implementation
    write(*,*) 'This the FOOing of a coo  sparse matrix with',&
         & a%nnz,' nonzero entries'
  end subroutine coo_foo
end module coo_mod
```

Figure 7.5: Code for the State pattern — inner object

Figure 7.4. The methods of the outer class delegate all operations to the inner-class methods. The inner class serves as the actual workhorse.

An interesting side effect for the State pattern is that is allows easy handling of heterogeneous computing platforms: the application program making use of the computational kernels will see a uniform outer data type, but the inner data type can be easily adjusted according to the specific features of the processing element that the current process is running on.

In our software, we implement such a flexible architecture based on the techniques outlined in [44]. In particular, it is possible to add support for accelerator devices resulting in the data structures described in [40, 45].

```
module spmat_mod
  ! The class for CONTEXT objects
  use base_mod
  type :: spmat_type
    class(base_sparse_mat), allocatable :: a
  contains
    procedure, pass(a) :: foo => spmat_foo
  end type spmat_type
contains
  subroutine spmat_foo(a)
    class(spmat_type) :: a
    call a%a%foo()
  end subroutine spmat_foo
end module spmat_mod

! Simple example
program try
  use spmat_mod
  use coo_mod
  type(spmat_type) :: a
  ! Start with the base STATE
  allocate(a%a)
  call foobar(a)
  ! Switch to COO
  deallocate(a%a)
  allocate(coo_sparse_mat :: a%a)
  call foobar(a)

contains
  ! Workhorse
  subroutine foobar(a)
    type(spmat_type) :: a
    call a%foo()
  end subroutine foobar
end program try
```

Figure 7.6: Code for the State pattern — outer context

Parallel Linear Algebra Software Design and Additional Features

Where is my data? 8

As everybody knows, there are three important factors for parallel performance: data location, data location and data location*. This is due to a fact we already mentioned briefly in Chapter 1: it is an unfortunate feature of the technological evolution that memory speed does not keep up with the speed of processors. Hence, the placement of data and its management play a significant role in determining the performance of software; moreover, the placement of data in the case of dense vs sparse linear algebra is driven by somewhat different considerations, and results in quite different layouts.

The basic principle when choosing a data layout is always the same: find the best possible tradeoff between the performance local to a given node and the communication behaviour of the algorithms, whilst at the same time keeping the best possible load balancing.

This objective requires quite a bit of analysis and work to be achieved; moreover, even if we apply the same basic principle, the actual outcome is very problem and algorithm dependent, as we shall soon see.

8.1 Dense linear algebra data distribution

To distribute data for dense linear algebra we have to keep track of the following issues:

1. The need to account for the data reuse techniques of the level 3 BLAS (blocking) and for the surface-to-volume effect, which expresses the trade-off between communication ("surface") and computation ("volume") when data are distributed across multiple processes. Although no explicit spatial geometry is involved here, the underlying principle of the surface-to-volume effect still influences data distribution strategies and is implicitly taken into account;
2. The load balancing features of the various algorithms (we'll use the LU factorization as a reference);
3. The need to choose the best possible process grid configuration (for MPI programs).

8.1.1 Simple LU factorization

Let's begin our discussion taking as a reference the *LU* factorization algorithm in its most elementary form with a concrete example of size 3 (we are ignoring pivoting for the time being):

* The phrase "location, location, location" used in reference to real estate is commonly attributed to the british real estate tycoon Harold Samuel (1912-1987), but is probably much older.

- Factor the diagonal (auxiliary constraint: $l_{ii} = 1$)

$$\text{Compute} \begin{pmatrix} a_{11} \end{pmatrix} \rightarrow \begin{pmatrix} l_{11} \end{pmatrix} (u_{11})$$

- Update the first column:

$$\begin{pmatrix} l_{21} \\ l_{31} \end{pmatrix} \leftarrow \begin{pmatrix} a_{21} \\ a_{31} \end{pmatrix} (u_{11})^{-1}$$

- Update the first row:

$$\begin{pmatrix} u_{12} & u_{13} \end{pmatrix} \leftarrow (l_{11})^{-1} \begin{pmatrix} a_{12} & a_{13} \end{pmatrix}$$

- Update the lower-right submatrix;

$$\begin{pmatrix} \hat{a}_{22} & \hat{a}_{23} \\ \hat{a}_{32} & \hat{a}_{33} \end{pmatrix} \leftarrow \begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix} - \begin{pmatrix} l_{21} \\ l_{31} \end{pmatrix} \begin{pmatrix} u_{12} & u_{13} \end{pmatrix}$$

- Apply recursively to lower-right submatrix;

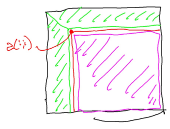

Figure 8.1: A simple LU factorization algorithm

As we can see, the algorithm proceeds column by column, and the amount of work to be performed at each step diminishes steadily over the course of the execution. Before proceeding, we can recast the previous considerations in a more formal setting where $a_{i,j}$ is the generic entry of the linear system matrix A, $a_{i,:}$ is the i-th row and similarly for the columns, as shown in algorithm 7; similar notation is used for the factors L and U. Note that the update of the row $(l_{11})^{-1} \begin{pmatrix} a_{12} & a_{13} \end{pmatrix}$ is clearly a no-op in this formulation since $l_{11} = 1$.

Algorithm 7: Simple *LU* factorization

```
for $i \leftarrow 1$ to $n$ do
    $l_{i,i} \leftarrow 1$;
    $u_{i,i} \leftarrow a_{i,i}$;
    $l_{i+1:n,i} \leftarrow a_{i+1:n,i} \cdot u_{i,i}^{-1}$;
    $a_{i+1:n,i+1:n} \leftarrow a_{i+1:n,i+1:n} - l_{i+1:n,i} \cdot u_{i,i+1:n}$;
```

Let us now consider how we could parallelize the above program in a distributed memory environment.

8.1.2 A 1-dimensional layout for LU

We start with the simplest posssible arrangement of n_p processes (MPI tasks): a 1-dimensional array. To begin with, we allocate to each process a chunk of $n_b = n/n_p$ consecutive columns (assuming they divide exactly), and let the algorithm 7 execute. In other words, each process $p, p = 0 \ldots n_p - 1$ owns a vertical stripe of the input matrix $a_{1:n,(p \cdot n_b)+1:(p+1) \cdot n_b - 1}$. This is also known as a BLOCK data layout Figure Figure 8.2.

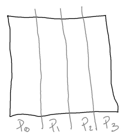

Figure 8.2: A BLOCK 1-D data distribution

Given that in this configuration any column $a_{:,i}$ is owned by a single process, that process can perform the first two steps of the algorithm; then, it has to communicate to all others the value of $l_{i+1:n,i}$ so that each process can proceed to update its own stripe of the result as shown in algorithm 8. This algorithm gives a first view of what a parallel algorithm typically looks like: it alternates between computations and

Algorithm 8: 1-D parallel *LU* factorization

```
Every process has index $i_p$;
Every process owns $n_b \leftarrow n/n_p$ columns;
for $i \leftarrow 1$ to $n$ do
    $i_p = (i-1)/(n_b)$;
    if I am $i_p$ (I own column i) then
        $l_{i,i} \leftarrow 1$;
        $u_{i,i} \leftarrow a_{i,i}$;
        $l_{i+1:n,i} \leftarrow a_{i+1:n,i} \cdot u_{i,i}^{-1}$;
        Broadcast send $l_{i+1:n,i}$;
    else
        Broadcast receive $l_{i+1:n,i}$;
    $j_{st} \leftarrow \max(i_p \cdot n_b + 1, i + 1)$ (first column I own);
    $j_{en} \leftarrow \min((i_p + 1) \cdot n_b - 1, n)$ (last column I own);
    $a_{i+1:n,j_{st}:j_{en}} \leftarrow a_{i+1:n,j_{st}:j_{en}} - l_{i+1:n,i} \cdot u_{i,j_{st}:j_{en}}$;
```

communication phases. Naturally, to optimize the runtime of an algorithm it is necessary to achieve that:

1. The computational phases are as efficient and as balanced as possible;
2. The communication phases are kept to a minimum, and if at all possible shuld be overlapped with the computations.

In this particular formulation of the *LU* algorithm we can immediately notice that the only part of the computation that is really happening in parallel is the rank-1 update of the lower-right submatrix $a_{i+1:n,j_{st}:j_{en}} \leftarrow a_{i+1:n,j_{st}:j_{en}} - l_{i+1:n,i} \cdot u_{i,j_{st}:j_{en}}$; fortunately, this is also the most expensive part.

There is one major point that needs to be highlighted though: as the loop index i progresses, it will eventually be the case that $i > n_b$. At this point, process 0 will stop doing any work, and the rest of the calculation will be carried out by processes 1 through $n_p - 1$, that is, the degree of parallelism will reduce. A similar phenomenon will happen every time the index i crosses the boundary between one process and the next; therefore, the efficiency of the parallel algorithm is greatly diminished.

A CYCLIC layout for LU

Is it possibile to improve? Let's rethink the data layout. When we describe the algorithm for the *LU* factorization as in 7 we are naturally inclined to think of columns i and $i+1$ as being next to each other in the matrix layout, which is what happens in the memory of a single node for a serial implementation. And yet, this is by no means necessary: all that is required is that the order of processing follows the index i in the *logical* view of the underlying matrix A.

In other words, if column $i+1$ is stored in a memory space that is not adjacent to that for column i, we can still apply the same algorithm provided that we can figure out where exactly columns i and $i+1$ are located. This is a first example of a general concept: an index space $\mathcal{I} = \{i, i = 1 \ldots n\}$ can be *distributed* over a set of processes, but this does not prevent our algorithms from working as long as we can figure out the *mapping* between the abstract index i and its physical counterpart, say local index j on process p.

As an example of this, we can store column 1 on the first process, column 2 on the second, and so on, with column i being assigned to process $\mod(i-1, n_p)$ where the -1 is needed to adjust for the range of processes being $\{0, \ldots n_p - 1\}$. Thus, each process will have a bunch of columns that can be stored adjacently, because the step $a_{i+1:n,i+1:n} \longleftarrow a_{i+1:n,i+1:n} - l_{i+1:n,i} \cdot u_{i,i+1:n}$ does not really depend on the ordering of columns, as long as the physical position of i and $i+1$ can be computed. The distribution we have discussed above, corresponding

Algorithm 9: CYCLIC 1-D parallel *LU* factorization

```
1  Every process has index $i_p$;
2  Columns are assigned to processes in a round-robin fashion;
3  Thus, column $i$ is owned by process $\mod((i-1), n_p)$;
4  for $i \longleftarrow 1$ to $n$ do
5      $i_p = \mod((i-1), n_p)$;
6      $j_{st}$ first column I own that is $\geq i$;
7      $j_{en}$ last column I own ;
8      if I am $i_p$ (I own column i) then
9          $l_{i,i} \longleftarrow 1$;
10         $u_{i,i} \longleftarrow a_{i,i}$;
11         $l_{i+1:n,i} \longleftarrow a_{i+1:n,i} \cdot u_{i,i}^{-1}$;
12         Broadcast send $l_{i+1:n,i}$;
13         $\delta_j = 1$;
14     else
15         Broadcast receive $l_{i+1:n,i}$;
16         $\delta_j = 0$;
17     $a_{i+1:n,j_{st}+\delta_j:j_{en}} \longleftarrow a_{i+1:n,j_{st}+\delta_j:j_{en}} - l_{i+1:n,i} \cdot u_{i,j_{st}+\delta_j:j_{en}}$;
```

Can you figure out the values of j_{st} and j_{en}?

to a round-robin assignment, is called a CYCLIC distribution of data.

In the context of the present discussion, the main advantage with respect to the BLOCK distribution is that process 0 only stops doing useful computations in the last n_p iterations of the main factorization loop; therefore all processes remain active up until the very end of the algorithm, and there is an imbalance only in the last few iterations. Since n is usually much larger than n_p, the resulting imbalance is negligibile.

A BLOCK-CYCLIC 1D layout for LU

The previous arrangement is very good for one of the main performance factors, that is, load balancing; however, it relies on the level-2 update kernel implementing the operation

$$a_{i+1:n,i+1:n} \longleftarrow a_{i+1:n,i+1:n} - l_{i+1:n,i} \cdot u_{i,i+1:n},$$

which is to be understood as being applied to the reordered columns of matrix A in a split fashion, with each process updating its own part.

This strategy is not satisfactory because it does not apply the level-3 BLAS kernels, which we know are necessary to achieve best performance on available processing cores.

A cure for this problem is readily available: apply a CYCLIC distribution not to *individual* columns, but to *sections* of n_b adjacent columns.

If the value for n_b is chosen properly, we can achieve a good tradeoff between the need to have optimal "local" performance, that is use level 3 computations, and optimal "parallel" performance, that is achieving load balancing and good surface-to-volume ratio.

The price to be paid is that the update of $u_{i,i+1:n}$ which was a no-op in the scalar algorithm, requires now the solution of a triangular system with multiple right-hand sides (TRSM). Most of the computations happen in

Algorithm 10: Block-Cyclic 1-D parallel LU factorization

```
Every process has index $i_p$;
Every process owns a certain number of blocks of $n_b$ columns;
for $k \leftarrow 1$ to $n/n_b$ do
    $i \leftarrow (k-1) \cdot n_b + 1$;
    $i_p \leftarrow \mod(k-1, n_p)$;
    Let $j_{st}$ the first column I own beyond $i + n_b - 1$;
    Let $j_{en}$ the last column I own (the range $j_{st} : j_{en}$ might be empty);
    if I am process $i_p$ (I own block column $k$) then
        Factor block column
          $A_{i:n,i:i+n_b-1} \Rightarrow (L_{i:n,i:i+n_b-1}, U_{i:i+n_b-1,i:i+n_b-1})$;
        Broadcast send $L_{i:i+n_b-1,i:i+n_b-1}$;
    else
        Broadcast receive $L_{i:i+n_b-1,i:i+n_b-1}$;
    On my block row execute $L^{-1}_{i:i+n_b-1,i:i+n_b-1} \cdot U_{i:i+n_b-1,j_{st}:j_{en}}$;
    $A_{i+n_b:n,j_{st}:j_{en}} \leftarrow A_{i+n_b:n,j_{st}:j_{en}} - L_{i+n_b-1:n,i:i+n_b-1} \cdot U_{i:i+n_b-1,j_{st}:j_{en}}$;
```

the last two steps of the algorithm, which can be formulated as calls to TRSM and GEMM.

8.1.3 A 2-dimensional layout for LU

The final step is to look again at the balance between communications and computations. As we have seen, algorithms 8, 9 and 10 require at each iteration the execution of a BROADCAST collective communication. The optimal implementation of collective operations is a very fascinating and complicated issue in itself; suffice it to say that the optimal choice changes depending on the amount of data being treated, on the network connectivity and on the network parameters. All collective algorithms will however have a completion time that depends on the number of processes involved, in our case n_p.

A better balance between communication and computation can be achieved by adopting a blocked version of the LU factorization, in which the involved matrices are partitioned into square blocks rather than vertical block panels, with each block of size $n_b \times n_b$, as illustrated in the following example with 3×3 blocks:

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ & U_{22} & U_{23} \\ & & U_{33} \end{pmatrix}.$$

Suppose that red is the factorized panel, blue is the row-block to be updated, black is the “active" matrix to be updated and then factorized. It holds:

$$A_{11} = L_{11}U_{11}, \qquad A_{12} = L_{11}U_{12}, \qquad A_{13} = L_{11}U_{13};$$

therefore, we can update the blue blocks by solving triangular systems with multiple right-hand sides (TRSM):

$$U_{12} = (L_{11})^{-1}A_{12} \qquad U_{13} = (L_{11})^{-1}A_{13}$$

and update the “active" matrix by matrix-matrix multiplications (GEMM):

$$\begin{array}{ll} A_{22} - L_{21}U_{12} \longrightarrow \tilde{A}_{22} & \quad A_{23} - L_{21}U_{13} \longrightarrow \tilde{A}_{23} \\ A_{32} - L_{31}U_{12} \longrightarrow \tilde{A}_{32} & \quad A_{33} - L_{31}U_{13} \longrightarrow \tilde{A}_{33} \end{array}$$

If we now consider a logical two-dimensional process grid, the processes can be arranged in a matrix-like topology with P_r rows and P_c columns. A block-cyclic distribution can then be applied both along the rows and along the columns of the global matrix. In this scheme, the global matrix is partitioned into square or rectangular blocks, which are assigned to the processes in a cyclic manner both horizontally and vertically across the process grid. We now have to modify the algorithm by including not only a broadcast of the L block-column (along the rows of the process grid), but also a broadcast of the U block-row (along the columns). The advantage is obviously that each broadcast should now involve only $O(\sqrt{n_p})$ processes.

A 2D block-cyclic distribution provides good load balancing and an efficient computation-to-communication ratio, especially in algorithms where the active matrix shrinks (e.g., LU, QR, Cholesky). A 1D block-cyclic distribution can be advantageous on heterogeneous nodes, enabling efficient use of local accelerators such as GPUs. Finally, pure block distributions (1D or 2D, non-cyclic) are often preferable for uniform workloads like GEMM, where communication overhead is minimal and perfect balance is easily maintained.

An interesting side effect is that we need to be a bit more careful with the choice of the process arrangement. If we have 16 processes, they can be arranged as 1×16, 2×8, 4×4, 8×2 or 16×1 process grids, and the optimum arrangement providing the fastest execution is likely to be either 4×4 or 2×8. If we now want to increase the degree of parallelism we might add 1 process, but since 17 is a prime number, we would be stuck with either a 1×17 or a 17×1 grid, and this is likely to be worse in terms of performance; so we would really like to grow the number of processes in a balanced way.

The final version of the LU factorization can now be presented in algorithm 11.

Obviously a number of details have to be taken care of, such as the precise computation of all the block boundaries, accounting for a size n that is not exactly divisible by the size of the process grid (times n_b), and possibly for a matrix storage that has its starting position $(1, 1)$ stored in a process different from $(0, 0)$. Moreover, we have not discussed explicitly the use of partial pivoting, which is of course necessary for the LU algorithm to proceed as is normal in the serial version.

All of these "details" are handled in the ScaLAPACK software, and you can learn many tricks of the trade by looking at how that software is organized. It is worth to note that, following the modular software design principles established with BLAS standard operations, ScaLAPACK was conceived as a portable and extensible library for dense linear algebra on distributed-memory architectures. To achieve this goal, its developers introduced two fundamental building blocks: the Parallel Basic Linear Algebra Subprograms (PBLAS) and the Basic Linear Algebra Communication Subprograms (BLACS) [46, 47]. The PBLAS extend the functionality of the sequential BLAS to distributed environments by operating on block-cyclically distributed matrices, allowing for scalable implementations of matrix–matrix and matrix–vector operations across multiple processes. The BLACS, in turn, provide a uniform communication layer built on top of a message-passing interface, such as MPI or vendor-specific communication libraries, defining communication contexts, process grids, and point-to-point or collective data transfers in a consistent and hardware-independent way.

Algorithm 11: Block-Cyclic 2-D parallel *LU* factorization

```
Every process has a 2D index $(i_p, j_p)$ in the range
  $(0 : m_p - 1, 0 : n_p - 1)$;
Every process owns a certain number of blocks of size $n_b \times n_b$;
for $k \leftarrow 1$ to $n/n_b$ do
    $i \leftarrow (k-1) \cdot n_b + 1$;
    $i_p \leftarrow \mod (k-1, m_p)$;
    $j_p \leftarrow \mod (k-1, n_p)$;
    Let $i_{st}$ the first row I own which is $\geq i$;
    if my process row index is $i_p$ then
        $\delta_i \leftarrow n_b$;
    else
        $\delta_i \leftarrow 0$;
    Let $i_{en}$ the last row I own ;
    Let $j_{st}$ the first column I own which is $\geq i$;
    if my process col index is $j_p$ then
        $\delta_j \leftarrow n_b$;
    else
        $\delta_j \leftarrow 0$;
    Let $j_{en}$ the last column I own;
    if my process col index is $j_p$ then
        With all processes in my column $j_p$, factor block column
          $A_{i_{st}:i_{en},j_{st}:j_{st}+n_b-1} \Rightarrow (L_{i_{st}:i_{en},j_{st}:j_{st}+n_b-1}, U_{i_{st}:i_{st}+n_b-1,j_{st}:j_{st}+n_b-1})$;
        Copy the $L$ block $\hat{L}_{i_{st}:i_{en},1:n_b} \leftarrow L_{i_{st}:i_{en},j_{st}:j_{st}+n_b-1}$ ;
        Broadcast send $\hat{L}_{i_{st}:i_{en},1:n_b}$;
    else
        Broadcast receive $\hat{L}_{i_{st}:i_{en},1:n_b}$;
    if my process row index is $i_p$ then
        On my block row execute $\hat{L}^{-1}_{i_{st}:i_{st}+n_b-1,1:n_b} \cdot U_{i_{st}:i_{st}+n_b-1,j_{st}:j_{en}}$;
        Copy the $U$ block $\hat{U}_{1:n_b,j_{st}:j_{en}} \leftarrow U_{i_{st}:i_{st}+n_b-1,j_{st}:j_{en}}$;
        Broadcast send $\hat{U}_{1:n_b,j_{st}:j_{en}}$;
    else
        Broadcast receive $\hat{U}_{1:n_b,j_{st}:j_{en}}$;
    $A_{i_{st}+\delta_i:i_{en},j_{st}+\delta_j:j_{en}} \leftarrow A_{i_{st}+\delta_i:i_{en},j_{st}+\delta_j:j_{en}} - \hat{L}_{i_{st}+\delta_i:i_{en},1:n_b} \cdot \hat{U}_{1:n_b,j_{st}+\delta_j:j_{en}}$;
```

8.2 Evolution of Parallel Dense Linear Algebra Software

As we already pointed out in the introduction, the evolution of dense linear algebra software has not only mirrored advances in computer architectures but has also driven theoretical progress in numerical linear algebra, encouraging researchers to reformulate classical algorithms to better exploit modern hardware while preserving numerical stability. The earliest libraries, EISPACK and LINPACK, developed in the 1970s and early 1980s, provided standard solutions for eigenvalue problems and linear systems on vector and serial architectures, using Fortran implementations and column-major storage schemes [11, 48].

As hierarchical memory systems became dominant, the community recognized the need for algorithms that minimized data movement and improved cache reuse. This led to the development of LAPACK (Linear Algebra PACKage) [49], which reimplemented EISPACK and LINPACK in a blocked form. The blocking technique enabled the use of Level 3 BLAS operations, significantly improving performance on shared-memory architectures. The shift from vector to cache-based machines also inspired extensive numerical analysis studies on the reformulation of Gaussian elimination and orthogonal factorizations, emphasizing both computational efficiency and algorithmic stability.

With the emergence of distributed-memory architectures in the 1990s, ScaLAPACK extended LAPACK's concepts to large-scale parallel systems, employing MPI-based message passing and block-cyclic data distribution to balance computation and communication [50]. These developments further stimulated the analysis of communication-avoiding algorithms, a major research direction that linked algorithmic theory and parallel performance.

The rise of multicore and manycore architectures in the 2000s prompted a shift toward fine-grained, task-based parallelism. PLASMA (Parallel Linear Algebra for Scalable Multi-core Architectures) introduced dynamically scheduled tile algorithms to fully exploit multicore systems [51], while MAGMA (Matrix Algebra on GPU and Multicore Architectures) extended these ideas to heterogeneous CPU–GPU systems, merging algorithmic redesign with hardware-aware optimization [52].

The latest generation, SLATE (Software for Linear Algebra Targeting Exascale), represents a comprehensive rethinking of dense linear algebra for exascale and heterogeneous systems, incorporating asynchronous execution, fault tolerance, and communication-avoiding strategies. Beyond software engineering, SLATE and related efforts embody decades of co-evolution between numerical analysis and high-performance computing, where concerns for accuracy, stability, and scalability remain central to algorithmic innovation [53].

8.3 Sparse linear algebra data distribution

The data layout for sparse matrix problems in parallel is driven by the same considerations for parallel efficiency that we have seen before, but the outcome of the analysis will be very different.

8.3.1 A simple iterative solver revisited

Let us now go back to the Richardson iteration 7.1:

$$x_{k+1} = x_k + \omega(b - Ax_k). \tag{8.1}$$

As we have seen, the convergence is measured against the norm of the residual, and the other operators needed are sums of vectors and matrix-vector products.

Thus we need to figure out a way to perform these kernels, and most importantly the matrix-vector product $y \longleftarrow Ax$, and choosing a parallel data layout will be critical in this regard[†].

It is now time to point out a few things:

- When we compute entry i of the output vector, we are multiplying row $A_{i,:}$ of the matrix by the vector x, and this multiplication will only need to involve the nonzero coefficients;
- By the same token, for each entry i of the output vector y, only a subset of the entries of the x vector will actually be involved in the computation.

We will distribute the coefficient matrix for the linear system based on the "owner computes" rule: the variable associated to each mesh point is assigned to a process that will own the corresponding row in the coefficient matrix and will carry out all related computations. This allocation strategy is equivalent to a partition of the discretization mesh into *sub-domains*, and is naturally associated with a 1-dimensional process grid structure, with the matrix being allocated to processes by (blocks of) rows.

Is this a sensible idea?

Well, actually yes (in most cases). Indeed, when we discussed the 1-dimensional distribution of section 8.1.2, we split the matrix by *columns*: this strategy will most likely not work here because (as already mentioned) each row of the sparse matrix has a number of nonzero entries that is typically bounded by a constant k, independently of n. Whenever that constant is small, and for most PDE discretization schemes k is only a few tens at most, there are too few arithmetic operations to make it worth splitting them between different processes. Thus we will distribute *rows* of the matrix onto our parallel processes[‡].

[†] This is a very simple iteration, and for the time being we will not search for alternatives with faster convergence; anyway, other parallel iterative algorithms tend to share the same considerations discussed in the sequel.

[‡] There are cases where the number of nonzeros per row warrants distribution across multiple processes, but they tend to come from different application domains and require different problem solution strategies.

8.3.2 Basic observations

Our computational model implies that the data allocation on the parallel distributed memory machine is guided by the structure of the physical model, and specifically by the discretization mesh of the PDE.

Each point of the discretization mesh will have (at least) one associated equation/variable, and therefore one index i. We say that point i *depends* on point j if the equation for a variable associated with i contains a term in j, or equivalently if $a_{ij} \neq 0$. After the partition of the discretization mesh into *sub-domains* assigned to the parallel processes, we classify the points of a given sub-domain as following.

Internal. An internal point of a given domain *depends* only on points of the same domain. If all points of a domain are assigned to one process, then a computational step (e.g., a matrix-vector product) of the equations associated with the internal points requires no data items from other domains and no communications.

Boundary. A point of a given domain is a boundary point if it *depends* on points belonging to other domains.

Halo. A halo point for a given domain is a point belonging to another domain such that there is a boundary point which *depends* on it. Whenever performing a computational step, such as a matrix-vector product, the values associated with halo points are requested from other domains. A boundary point of a given domain is usually a halo point for some other domain[§]; therefore the cardinality of the boundary points set determines the amount of data sent to other domains.

Overlap. An overlap point is a boundary point assigned to multiple domains. Any operation that involves an overlap point has to be replicated for each assignment.

Overlap points do not usually exist in the basic data distributions; however they are a feature of Domain Decomposition Schwarz preconditioners which are the subject of related research work.

We denote the sets of internal, boundary and halo points for a given subdomain by $\mathcal{I}$, $\mathcal{B}$ and $\mathcal{H}$. Each subdomain is assigned to one process; each process usually owns one subdomain, although the user may choose to assign more than one subdomain to a process. If each process i owns one subdomain, the number of rows in the local sparse matrix is $|\mathcal{I}_i| + |\mathcal{B}_i|$, and the number of local columns (i.e. those for which there exists at least one non-zero entry in the local rows) is $|\mathcal{I}_i| + |\mathcal{B}_i| + |\mathcal{H}_i|$.

This classification of mesh points guides the naming scheme that we adopted in the library internals and in the data structures. We explicitly note that "Halo" points are also often called "ghost" points in the literature.

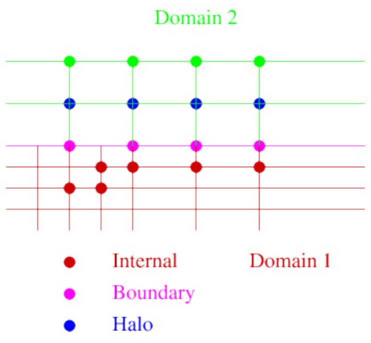

Figure 8.3: Point classfication.

[§] This is the normal situation when the pattern of the sparse matrix is symmetric, which is equivalent to say that the interaction between two variables is reciprocal. If the matrix pattern is non-symmetric we may have one-way interactions, and these could cause a situation in which a boundary point is not a halo point for its neighbour.

8.3.3 Sparse Matrix-Vector Product in Parallel

The classification of points that we saw before in Fig. 8.3 formulated in terms of the discretization mesh can be carried over in terms of matrix structure as in Fig. 8.4. The pink area corresponds to the local coefficients, the green area corresponds to the halo. There exist a *Surface to Volume effect*: for "sensible" data distributions, most of the nonzeros are in the pink area, whilst the green area is almost empty. In particular, the green area contains many column sections that are completely empty, meaning that their nonzeros are outside the range of rows. So, even if it looks like the matrix-vector product requires a copy of the full vector x, what is really needed is only a small subset of the entries of x from other processes (since obviously there is no actual need to perform multiplication by zeros).

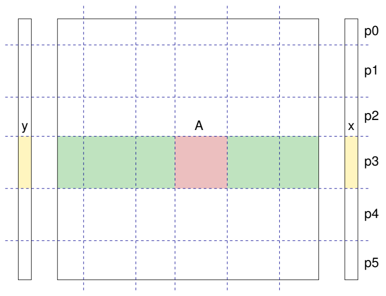

Figure 8.4: Matrix structure.

To compute the SpMV, we then have to retrieve values of X corresponding to entries in the green area with a *halo data exchange*.

What constitutes a "sensible" data distribution?

8.3.4 Graph partitioning

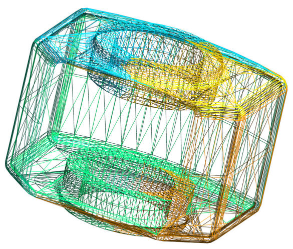
Figure 8.5: A (partitioned) graph.

We begin by looking at the implications of the "owner computes" paradigm as applied to parallel computations of sparse matrix-vector products.

Let us first recall the notion of a *graph* which is an entity composed of two sets: $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, where

$$\begin{aligned} \mathcal{V} &= \{v_1, \ldots, v_n\} \\ \mathcal{E} &\subseteq \mathcal{V} \times \mathcal{V}. \end{aligned}$$

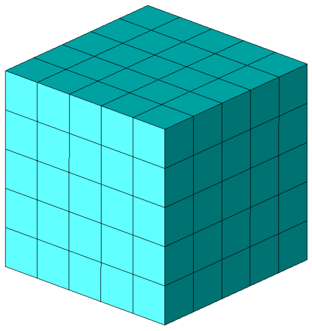
Figure 8.6: A structured mesh.

The set $\mathcal{V}$ is called the vertex set, whereas $\mathcal{E}$ is the edge set; every edge connects two vertices. The degree of a vertex is defined by the number of edges having that vertex as an endpoint.

We can now state a rather simple fact: there is an isomorphism between a (square) matrix and a graph where

- To each row (column) i there corresponds a vertex v_i;
- To each coefficient a_{ij} there corresponds an edge $e_{ij} = (v_i, v_j)$;

A sparse matrix will then be associated with a graph that is not *fully connected*, that is, there exist (many) pairs of vertices that are not linked directly by an edge. In some cases a graph may have a value associated with either the nodes or the edges.

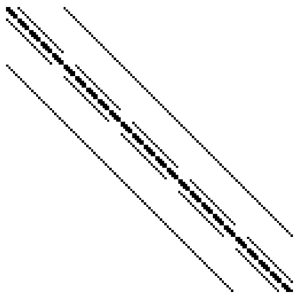
Figure 8.7: The resulting pattern.

The correspondence between a matrix and a graph is established once we associate an index i, the variable x_i and the matrix row $a_{i,:}$ with one of the graph nodes; using this association, we can easily see that the amount of work needed to compute y_i in a matrix-vector product is proportional to the number of entries in rows $a_{i,:}$, that is, to the degree of node i.

The first objective in establishing an allocation of data to multiple processes should by now be clear: we want to allocate vertices (rows of the matrix) to processes in such a way that the sum of their degrees (the number of nonzeros) and therefore the amount of work in a matrix-vector product is spread uniformly across the processes. If the number of nonzeros per row is roughly constant, this is essentially equivalent to allocating an equal number of vertices to each process.

If we now look back at Fig. 8.3, we can immediately see the second objective we have to strive for: every boundary point needs to know the value of the entry of x associated with (at least) one halo point. Thus, we will need to exchange an amount of data proportional to the number of halo points, and we clearly want to minimize this overhead.

We can now state our problem:

> Given a graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, find a partition of $\mathcal{V}$ into subsets $\mathcal{V}_i$ such that $\mathcal{V} = \bigcup \mathcal{V}_i$, and define a set $\mathcal{H} = \{e = (i, j) \in \mathcal{E} : i \in \mathcal{V}_p, j \in \mathcal{V}_q, \quad p \neq q\}$ such that:
>
> ▶ $|\mathcal{V}_i| = |\mathcal{V}_j| \qquad \forall i, j;$
>
> ▶ $|\mathcal{H}|$ is minimized.

Note that the first criterion, equal distribution of vertices, may not be attainable in practice; if the number of nonzeros per row has a large variance, we may associate a weight to each vertex, and the partition of the vertices should then strive to obtain a uniform distribution of weight.

This is called a *graph partitioning* problem, and it is unfortunately an $\mathcal{NP}$-complete problem, meaning that it is too expensive to solve exactly; it is therefore necessary to employ *heuristics* to get at a reasonable solution in a reasonable amount of time (fig 8.10).

A discussion of the properties of $\mathcal{NP}$-complete problems is outside the scope of the present work; more details can be found in [54, 55].¶

What is the effect of a graph partition algorithm? Essentially it splits the matrix into multiple blocks of rows, but there may be multiple blocks assigned to a given process; if we list the matrix rows in the order of the processes we have implicitly applied a renumbering of the equations/variables (fig 8.11).

For example, if we have a 2-dimensional rectangular domain, then the obvious way to partition the domain is to split into subrectangles, and actuall this is what a graph partitioner heuristics will likely do (see Fig. 8.12); however, if we apply a natural numbering to the mesh points (say, by rows or by columns), then each subdomain will receive many blocks of rows.

Applying a renumbering of the mesh points is equivalent to applying a symmetric permutation to the coefficient matrix A; this has only very minor effects on the numerical convergence of an iterative solver, but it could affect performance in rather significant ways. In any case, applying a graph partitioner is equivalent to applying a renumbering and then cutting the matrix into stripes.

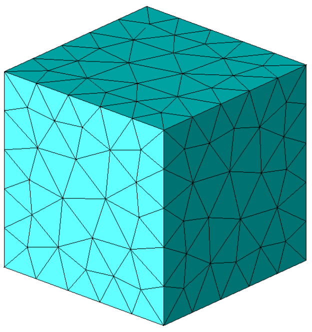
Figure 8.8: An unstructured mesh.

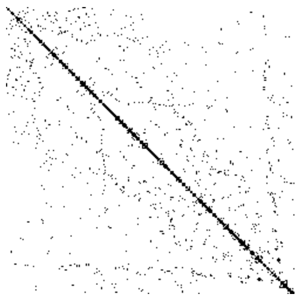
Figure 8.9: The resulting pattern.

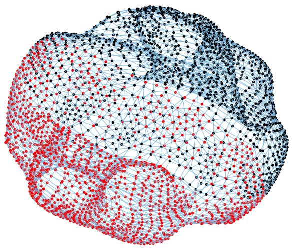
Figure 10: A graph partition.

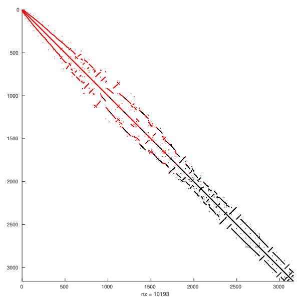

Figure 8.11: A matrix partition.

¶ Strictly speaking, it is not known whether any $\mathcal{NP}$-complete problem admits an efficient algorithm for its solution, but it can be proven that if such an efficient algorithm exists for any one of the problems in this very large class, then an efficient algorithm exists for all other problems; since none is known, it is deemed very unlikely that any such algorithm actually exists. The question of the existence of an efficient algorithm, that is, whether we actually have $\mathcal{P} = \mathcal{NP}$, is the most famous open problem in theoretical computer science.

Generating a discretization mesh is a non trivial matter in its own right, and many algorithms generate a numbering that may appear rather strange and involved; in any case, it is worth remembering that partitioning a graph requires usage of heuristics, and these may well depend on the initial numbering. An example of a partition is shown in fig. 8.13, for a mesh designed to study the thermal diffusion in a mechanical part.

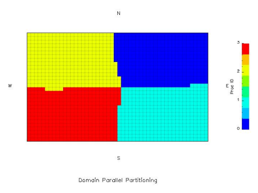

Figure 8.12: A 2D domain partition.

A final observation is now in order. If we look at "sensible" data distributions, such as those in fig. 8.12 and 8.13, we can notice that the data exchange happens at the boundaries of the subdomains, and these boundaries grow as the "surface" of the subdomain, whereas the amount of computations grows as the "volume" of the subdomain. This ratio between communication and computation, known as the surface-to-volume effect, is a fundamental factor influencing the scalability and overall efficiency of parallel algorithms. What is a "non-sensible" data distribution? We show one in fig. 8.14: all of the definitions and strategies we have discussed will "work" (from a logical point of view), but the communication overhead will be ridiculous.

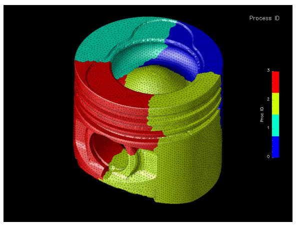

Figure 8.13: A complex domain partition.

8.3.5 The correspondence between indices and processes

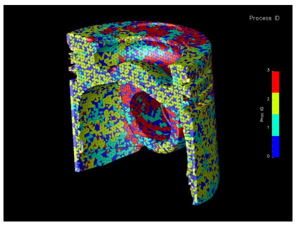

Figure 8.14: A rather silly domain partition.

As mentioned before, an index space $\mathcal{I} = \{i, i = 1 \dots n\}$ can be *distributed* over a set of processes, as long as we can figure out the *mapping* between the abstract index i and its physical counterpart, say local index j on process p. As we shall see, this is particularly important for sparse matrices.

Let us begin by restating our situation: *There exists an index set spanning a problem space, and this index set is partitioned among multiple processors.* This partition can be realized in many different ways, for example:

- serial/replicated distribution, where each process owns a full copy and no communication is needed (this also cover the case of a serial run);
- block distribution, where each process owns a subrange of the indices;
- list assignment, where we have for each process a list specifying the indices it owns;
- global list assignment, where we have a list specifying for each index its owner process, replicated on all processes.

Encapsulating these variations under a uniform outer shell allows for having a single entry point with no need for conditional compilation in the case in which the user is actually running in serial mode.

Given a data distribution, we need to answer questions that hinge upon relating the "global" indices to their "local" counterparts:

1. Which global index is the image of a certain local one?
2. Which processor owns a certain global index, and to what local index does it correspond?

Notice that the first question is much easier to answer, because when we assign a set of indices to a process, and define their local counterparts, we also keep track of their global counterpart; this only requires an amount of memory that is proportional to the local number of points, and is therefore scalable. The second question is instead much more complex, and can be split in three parts:

1. If a global index is owned by a certain process, that process can answer the query *provided* it has a mapping from global to local indices; however, finding any one of the global indices may require searching through a set that is potentially disordered, therefore proper data structures should be designed to facilitate this task;
2. If the query originated from a process such that the global index corresponds to one of the halo indices, then the process will likely also know which other process owns it, but it will not necessarily know what local index is in use on that other process;
3. If the query is about an arbitrary global index that may be owned by any process, it may be necessary to have an expensive search phase in which all processes cooperate.

Knowing the location of indices is necessary to retrieve the values associated with the halo of any given subdomain; thus, it is normally the case that the halo indices are listed in a preprocessing step, defining a set of communications that need to take place at every matrix-vector product to guarantee that each process has access to the entries of x it needs, and these entries are up to date. A very useful set of auxiliary information for this purpose would be building a subdomain adjacency graph, that is, a graph that has one vertex for each subdomain, and one edge between two subdomains whenever there is a direct exchange between the two.

8.3.6 Data Exchange for Matrix-Vector Products

We mentioned in Sec. 8.3.3 the need to retrieve entries of the x vector from the other processes (see also fig. 8.4). The sparse case is however quite different from the dense parallel BLAS operations: we want to transfer the minimum possibile amount of data, but the minimum set of entries that needs to be transferred depends on the *pattern* of the sparse matrix A, that is, on the nummber and location of the nonzero coefficients. For "sensible" matrix patterns and data distributions, this minimum amount of data is much smaller than the total number of entries in the vector x; moreover, the positions of the entries remain the same for as long as we are dealing with the same matrix pattern.

In the context of an iterative solver, the data exchange will happen at every matrix-vector product, which happens at least once per iteration; it is therefore convenient to prepare auxiliary data structures detailing the exact locations of the entries to be sent/received, and to organize the send/receive operations between each pair of processes. The time it takes to prepare the auxiliary structure will be amortized over the multiple matrix-vector products in the course of the system solution process.

In the MPI jargon, this operation might be called a *persistent variable all-to-all neighborhood collective communication*:

All-to-all: Each process may send and receive to/from any other;
Variable: The amount of data is specific to any given send/receive pair;
Persistent: The operation is repeated involving the values associated with the same set of vector indices, multiple times;

Neighborhood: Many pairs of processes would exchange an empty set of data, and we can obviously drop the send/receive pair; thus, each process will only actually communicate with a subset of the other processes (its neighborhood).

In our library this is called a *Halo data exchange*.

The exact details, such as the specific lists of indices, are determined by the pattern of the sparse matrix A; this pattern will be common to all matrices built on the same discretization mesh with the same discretization method. In our software, the lists needed to organize the data exchange are stored together with the *index map* in a *communication descriptor* object, which therefore encapsulates all the necessary information.

Since the communication descriptor depends on the matrix pattern, it cannot be fully set up when we initially allocate indices to processes, but requires going over all the indices in the sparse matrix pattern, either implicitly (during the build phase of the matrix itself including the coefficients), or explicitly (listing just the positions of the entries).

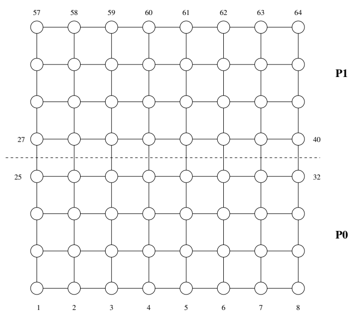

Figure 8.15: Halo exchange.

Consider the discretization mesh depicted in fig. 8.15, partitioned among two processes as shown by the dashed line; the data distribution is such that each process will own 32 entries in the index space, with a halo made of 8 entries placed at local indices 33 through 40. If process 0 assigns an initial value of 1 to its entries in the x vector, and process 1 assigns a value of 2, then after a call to `psb_halo` the contents of the local vectors will be as shown in table **??**.

I	GLOB(I)	X(I)	I	GLOB(I)	X(I)
1	1	1.0	1	33	2.0
2	2	1.0	2	34	2.0
3	3	1.0	3	35	2.0
4	4	1.0	4	36	2.0
5	5	1.0	5	37	2.0
6	6	1.0	6	38	2.0
7	7	1.0	7	39	2.0
8	8	1.0	8	40	2.0
9	9	1.0	9	41	2.0
10	10	1.0	10	42	2.0
11	11	1.0	11	43	2.0
12	12	1.0	12	44	2.0
13	13	1.0	13	45	2.0
14	14	1.0	14	46	2.0
15	15	1.0	15	47	2.0
16	16	1.0	16	48	2.0
17	17	1.0	17	49	2.0
18	18	1.0	18	50	2.0
19	19	1.0	19	51	2.0
20	20	1.0	20	52	2.0
21	21	1.0	21	53	2.0
22	22	1.0	22	54	2.0
23	23	1.0	23	55	2.0
24	24	1.0	24	56	2.0
25	25	1.0	25	57	2.0
26	26	1.0	26	58	2.0
27	27	1.0	27	59	2.0
28	28	1.0	28	60	2.0
29	29	1.0	29	61	2.0
30	30	1.0	30	62	2.0
31	31	1.0	31	63	2.0
32	32	1.0	32	64	2.0
33	33	2.0	33	25	1.0
34	34	2.0	34	26	1.0
35	35	2.0	35	27	1.0
36	36	2.0	36	28	1.0
37	37	2.0	37	29	1.0
38	38	2.0	38	30	1.0
39	39	2.0	39	31	1.0
40	40	2.0	40	32	1.0

Appendix

A
The Errors of Our Way

When a physical problem goes through the modeling process to be solved on a computer, it is almost always the case that the computed solution will be only an approximation to the "true" mathematical solution. Many different factors concur to this result:

- The mathematical model, that is, the set of equations describing the phenomenon, is only valid to within a certain degree of approximation;
- The inputs to the problem consist of experimental data, known through measurements which are inherently affected by a certain amount of error;
- The data from the measurements is taken into the computer, where real numbers are represented in the floating-point number system, with its limitations;
- The techniques employed to solve the problem may require:
  1. Truncation of a series which is capable of giving the exact solution only in an asymptotic sense;
  2. Usage of a heuristics which produces only an approximate solution, because the strategy needed to attain the exact solution is too expensive.
- The process of executing an algorithm undergoes rounding errors, thereby introducing further uncertainty.

These statements are only qualitative in nature; to give them a more precise, quantitative meaning, it is necessary to present some material on computer arithmetic and on error analysis.

First of all it is necessary to define formally the very basic concepts of *absolute* and *relative* errors. Given an exact quantity x and an approximation $\hat{x} = x + \delta x$ we define:

Absolute error:

$$E_{\text{abs}}(\hat{x}) = |x - \hat{x}| = |\delta x|$$

Relative error:

$$E_{\text{rel}}(\hat{x}) = \frac{|x - \hat{x}|}{|x|} = \frac{|\delta x|}{|x|}$$

Among the two, the relative error is more useful in most scientific calculations, because it is scale-independent: scaling both $x \longrightarrow \alpha x$ and $\hat{x} \longrightarrow \alpha \hat{x}$ leaves $E_{\text{rel}}(\hat{x})$ invariant.

Most engineers will discuss of computed quantities by stating that they have a certain number p of correct significant digits. This concept is intuitively clear; and yet, a formal definition is surprisingly difficult. First of all, let us recall that the significant digits are digits from the first nonzero to the last; thus 1.123 has four, while 0.012 has two significant digits. Consider now the following two examples:

$$\begin{aligned} x &= 1.00000, & \hat{x} &= 1.00499, & E_{\text{rel}}(\hat{x}) &= 4.99 \times 10^{-3}, \\ x &= 9.00000, & \hat{x} &= 8.99899, & E_{\text{rel}}(\hat{x}) &= 1.12 \times 10^{-4}; \end{aligned}$$

by any reasonable definition we should have three correct significant digits in both cases, yet the relative error differs by a factor of 44.

A seemingly sensible definition would be: *an approximation $\hat{x}$ has p correct significant digits if x and $\hat{x}$ rounded to p digits produce the same result*.

However this definition leaves the door open to some rather curious situations; consider in fact the two numbers

$$x = 0.9949, \quad \hat{x} = 0.9951.$$

Using the previous definition, we have that $\hat{x}$ has either one or three correct digits but not two!

Thus, while the number of correct significant digits can be a useful tool in visualizing the situation, it is at best a crude quantitative measure, and for precise statements it is far better to use the relative error.

A.1 Numbers in a Computer

Numbers in a computer are represented as finite strings of binary digits, or bits. Since the strings of digits are finite, it is obviously impossible to store arbitrary real numbers; at most, it will be possible to store in a computer a subset of the reals, indeed a subset of the rationals.

Consider first a simple representation method in which the numbers are composed of a string of digits in a certain base β as follows:

$$r = \pm d_{k-1} \cdots d_1 d_0 . d_{-1} d_{-2} \cdots d_{-t}, \qquad \text{(A.1)}$$

with k digits for the integer part and t digits for the fractionary part, where for each digit we have $0 \le d < \beta$. Now, by definition the value is given (apart from the sign) by the following expression:

$$r = d_{k-1} \times \beta^{k-1} + d_{k-2} \times \beta^{k-2} + \cdots + d_o \times \beta^0 + d_{-1}\beta^{-1} + \cdots + d_{-t}\beta^{-t}.$$

Let us now ask the question:

> Which numbers are exactly represented in this number system?

The answer should be quite clear: those rational numbers whose expansion in base β is finite, with a number of digits after the point less than or equal to t. Ignoring for the time being the integer part, a direct translation of the above statement implies that:

$$r = \frac{p}{q} = .d_{-1} \cdots d_{-t};$$

now, if this is exact, it means that multiplying by an appropriate power of the base we should get an integer number:

$$r \times \beta^t \in \mathbb{Z}.$$

This simple fact also translates into

$$\frac{p}{q} \times \beta^t \in \mathbb{Z},$$

but since p/q is a reduced representation of a rational number, i.e. p and q are mutually prime, it follows that

$$\frac{\beta^t}{q} \in \mathbb{Z}.$$

For this to be true we must have that

> The prime factors of q form a subset of the prime factors of β.

This simple observation implies that it is impossible to represent exactly the number $1/10$ on a binary computer ($\beta = 2$)!

The simple representation of equation (A.1) is called *fixed-point* and it was used in the early days of electronic computing. Indeed, the seminal book [56] contains many examples of error analysis in the context of fixed-point operations. At first sight it appears to be a very natural choice, but a more careful consideration (aided by practical experience) shows some rather substantial disadvantages. To appreciate this point, let us first notice that given a finite number of digits available $p = k + t$, it is clear that the range of the representable numbers

$$FXPF = [-(\beta^k - \beta^{-t}), (\beta^k - \beta^{-t})],$$

is substantially smaller than the set of integers that can be represented with the same number of digits

$$INTS = [-(\beta^p - 1), (\beta^p - 1)];$$

we have paid a price in the range to achieve the finer β^{-t} spacing between any two consecutive fixed point numbers accrued by the t fractional digits. Fixed point numbers are usually defined with $k = 0$, so that the range is $[-1, 1]$; this is exceedingly cumbersome by modern standards, since all data must be preprocessed and scaled to fit into this range.

Another less evident problem is the issue of how much error do we actually have in the representation of data. Since only a subset of the real numbers is represented, it is necessary to approximate each real number (within the range) with a fixed-point number; taking the obvious step of rounding to the nearest neighbour, we find that the absolute error is bounded by

$$|\delta x| \leq \frac{1}{2}\beta^{-t}$$

over the entire range. The relative error, however, tells us a different story; consider the first nonzero fixed-point number β^{-t}; the absolute error in its representation is bounded by $1/2\beta^{-t}$, which means that the relative error can be up to 0.5, or 50%. The relative error slowly decreases until at the other extreme of the range, the largest fractional number representable is $1-\beta^{-t}$, and its associated relative error is practically equal to the absolute error $1/2\beta^{-t}$; if t is large, then the difference between the largest and smallest value of the relative error can be very significant. If we now ask the question of how best to use an increase in hardware resources, and specifically how best to allocate P available digits within a number, we have to choose between improving the error by increasing t or improving the range by increasing k. While the absolute error will improve with increasing t, the relative error will improve unequally over the range: at large values, an already small relative error is further diminished, while there is little to no effect at small values.

The above considerations make it clear that it is desirable to have a better scheme for representing real numbers on a digital computer, which will be introduced in the next subsection. Fixed-point number systems survive today only in selected application areas, for instance in special processors for Digital Signal Processing (DSP). One interesting usage of fixed point numbers is in the TEX typesetting system used to typeset this work and most articles and books in mathematics [57].

A.1.1 Floating-point Numbers

The floating-point representation of real numbers is today used universally on general purpose computing devices. Its origin may be traced to the scientific notation for real numbers; it is based on the idea of *normalization* of a real number. As an example, the (base 10) number −2.71828 can be represented in scientific notation as

$$-.271828 \times 10^1,$$

with the following components:

− is the sign of the number;

.271828 is the fractional part; it is shifted so that the first nonzero digit comes right after the point;

1 is the exponent.

In principle any base can be chosen; of course electronic computers are binary, but this would not rule out the choice of, say, $\beta = 8$; indeed, the IBM 360 mainframe architecture defined its native floating-point format with $\beta = 16$, with each hexadecimal digit being composed of four bits.

Formally, a floating point number is represented as follows:

$$(s, e, f) = \pm\beta^e \left(\frac{d_1}{\beta^1} + \frac{d_2}{\beta^2} + \dots \frac{d_t}{\beta^t} \right)$$

with $f < 1$ represented on t figures. This is actually equivalent to the representation

$$(s, e, f) = \pm f \times \beta^{e-t},$$

a form perhaps easier to work with. The number is supposed to be *normalized*, i.e. $d_1 \neq 0$. To see the advantage of this system in terms of maximizing the effective range with the available digits, consider that

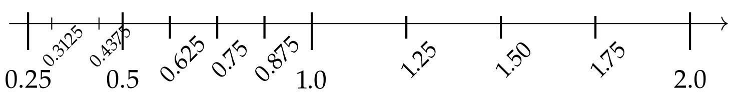

Figure A.1: An example floating-point number system

if we are using k digits for the exponent, then the largest representable number would be

$$(1-\beta^{-t})\times\beta^{\beta^{k}-1}$$

a large improvement over the limit β^k-1 that would apply to fixed-point numbers with k integer digits. Notice that the range of the floating-point numbers is logically partitioned into sections corresponding to the different values of the exponent e; when $\beta=2$ these sections are called octaves*. Also, normally the k digits of the exponent fields are employed to represent negative exponents as well as positive ones; thus, the effective range would be closer to $\beta^{\beta^{k-1}-1}$. In the sequel $\mathrm{fl}(x)$ will be used to denote the best approximation to x in the floating-point number system in use; the set of all floating point numbers will be denoted by F.

An example will now help visualize the situation; if we choose $\beta=2, t=3$, $e_{min}=-1$, $e_{max}=3$ we can represent the following set of numbers:

$$0, 0.25, 0.3125, 0.3750, 0.4375, 0.5, 0.625, 0.750, 0.875$$
$$1.0, 1.25, 1.50, 1.75, 2.0, 2.5, 3.0, 3.5, 4.0, 5.0, 6.0, 7.0,$$

depicted graphically in Figure A.1 The relative error in the machine representation of numbers can be bounded in terms of the quantity ϵ_m, or *machine epsilon*, which is the distance between the number 1.0 and the next floating point number 1.0^+. In a normalized number system we must have

$$\begin{aligned}\mathrm{fl}(1.0) &= \beta^1\times(.10\cdots0_t)\\ \mathrm{fl}(1.0^+) &= \beta^1\times(.10\cdots1_t)\end{aligned}$$

hence

$$\epsilon_m = 1.0^+ - 1.0 = \beta^1\times(.00\cdots1_t) = \beta^{(1-t)}. \qquad \text{(A.2)}$$

In the sample floating point system we are using, we have $\epsilon_m = 2^{(1-3)} = 0.25$, which can be confirmed visually. Let us now consider the relative error in the floating point representation; assuming that the process of approximating the number x by $\mathrm{fl}(x)$ is performed by rounding to the nearest neighbour, it is easy to see that the largest relative error will be incurred when we approximate with 1.0 the number $1.0+\epsilon_m/2$; for this number we have

$$\left|\frac{\mathrm{fl}(x)-x}{x}\right| \approx \frac{1}{2}\beta^{(1-t)}.$$

The smallest error will be incurred at the other extreme of the octave where we approximate $2.0-\epsilon_m/2$ with 2.0, thus giving

$$\left|\frac{\mathrm{fl}(x)-x}{x}\right| \approx \frac{1}{2}\beta^{-t}.$$

* The name octave is derived from musical notation: two notes with the same name in consecutive octaves have a base frequency in the ratio 1 : 2.

From the above considerations we may derive the following rule

$$\mathrm{fl}(x) = x(1+\delta) \qquad |\delta| \le u,$$

where u is the *unit roundoff*, and we have

$$u = \frac{1}{2}\beta^{(1-t)}.$$

Moving from one section to the next implies that the absolute spacing among the numbers will be multiplied by a factor of β, but the *relative* error for two floating-point numbers with the same f and different e will be exactly the same. Put it another way, the relative error in a section will vary by a factor of β, decreasing over the section, and this pattern will be repeated identically over the various section composing the range of the floating-point numbers. The variation between the relative error at the beginning and at the end of a section is called "wobbling"; since the wobble factor is equal to β, it is advantageous to have as small a value as possible, so that the relative error oscillates in a band as tight as possible. This is a very important reason to prefer systems with $\beta = 2$.

A problem with this sort of number system is apparent from Figure A.1: there is a "hole" around the origin, which is caused by the requirement for the numbers to be normalized, i.e. for d_1 to be nonzero. Relaxing this requirement allows to obtain a uniform spacing around the origin, and introduces the so-called "denormalized" numbers having $e = e_{\min}, d_0 = 0$; in our example these are

$$0.0625, 0.125, 0.1875.$$

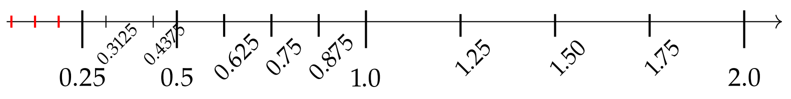

Figure A.2: An example floating-point number system with highlighted denormalized numbers

A.1.2 The IEEE 754 Floating-Point Standard

The IEEE Standard 754 was published in 1985, at the end of a design process that lasted through many years. It defines a binary floating point system designed to enable the development of robust and portable numerical software; it is the floating-point format of choice for practically all computers in common use.

The IEEE format conforms to the model described in section A.1.1. It is a binary system, that is $\beta = 2$; therefore the first digit for a normalized number can only be 1 and can be assumed. For this reason it is often called a "phantom" bit, since it is not stored explicitly. The format currently specifies four variations of differing size:

Single precision (32 bits) $t = 23 + 1$ $e_{min} = -127$, $e_{max} = 127$, $u = 6.0 \times 10^{-8}$, range $10^{\pm 38}$

Double precision (64 bits) $t = 52 + 1$ $e_{min} = -1023$, $e_{max} = 1023$, $u = 1.1 \times 10^{-16}$, range $10^{\pm 308}$

Extended precision (80 bits) $t = 63 + 1$ $e_{min} = -16383$, $e_{max} = 16383$, $u = 5.4 \times 10^{-20}$

Quad precision (128 bits) $t = 112 + 1$ $e_{min} = -16383$, $e_{max} = 16383$, $u = 9.6 \times 10^{-35}$

The exponent is represented with an unsigned number to which a displacement is implicitly subtracted; as an example, in single precision the exponent field is 8-bits wide, and the displacement is -127, so that the minimum representable value is -127. The maximum normalized value is given by $e = 254 - 127 = 127$. When the exponent part is at its minimum possible value the phantom bit is assumed to be 0, and therefore we are including unnormalized numbers, often called “denormals”.

The standard provides means to represent so-called “symbols”, which comprise infinity and “Not a Number” (NaN). An infinite value is represented by a number having the maximum possible value for the exponent field and a fractional part equal to 0; for a single precision number this happens when the actual exponent is $e = 255 - 127 = 128$. When the exponent is at its maximum value but the fractional part is nonzero, then the bit pattern is interpreted as a NaN (Not a Number); this is used to signal the result of an invalid operation. The following list summarizes the possible occurrences of these exceptional values:

- $1/0 = (-1)/(-0)\infty$
- $(-1)/0 = 1/(-0) = -\infty$
- $0/0 = \infty - \infty = \infty/\infty = 0 \times \infty = \sqrt{-1} = NaN$.

As we can se the sign of zero influences the outcome of a division, whereas it is not detectable in e.g. a comparison. IS THIS COMPLETELY CORRECT???.

A.2 Floating-point Arithmetic Properties

Floating point arithmetic is universally employed in modern computer systems; thus the error analysis of algorithms is usually carried out referring to a model of its behaviour. The most widely used is the so-called *standard model* which is summarized in the following equation:

$$\mathrm{fl}(x \textcircled{\scriptsize op} y) = (x \textcircled{\scriptsize op} y)(1+\delta), \quad |\delta| \le u, \ \textcircled{\scriptsize op} = + - */ \ x, y \in F. \tag{A.3}$$

Note that the assumption that the operands x, y belong to F does *not* guarantee that the *exact result* will also belong to F. That the result is exactly representable in F is a fact that is only true for certain combinations of operation and operands. In particular it is obiously true of multiplications and divisions by β, since these operations amount to adding/subtracting into the exponent field (barring overflow or underflow). A less immediate result, in base 2, is the following

Theorem A.2.1 (Sterbenz) *If x and y are floating point numbers with $y/2 \le x \le 2y$ and $x - y$ does not underflow, then* $\mathrm{fl}(x - y) = x - y$.

Proof. See [58]. □

In any case, let us repeat that any occurrence of an exact result is a very rare exception among floating computations.

The IEEE 754 arithmetic standard requires that the result of any individual aritmetic operatiom be the same as the rounding of the exact result, for instance:

$$u \oplus v = \text{round}(u + v);$$

this requirement is more stringent than that of the standard model summarized in (A.3). It also implies that if the "true" result is exactly representable, it must be correctly reproduced.

An implication of the rounding requirement is that the implementor of the arithmetic subsystem needs to provide more digits for storing the intermediate results than are specified in the floating point format. This can be shown by a rather simple example, employing the same sample floating point format presented in sec. A.1.1. Consider the computation of $1.0 - 0.875$; employing 4 bits for the intermediate results, the computation is carried out as:

$$\begin{array}{lcll} 2^1 \times 0.100 & - & 2^1 \times 0.100 & - \\ 2^0 \times 0.111 & \longrightarrow & 2^1 \times 0.0111 & \\ \cline{3-4} & & 2^1 \times 0.0001 & = \; 2^{-1} \times 0.010. \end{array}$$

Without the extra bit in the intermediate quantitites, the computation would proceed as follows:

$$\begin{array}{lcll} 2^1 \times 0.100 & - & 2^1 \times 0.100 & - \\ 2^0 \times 0.111 & \longrightarrow & 2^1 \times 0.011 & \\ \cline{3-4} & & 2^1 \times 0.001 & = \; 2^{-1} \times 0.100, \end{array}$$

thereby producing a final result in error by 100 %, despite the fact that the "true" result is exactly representable! This extra digit is called a *guard digit*. In the history of computing, many machines, among them several models from Cray, have been designed without a guard digit; the early IBM 360 computers also lacked a guard (hexadecimal) digit, but pressure from the users led to a correction of this feature, with some installed machines being retrofitted in the field. Surprisingly, the requirement of rounding of the exact result in IEEE 754 can be satisfied for floating point addition with just three extra bits; see [59] for a proof and a full discussion of computer arithmetic.

Floating-point operations are:

Commutative (where it makes sense)

$$\begin{aligned} u \oplus v &= v \oplus u \\ u \otimes v &= v \otimes u \end{aligned}$$

Non associative

$$(x \oplus y) \oplus z \neq x \oplus (y \oplus z)$$

Non distributive

$$x \otimes (y \oplus z) \neq (x \otimes y) \oplus (x \otimes z)$$

The breaking of associative and distributive properties is perhaps the most important feature of floating-point arithmetic, and certainly one that gives rise to many surprising results.

One immediate example of this statement is that if sums are not associative, then the order of summation can alter the result. Consider for instance the computations detailed below in single precision IEEE 754, with about 7-8 decimal digits (24 binary digits):

$$
\begin{aligned}
1.0 \oplus 2.0 \oplus 3.0 \oplus 4.0 \oplus 2.68435456 \times 10^8 \oplus -2.68435456 \times 10^8 &= 0.0\\
2.0 \oplus 3.0 \oplus 4.0 \oplus 2.68435456 \times 10^8 \oplus -2.68435456 \times 10^8 \oplus 1.0 &= 1.0\\
3.0 \oplus 1.0 \oplus 2.68435456 \times 10^8 \oplus 4.0 \oplus -2.68435456 \times 10^8 \oplus 2.0 &= 2.0\\
3.0 \oplus 4.0 \oplus 2.68435456 \times 10^8 \oplus -2.68435456 \times 10^8 \oplus 1.0 \oplus 2.0 &= 3.0\\
2.0 \oplus 4.0 \oplus 2.68435456 \times 10^8 \oplus -2.68435456 \times 10^8 \oplus 1.0 \oplus 3.0 &= 4.0\\
1.0 \oplus 2.68435456 \times 10^8 \oplus 4.0 \oplus -2.68435456 \times 10^8 \oplus 3.0 \oplus 2.0 &= 5.0\\
4.0 \oplus 2.68435456 \times 10^8 \oplus -2.68435456 \times 10^8 \oplus 1.0 \oplus 2.0 \oplus 3.0 &= 6.0\\
2.68435456 \times 10^8 \oplus -2.68435456 \times 10^8 \oplus 1.0 \oplus 2.0 \oplus 3.0 \oplus 4.0 &= 10.0\\
-2.68435456 \times 10^8 \oplus 1.0 \oplus 2.0 \oplus 3.0 \oplus 4.0 \oplus 2.68435456 \times 10^8 &= 0.0
\end{aligned}
$$

From the point of view of strict adherence to the standard, all of the above results are equally correct, even though they do not satisfy our intuitive expectations. In the first instance, where the result is zero, the blame is often put on the subtraction

$$-2.68435456 \times 10^8,$$

but the real culprit is the previous sum

$$1.0 \oplus 2.0 \oplus 3.0 \oplus 4.0 \oplus 2.68435456 \times 10^8,$$

because the number of significant digits is such that the contribution of the small numbers is lost; the subsequent subtraction simply exposes this problem. This is an example of an unstable calculation, one in which our expectations run too high with respect to the real capabilities of the underlying number system. A more formal and quantitative analysis of such phenomena is the subject of the next section.

A.3 Backward Error Analysis

Backward error analysis is one of the most useful tools in the evaluation of accuracy of algorithms; this despite the fact that its basic idea is extremely simple. To see it, consider again the standard model of floating point computations (A.3):

$$\mathrm{fl}(x \textcircled{\text{op}} y) = (x \textcircled{\text{op}} y)(1+\delta).$$

If $\textcircled{\text{op}}$ is distributive, a seemingly trivial rewriting gives us the basis for a very important insight:

$$\mathrm{fl}(x \textcircled{\text{op}} y) = (x \textcircled{\text{op}} y)(1+\delta) = (x \cdot (1+\delta)) \textcircled{\text{op}} (y \cdot (1+\delta)). \qquad \text{(A.4)}$$

In plain english, the rounded result of the floating point operation is equal to the *exact* result of the operation applied to *perturbed* data; note that the perturbed data might not be exactly representable. This is an extremely important shift in the point of view regarding the error, for the following reasons:

- Interpreting the error as the effect of perturbation in the input allows us to draw on the rich mathematical theory of perturbations;
- For realistic problems, perturbations are already present in the input data anyway, both because of measurement errors and also because of the approximation inherent in storing the numbers in finite precision.

Let us then formally define the backward error:

Definition A.3.1 *Given the function $y = f(x)$, and its computed approximation $\hat{y} = \hat{f}(x)$, the* backward error *is the perturbation δx such that*

$$\hat{y} = f(x + \delta x).$$

The basic tenet of backward error analysis can be thus stated as:

Definition A.3.2 *An algorithm is stable in the sense of backward error analysis (or for short* backward stable*) if the perturbation δx introduced in the data is not too large compared to the uncertainty with which the original data x was known.*

Now the obvious question arises: is it possible to expect a small error in the *result* given a small backward error?

The link between the *forward* error (i.e. the error in the function result) and the backward error is provided by the *condition number*. Let us consider the computation of $f(x)$ where f is a real differentiable function; expanding in Taylor series we have

$$\hat{f}(x) = f(x + \delta x) = f(x) + f'(x)\delta x + O(\delta x^2);$$

this is equivalent to stating that

$$|f(x + \delta x) - f(x)| \approx |f'(x)||\delta x|.$$

Therefore: if the error in the computed value can be interpreted as the effect of a perturbation in the input, then the perturbation is amplified by a factor $|f'(x)|$ that depends only on the *true* function f, not the approximate $\hat{f}$.

Moving to relative errors, simple algebraic manipulations give

$$\frac{|f(x + \delta x) - f(x)|}{|f(x)|} \approx \frac{|\delta x|}{|x|} \frac{|f'(x)| \cdot |x|}{|f(x)|} \tag{A.5}$$

where the quantity $|f'(x)| \cdot |x|/|f(x)|$ is called the *relative condition number*. A problem is well conditioned if its condition number is small, ill-conditioned if it is large, and ill-posed if its condition number is infinite[†]. This formulation achieves two very important goals:

1. It separates neatly the effect of the algorithm (which is felt mainly through δx) from the features of the "true" function $f(x)$;
2. It allows to draw from the powerful analytical tools of the theory of perturbations.

[†] Ill-posed problems also include those for which the solution is not differentiable.

The first point can hardly be overemphasized. Given that perturbations in the data are completely unavoidable, in most situations what we can reasonably ask for is that the method employed for the solution should introduce an uncertainty not significantly larger than that which was already present. Whether this is sufficient to get a good solution is a completely different question; in particular, it is extremely ill-advised to expect a reliable answer to a severely ill-conditioned problem.

To give a concrete example, let us analyze the simple task of computing the sum of n floating-point numbers:

$$S = \sum_{i=1}^{n} x_i. \tag{A.6}$$

If we compute the sum in the natural order we have for $n = 3$

$$S = ((x_1 + x_2)(1 + \delta_1) + x_3)(1 + \delta_2)$$

and generalizing

$$S \sum_{i=1}^{n} x_i \prod_{k=1}^{\max(1,i-1)} (1 + \delta_k).$$

The bound $|\delta| \leq u$ carries over to the following result

$$\prod_{k=1}^{n}(1 + \delta_k) = (1 + \theta)$$

with

$$|\theta| \leq \gamma_n = \frac{nu}{1 - nu}$$

under the hypothesis that $nu < 1$.

In the IEEE single precision arithmetic we have $u = 6.0 \times 10^{-8}$; this means that a sum involving 10^7 terms can be expected to have very significant perturbations. Such sums are necessary to compute scalar products in the context of iterative solvers; it is therefore clear that very large systems, of the order of hundreds of millions of unknowns, need careful attention, and usually double precision arithmetic, to be solved reliably.

A.4 Vector and Matrix Norms

We have seen how we can build an understanding of the effects of errors on computations of function values; quantitative statements about simple *scalar* functions can be formulated by relying on the concept of the absolute value of a number.

Since most of this book will deal with linear systems, matrices and vectors, it is natural to ask how can we carry over to this new context of linear spaces the concepts of error and conditioning we have just seen; to achieve this goal we need to generalize the concepts of absolute value and distance into the concepts of *norm* and *metric* on a linear space.

Definition A.4.1 *Let $\mathcal{V}$ be a linear space on the real or complex numbers, e.g. $\mathbb{R}^n$. A* metric *(or distance) is a function $d(\cdot,\cdot) : (\mathcal{V} \times \mathcal{V}) \rightarrow \mathbb{R}$ that* *satisfies all of the following properties:*

- $d(x, y) \geq 0$, *with equality if and only if* $x = y$ *(positive definiteness);*
- $d(x, y) = d(y, x)$ *(simmetry);*
- $d(x, y) \leq d(x, z) + d(z, y)$ *(triangle inequality).*

Definition A.4.2 *Let* $\mathcal{V}$ *be a linear space on the real or complex numbers, e.g.* $\mathbb{R}^n$*. A* norm *on* $\mathcal{V}$ *is a function* $\|\cdot\| : \mathcal{V} \to \mathbb{R}$ *that satisfies all of the following properties:*

- $\|x\| \geq 0$, *and* $\|x\| = 0$ *if and only if* $x = 0$ *(positive definiteness);*
- $\|\alpha x\| = |\alpha|\|x\|$ *(homogeneity);*
- $\|x + y\| \leq \|x\| + \|y\|$ *(triangle inequality).*

The most familiar example is the Euclidean norm

$$\|x\|_2 = \left(\sum_i |x_i|^2\right)^{1/2},$$

which is readily generalized to the p-norm

$$\|x\|_p = \left(\sum_i |x_i|^p\right)^{1/p}.$$

In the limit we obtain the infinity-norm $\|x\|_\infty = \max_i(|x_i|)$. Note that any norm induces a metric function via the formula $d(x, y) = \|x - y\|$, whereas a metric does not necessarily correspond to a norm because it lacks the homogeneity property. Thus we will always in the sequel make use of norms and norm-induced distances.

Definition A.4.3 *Let* $\mathcal{V}$ *be a linear space on the real or complex numbers. An* inner product *is a function* $\langle\cdot,\cdot\rangle : (\mathcal{V} \times \mathcal{V}) \to \mathbb{R}$ *that satisfies all of the following properties:*

1. $\langle x, y\rangle = \langle y, x\rangle$ *in* $\mathbb{R}$ *or* $\overline{\langle y, x\rangle}$ *in* $\mathbb{C}$ *(simmetry);*
2. $\langle x, y + z\rangle = \langle x, y\rangle + \langle x, z\rangle$*;*
3. $\langle \alpha x, y\rangle = \alpha\langle x, y\rangle$ *for any real or complex* α*;*
4. $\langle x, x\rangle \geq 0$, *with equality if and only if* $x = 0$.

The usual scalar product among vectors in $\mathbb{C}^n$

$$\langle x, y\rangle = y^* x = \sum_i \overline{y_i} x_i$$

is an inner product; two vectors are said to be *orthogonal* if their inner product is zero. Any inner product satisfies the Cauchy-Schwartz inequality

$$|\langle x, y\rangle| \leq \sqrt{\langle x, x\rangle \cdot \langle y, y\rangle}.$$

Any inner product induces a norm:

$$\|x\| = \sqrt{\langle x, x\rangle},$$

but not all norms are induced by an inner product, e.g. the infinity norm does not correspond to any inner product.

Definition A.4.4 *A real symmetric (complex Hermitian) matrix is* positive definite *if $x^T Ax > 0$ ($x^* Ax > 0$) for all $x \neq 0$.*

A positive definite matrix induces an inner product and a norm via the formulae

$$\begin{aligned} \langle x, y \rangle_A &= y^* Ax \\ \|x\|_A &= \sqrt{x^* Ax} \end{aligned}$$

The A-norm is sometimes called the *energy* norm.

Since the space of all $m \times n$ matrices is itself a linear space, we can define a matrix norm on it just like the norms we have already seen in Definition A.4.2. However it is useful to define a particular class of norms that takes into account the fact that a matrix is a representation of an operator linking two vectors spaces.

Definition A.4.5 *Let A be an $m \times n$ matrix, $\|\cdot\|_{\hat{m}}$ a norm on $\mathbb{R}^m$ and $\|\cdot\|_{\hat{n}}$ a norm on $\mathbb{R}^n$. Then*

$$\|A\|_{\hat{m}\hat{n}} = \max_{x \neq 0} \frac{\|Ax\|_{\hat{m}}}{\|x\|_{\hat{n}}}$$

is called an operator norm*, or* induced norm*, or* subordinate matrix norm*.*

Any operator norm is also a matrix norm. Operator norms satisfy the following additional properties:

1. $\|Ax\| \leq \|A\| \|x\|$
2. $\|AB\| \leq \|A\| \|B\|$

For the norm induced by the Euclidean 2-norm we have $\|QAZ\| = \|A\|$ for all orthogonal or unitary Q and Z. Other useful facts about norms are:

- $\|A\|_\infty = \max_{x \neq 0} \frac{\|Ax\|_\infty}{\|x\|_\infty} = \max_i (\sum_j |a_{ij}|)$;
- $\|A\|_1 = \max_{x \neq 0} \frac{\|Ax\|_1}{\|x\|_1} = \max_j (\sum_i |a_{ij}|) = \|A^T\|_\infty$;
- $\|A\|_2 = \max_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2} = \sqrt{\lambda_{\max}(A^* A)}$ where $\lambda_{\max}$ is the largest eigenvalue;
- $\|A\|_2 = \|A^T\|_2$.

A.5 Perturbation Theory for Linear Systems

Considering a linear system $Ax = b$, its computed solution $\hat{x}$ will in general satisfy a backward error result of the type

$$(A + \delta A)(x + \delta x) = (b + \delta b).$$

Subtracting $Ax = b$ and assuming that A is nonsingular, i.e. A^{-1} exists, we obtain

$$\delta x = A^{-1}(-\delta A \hat{x} + \delta b).$$

Applying any operator norm we get

$$\|\delta x\| \leq \|A^{-1}\|(\|\delta A\|\hat{x}\| + \|\delta b\|),$$

and by simple algebraic manipulations

$$\frac{\|\delta x\|}{\|\hat{x}\|} \leq \|A^{-1}\|\|A\|(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|A\|\|\hat{x}\|}). \tag{A.7}$$

The quantity $\kappa(A) = \|A^{-1}\|\|A\|$ is the *condition number* associated with matrix A, because equation (A.7) says that it links the relative change in the answer $\frac{\|\delta x\|}{\|\hat{x}\|}$ to the relative change in the data $\frac{\|\delta A\|}{\|A\|}$.

The condition number admits another important interpretation: it can be seen as the (reciprocal of the) distance from singularity, in the sense of the following theorem:

Theorem A.5.1 *Let A be a nonsingular matrix, and let δA be any perturbation δA such that $A + \delta A$ is singular. Then*

$$\min\left\{\frac{\|\delta A\|_2}{\|A\|_2} : A + \delta A singular\right\} = \frac{1}{\|A^{-1}\|\|A\|} = \frac{1}{\kappa(A)}. \tag{A.8}$$

Proof. Let us first prove that $\frac{\|\delta A\|}{\|A\|} \geq \frac{1}{\kappa(A)}$. The definition of singularity implies that there exists a vector x such that $(A + \delta A)x = 0$; then, with simple algebraic manipulations, we obtain

$$\begin{array}{rcl} \delta A x & = & -Ax \\ A^{-1}\delta A x & = & -x \\ \|A^{-1}\delta A x\| & = & \|x\| \\ \|A^{-1}\|\|\delta A\|\|x\| & \geq & \|x\| \\ \|\delta A\| & \geq & \dfrac{1}{\|A^{-1}\|} \\ \dfrac{\|\delta A\|}{\|A\|} & \geq & \dfrac{1}{\|A\|\|A^{-1}\|} = \dfrac{1}{\kappa(A)} \end{array}$$

This part of the proof is valid for any operator norm $\|\cdot\|$; if we now restrict ourselves to the 2-norm, we can prove that the minimum can be attained. By definition $\|A^{-1}\|_2 = \max_{x\neq 0} \frac{\|A^{-1}x\|_2}{\|x\|_2}$; since norms are homogeneous, there exists an x with $\|x\|_2 = 1$ where this maximum is attained. Building a unit vector $y = \frac{A^{-1}x}{\|A^{-1}x\|_2} = \frac{A^{-1}x}{\|A^{-1}\|_2}$, we can construct $\delta A = y = \frac{-xy^T}{\|A^{-1}\|_2}$. Then

$$\|\delta A\|_2 = \max_{z\neq 0} \frac{\|xy^Tz\|_2}{\|A^{-1}\|_2\|z\|_2} = \max_{z\neq 0} \frac{|y^Tz|\|x\|_2}{\|z\|_2} \frac{\|x\|_2}{\|A^{-1}\|_2} = \frac{1}{\|A^{-1}\|_2}$$

where the maximum is attained when z is a multiple of y. Finally,

$$(A + \delta A)y = Ay - \frac{xy^Ty}{\|A^{-1}\|_2} = \frac{x}{\|A^{-1}\|_2} - \frac{xy^Ty}{\|A^{-1}\|_2} = 0,$$

hence $A + \delta A$ is singular. $\square$

A.6 Notes and References

This chapter owes much of its content to the excellent book by Higham [58]; the linear algebra book [60] is also heartily recommended. The IEEE 754 standard official publication is [61]; a readable and comprehensive treatment of its arithmetic properties is found in [62], reprinted in [59] and [63]. A thorough discussion of many arithmetic algorithms can be found in [64].

B Spack

Spack is a flexible package manager designed for high-performance computing (HPC) environments. It simplifies the process of building, installing, and managing software dependencies by allowing users to specify configurations in a highly customizable manner. Spack supports multiple versions, configurations, and compilers. This kind of tools has become an essential tool for researchers and developers working on complex software stacks. Its versatility and ease of use have made it a popular choice in the scientific computing community.

In the following we make a very brief introduction to Spack and its basic usage. The Spack documentation is extensive and well written, so we recommend to refer to it for more details on the website spack.readthedocs.io.

B.1 Installation

To install Spack, you can clone the repository from GitHub. Open a terminal and run the following command:

```
git clone git@github.com:spack/spack.git
```

This will create a directory named `spack` in your current working directory. You can then add Spack to your shell's environment by sourcing the `setup-env.sh` script:

```
source ./spack/share/spack/setup-env.sh
```

This command sets up the necessary environment variables and functions for Spack to work properly. You can also add the above line to your shell's configuration file (e.g., `.bashrc`) to make the changes permanent; sometimes you may need to hmi ave more than one installation of Spack on the same machine, in this case, adding the default loading to he `.bashrc` file may not be the best option. In this case, you can create a script that loads the Spack environment and add it to your `.bashrc` file. For example, you can create different aliases for different Spack installations:

```
alias spack1='source /path/spack1/share/spack/setup-env.sh'
alias spack2='source /path/spack2/share/spack/setup-env.sh'
alias spack3='source /path/spack3/share/spack/setup-env.sh'
```

Then, you can load the desired Spack installation by running the corresponding alias command in your terminal.

To verify that Spack is installed correctly, you can run the following command:

```
spack --version
```

This should display the version of Spack you have installed. You can also check the available commands by running:

```
spack help
```

This will show you a list of available Spack commands and their descriptions.

B.2 Basic Usage

Spack provides a simple command-line interface for managing software packages. The basic workflow involves searching for packages, installing them, and managing their dependencies.

In all cases the first thing we have to do is making a compiler available to Spack, this should be a compiler you have installed on your system. You can check the available compilers by running:

```
spack compilers
```

The typical aspect of the output should be something like:

```
==> Available compilers
-- gcc ubuntu24.04-x86_64 ------------------------------
gcc@13.3.0
```

In this case, we have a single compiler available, `gcc@13.3.0`. To add a new compiler, you can use the `spack compiler add` command:

```
spack compiler add /path/to/your/compiler
```

This command will add the specified compiler to Spack's list of available compilers. You can also let spack detect the compiler automatically by running:

```
spack compiler find
```

This command will search for compilers installed on your system and add them to Spack's list of available compilers. In case you plan having more than one version of Spack installed it is recommended to set the *scope* of the compiler to be only available to the current Spack installation. You can do this by running:

```
spack compiler find --scope site
```

As an example, we can think of installing the `gcc` compiler at version `14.2.0`. First we can run the following command to check for the **installation variants**[1] we can select:

1: To discover packages and variants you can visit the website packages.spack.io.

```
spack info gcc@14.2.0
```

This will show you the available variants, dependencies, and other information about the package, e.g., the available variants are:

```
binutils [false]                   false, true
    Build via binutils
bootstrap [true]                   false, true
    Enable 3-stage bootstrap
build_system [autotools]           autotools
    Build systems supported by the package
build_type [RelWithDebInfo]        Debug, MinSizeRel,
↪   RelWithDebInfo, Release
    CMake-like build type. Debug: -O0 -g; Release: -O3;
     ↪  RelWithDebInfo: -O2 -g; MinSizeRel: -Os
graphite [false]                   false, true
    Enable Graphite loop optimizations (requires ISL)
languages [c,c++,fortran]          ada, brig, c, c++, d,
↪   fortran, go, java, jit, lto, obj-c++, objc
    Compilers and runtime libraries to build
nvptx [false]                      false, true
    Target nvptx offloading to NVIDIA GPUs
piclibs [false]                    false, true
    Build PIC versions of libgfortran.a and libstdc++.a
strip [false]                      false, true
    Strip executables to reduce installation size
```

In our case it would be a good idea to activate the `nvptx` support to compile for NVIDIA GPUs, and select a `build_type=Release` to enable all the optimizations, so we can run the following command to install `gcc`[2]:

```
spack install gcc@14.2.0 +nvptx build_type=Release
↪   ^cuda@12.8.0
```

2: The + sign indicates that the variant is enabled, while the ~ sign indicates that the variant is disabled. The ^ sign indicates that the package is a dependency of another package. In this case we are installing the `gcc` package with the `nvptx` variant enabled which in turns depends on Cuda, to ensure compatibility, we select `^cuda@12.8.0`.

After the completion of the install procedure—which will require some time depending on your machine specifics—you can add it to the set of compilers available to spack by doing:

```
spack compiler add $(spack find --paths gcc@14.2.0)
↪  --scope=site
```

which will return

```
==> Added 1 new compiler to
↪   /home/user/.spack/linux/compilers.yaml
gcc@14.2.0
==> Compilers are defined in the following files:
/home/user/.spack/linux/compilers.yaml
```

You can now install any other package using the `gcc@14.2.0` compiler.

For example, you can install the `openblas` and `openmpi`[3] libraries by running:

```
spack install openblas%gcc@14.2.0
spack install openmpi%gcc@14.2.0 +cuda cuda_arch=89
↪  +legacylaunchers
```

3: The `openmpi` package is a popular implementation of the MPI standard. Since we want to use it conjunction with CUDA, we need to enable the `+cuda` variant and select the CUDA architecture/computing capabilities, in the case in the example we use the compute capabilities 89 which are compatible with the GPU we are using: to look for the matching compute capabilities you can visit the NVIDIA website CUDA GPUs.

We can now load the installed modules by running:

```
spack load gcc@14.2.0
spack load openblas%gcc@14.2.0
spack load openmpi%gcc@14.2.0
```

After loading the modules, you can check the GCC version by running:

```
gcc --version
```

This should display the version of GCC you have installed and loaded.

```
gcc (Spack GCC) 14.2.0
Copyright (C) 2024 Free Software Foundation, Inc.
This is free software; see the source for copying
↪  conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A
↪  PARTICULAR PURPOSE.
```

B.2.1 Environment Modules

To better manage the installed packages and their dependencies, it is better to use the Environment Modules system. This system allows you to load and unload different software packages and their dependencies easily.

To use Environment Modules with Spack the simplest configuration is to have `lmod` installed on your system. You can check if it is installed by running:

```
module --version
```

If it is available it should answer something like:

```
Modules based on Lua: Version 8.6.19  2022-04-16 13:46
↪ -05:00
by Robert McLay mclay@tacc.utexas.edu
```

If it is not installed, you can install it using the package manager of your system. For example, on Ubuntu, you can run:

```
sudo apt-get install lmod
```

Once you have `lmod` installed, you can configure Spack to use it by running:

```
spack lmod refresh --delete-tree
```

This command will generate the necessary module files for all installed packages. You then have to make aware the system about the modules, e.g., by adding the following line to your `.bashrc` file[4]:

4: This is the case for a Spack installed under an Ubuntu 22.04 system, if you are using a different system you can check the path by running: `ls /path/to/spack/share/spack/lmod/`.

```
export MODULEPATH=${MODULEPATH}:/path/to/spack/share/
↪  spack/lmod/linux-ubuntu22.04-x86_64/Core
```

You can then load the modules by running, e.g.,

```
module load gcc/14.2.0
```

The way in which the modules are written by the `spack lmod refresh` command is configured in the `modules.yaml` file located in the `spack/etc/spack` directory. An example of configuration is the following:

```yaml
modules:
  prefix_inspections:
    ./bin:
      - PATH
    ./include:
      - CPATH
    ./inc:
      - CPATH
    ./lib:
      - LIBRARY_PATH
      - LD_LIBRARY_PATH
    ./lib64:
      - LIBRARY_PATH
      - LD_LIBRARY_PATH
  default:
    enable:
    - lmod
    lmod:
      hash_length: 0
      core_compilers:
        - 'gcc@13.3.0'
        - 'llvm'
        - 'gcc@14.2.0'
      hide_implicits: true
      hierarchy:
        - 'mpi'
        - 'compiler'
      all:
        conflict:
          - '{name}'
        autoload: direct
```

You can read more about the configuration options in the Spack documentation.

Bibliography

Here are the references in citation order.

[1] Randall J. LeVeque. *Finite difference methods for ordinary and partial differential equations*. Steady-state and time-dependent problems. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2007, pp. xvi+341 (cited on page 2).

[2] Susanne C. Brenner and L. Ridgway Scott. *The mathematical theory of finite element methods*. Third. Vol. 15. Texts in Applied Mathematics. Springer, New York, 2008, pp. xviii+397 (cited on page 2).

[3] Randall J. LeVeque. *Finite volume methods for hyperbolic problems*. Cambridge Texts in Applied Mathematics. Cambridge University Press, Cambridge, 2002, pp. xx+558 (cited on page 2).

[4] John G. Kemeny and J. Laurie Snell. *Finite Markov chains*. The University Series in Undergraduate Mathematics. D. Van Nostrand Co., Inc., Princeton, N.J.-Toronto-London-New York, 1960, pp. viii+210 (cited on page 3).

[5] Gene H. Golub and Charles F. Van Loan. *Matrix computations*. Fourth. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, MD, 2013, pp. xiv+756 (cited on page 5).

[6] James W. Demmel. *Applied Numerical Linear Algebra*. Philadelphia, PA: SIAM, 1997 (cited on page 5).

[7] Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra*. SIAM, 1997 (cited on page 5).

[8] Sheldon Jay Axler. *Linear Algebra Done Right*. Undergraduate Texts in Mathematics. New York: Springer, 1997 (cited on page 5).

[9] Roger A. Horn and Charles R. Johnson. *Matrix analysis*. Second. Cambridge University Press, Cambridge, 2013, pp. xviii+643 (cited on page 5).

[10] Roger A. Horn and Charles R. Johnson. *Topics in matrix analysis*. Corrected reprint of the 1991 original. Cambridge University Press, Cambridge, 1994, pp. viii+607 (cited on page 5).

[11] Jack J. Dongarra et al. *LINPACK User's Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1979 (cited on pages 7, 111).

[12] Michael Metcalf et al. *Modern Fortran Explained: Incorporating Fortran 2023*. 6th ed. Numerical Mathematics and Scientific Computation. Oxford, UK: Oxford University Press, 2024 (cited on page 10).

[13] Alessandro Fanfarillo et al. 'OpenCoarrays: Open-source Transport Layers Supporting Coarray Fortran Compilers'. In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. PGAS '14. Eugene, OR, USA: Association for Computing Machinery, 2014. DOI: 10.1145/2676870.2676876 (cited on page 12).

[14] John Backus. 'The History of FORTRAN I, II and III'. In: *IEEE Ann. Hist. Comput.* 1.1 (July 1979), pp. 21–37. DOI: 10.1109/MAHC.1979.10013 (cited on page 25).

[15] M.J. Flynn. 'Very high-speed computing systems'. In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909. DOI: 10.1109/PROC.1966.5273 (cited on page 29).

[16] Michael 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 (cited on page 29).

[17] Kai Hwang and Zhiwei Xu. *Scalable Parallel Computing*. Mc Graw-Hill, 1998 (cited on pages 34, 35).

[18] Samuel Williams, Andrew Waterman, and David Patterson. 'Roofline: an insightful visual performance model for multicore architectures'. In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. DOI: 10.1145/1498765.1498785 (cited on page 38).

[19] John 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 (cited on page 39).

[20] John 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 (cited on page 39).

[21] David R. Butenhof. *Programming with POSIX threads*. USA: Addison-Wesley Longman Publishing Co., Inc., 1997 (cited on page 44).

[22] OpenMP Architecture Review Board. *OpenMP Application Programming Interface Specification 6.0*. 2024 (cited on pages 45, 46).

[23] OpenMP Architecture Review Board. *OpenMP Application Programming Interface Specification 5.2*. Ed. by Bronis de Supinski and Michael Klemm. 2021 (cited on page 45).

[24] Yun (Helen) He Timothy G. Mattson and Alice E. Koniges. *The OpenMP Common Core*. MIT Press, Cambridge, MA, 2019, p. 320 (cited on page 45).

[25] Gabriele Jost Barbara Chapman and Ruud van der Pas. *Using OpenMP*. MIT Press, Cambridge, MA, 2007, p. 384 (cited on page 45).

[26] Pasqua D'Ambra, Fabio Durastante, and Salvatore Filippone. 'Parallel Sparse Computation Toolkit'. In: *Software Impacts* 15 (2023), p. 100463. DOI: https://doi.org/10.1016/j.simpa.2022.100463 (cited on page 50).

[27] M. Bernaschi and G. Iannello. 'Collective Communication Operations: Experimental Results vs. Theory'. In: *Concurrency: Practice and Experience* 10.5 (1998), pp. 359–386 (cited on page 52).

[28] R. Thakur, R. Rabenseifner, and William Gropp. 'Optimization of Collective Communication Operations in MPICH'. In: *International Journal of High Performance Computing Applications* 19.1 (2005), pp. 49–66 (cited on page 52).

[29] Bo Kågström, Per Ling, and Charles van Loan. 'GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark'. In: *ACM Trans. Math. Softw.* 24.3 (Sept. 1998), pp. 268–302. DOI: 10.1145/292395.292412 (cited on page 85).

[30] T. Davis. 'Wilkinson's sparse matrix definition'. In: *NA Digest* 07.12 (Mar. 2007), pp. 379–401 (cited on page 87).

[31] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Second. Society for Industrial and Applied Mathematics, 2003 (cited on pages 89, 93, 94).

[32] J. A. Meijerink and H. A. van der Vorst. 'An Iterative Solution Method for Linear Systems of Which the Coefficient Matrix is a Symmetric M-Matrix'. In: *Math. Comp* 31 (1977), pp. 148–162 (cited on page 93).

[33] Barry F. Smith, Petter E. Bjørstad, and William D. Gropp. *Domain decomposition. Parallel multilevel methods for elliptic partial differential equations*. Cambridge, UK: Cambridge University Press, 1996, pp. xii+224 (cited on page 94).

[34] Klaus Stüben. *Algebraic Multigrid (AMG): An Introduction with Applications*. Tech. rep. 70. Schloss Birlinghoven, Sankt Augustin, Germany: GMD, 1999 (cited on page 94).

[35] Robert D. Falgout. 'An Introduction to Algebraic Multigrid'. In: *Computing in Science and Engineering* 8.3 (2006), pp. 24–33 (cited on page 94).

[36] Pasqua D'Ambra, Fabio Durastante, and Salvatore Filippone. 'AMG Preconditioners for Linear Solvers towards Extreme Scale'. In: *SIAM Journal on Scientific Computing* 43.5 (2021), S679–S703. DOI: 10.1137/20M134914X (cited on page 94).

[37] Daniele Bertaccini and Salvatore Filippone. 'Sparse approximate inverse preconditioners on high performance GPU platforms'. In: *Computers & Mathematics with Applications* 71.3 (2016), pp. 693–711. DOI: https://doi.org/10.1016/j.camwa.2015.12.008 (cited on page 94).

[38] Pasqua D'Ambra, Salvatore Filippone, and Panayot S. Vassilevski. 'BootCMatch: A Software Package for Bootstrap AMG Based on Graph Weighted Matching'. In: *ACM Trans. Math. Softw.* 44.4 (June 2018). DOI: 10.1145/3190647 (cited on page 94).

[39] I.S. Duff et al. 'Level 3 Basic Linear Algebra Subprograms for Sparse Matrices: a User Level Interface'. In: 23.3 (1997), pp. 379–401 (cited on page 95).

[40] Salvatore Filippone et al. 'Sparse matrix-vector multiplication on GPGPUs'. In: *ACM Trans. Math. Software* 43.4 (2017), Art. 30, 49 (cited on pages 96, 98).

[41] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Reading, MA: Addison–Wesley, 1983 (cited on page 97).

[42] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995 (cited on page 97).

[43] D. W. I. Rouson, J. Xia, and X. Xu. *Scientific Software Design: The Object-Oriented Way*. Cambridge University Press, 2011 (cited on page 97).

[44] S. Filippone and A. Buttari. 'Object-Oriented Techniques for Sparse Matrix Computations in Fortran 2003'. In: 38.4 (2012), 23:1–23:20 (cited on page 98).

[45] Valeria Cardellini, Salvatore Filippone, and Damian Rouson. 'Design patterns for sparse-matrix computations on hybrid CPU/GPU platforms'. In: 22.1 (2014), pp. 1–19 (cited on page 98).

[46] Jack J. Dongarra et al. 'The Design and Implementation of the PBLAS (Parallel Basic Linear Algebra Subprograms)'. In: *Concurrency: Practice and Experience* 7.1 (1995), pp. 23–42. DOI: 10.1002/cpe.4330070104 (cited on page 109).

[47] Jack J. Dongarra, R. Clint Whaley, and Antoine Petitet. 'Basic Linear Algebra Communication Subprograms (BLACS)'. In: *Concurrency: Practice and Experience* 9.11 (1997), pp. 875–897. DOI: 10.1002/(SICI)1096-9128(199709)9:11<875::AID-CPE324>3.0.CO;2-9 (cited on page 109).

[48] Brian T. Smith et al. *Matrix Eigensystem Routines – EISPACK Guide*. Vol. 6. Lecture Notes in Computer Science. Springer-Verlag, 1976 (cited on page 111).

[49] E. Anderson et al. *LAPACK Users' Guide*. 3rd. Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM), 1999 (cited on page 111).

[50] L. S. Blackford et al. *ScaLAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM), 1997 (cited on page 111).

[51] E. Agullo et al. *PLASMA Users' Guide*. Tech. rep. Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA). University of Tennessee, 2009 (cited on page 111).

[52] S. Tomov, J. Dongarra, A. Haidar, et al. 'MAGMA: Dense Linear Algebra for Heterogeneous Architectures'. In: *International Journal of High Performance Computing Applications* 24.3 (2010), pp. 277–288. DOI: 10.1177/1094342010369113 (cited on page 111).

[53] M. A. Gates et al. 'SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library'. In: *ACM Transactions on Mathematical Software* 47.2 (2021), pp. 1–29. DOI: 10.1145/3437806 (cited on page 111).

[54] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. First Edition. W. H. Freeman, 1979 (cited on page 115).

[55] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Reading, Mass.: Addison-Wesley, 1974 (cited on page 115).

[56] J.H. Wilkinson. *The Algebraic Eigenvalue Problem*. Clarendon Press, Oxford, 1965 (cited on page 125).

[57] Donald Ervin Knuth. *TEX: The Program*. USA: Addison-Wesley Longman Publishing Co., Inc., 1986 (cited on page 126).

[58] Nicholas J. Higham. *Accuracy and stability of numerical algorithms*. Second. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2002, pp. xxx+680 (cited on pages 129, 137).

[59] David Goldberg. 'Appendix A: Computer Arithmetic'. In: *Computer Architecture, a Quantitative Approach*. Ed. by John L. Hennessy and David A. Patterson. Vol. 2nd ed. Morgan-Kaufmann, 1996 (cited on pages 130, 137).

[60] James W. Demmel. *Applied numerical linear algebra*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1997, pp. xii+419 (cited on page 137).

[61] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. Revised 1990. A preliminary draft was published in the January 1980 issue of IEEE Computer, together with several companion articles [65–71]. The final version was republished in [72, 73]. See also [74]. Also standardized as *IEC 60559 (1989-01) Binary floating-point arithmetic for microprocessor systems*. Aug. 1985, p. 20 (cited on pages 137, 148).

[62] David Goldberg. 'What every computer scientist should know about floating-point arithmetic'. In: *ACM Computing Surveys* 67.1 (1991), pp. 71–92 (cited on page 137).

[63] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Fourth. San Francisco, CA: Morgan Kaufmann Publishers, Inc., 2007 (cited on page 137).

[64] Donald E. Knuth. *The art of computer programming. Vol. 2*. Second. Addison-Wesley Series in Computer Science and Information Processing. Seminumerical algorithms. Addison-Wesley Publishing Co., Reading, MA, 1981, pp. xiii+688 (cited on page 137).

[65] William J. Cody, Jr. 'Analysis of Proposals for the Floating-Point Standard'. In: 14.3 (Mar. 1981), pp. 63–68. DOI: `https://doi.org/10.1109/C-M.1981.220379` (cited on page 148).

[66] Jerome T. Coonen. 'Underflow and the Denormalized Numbers'. In: 14 (1981), pp. 75–87 (cited on page 148).

[67] Jerome T. Coonen. 'An Implementation Guide to a Proposed Standard for Floating Point Arithmetic'. In: 13.1 (Jan. 1980). See errata in [68]., pp. 68–79 (cited on page 148).

[68] Jerome T. Coonen. 'Errata: An Implementation Guide to a Proposed Standard for Floating Point Arithmetic'. In: 14.3 (Mar. 1981). See [67]., 62–?? (Cited on page 148).

[69] David Hough. 'Applications of the Proposed IEEE-754 Standard for Floating Point Arithmetic'. In: 14.3 (Mar. 1981), pp. 70–74 (cited on page 148).

[70] David Stevenson. 'A Proposed Standard for Binary Floating-Point Arithmetic'. In: 14.3 (Mar. 1981). See [61, 75]., pp. 51–62 (cited on page 148).

[71] David Stevenson. *A proposed standard for binary floating-point arithmetic: draft 8.0 of IEEE Task P754*. See [61, 75]. 1981, p. 36 (cited on page 148).

[72] IEEE. 'IEEE Standard for Binary Floating-Point Arithmetic'. In: *ACM SIGPLAN Notices* 22.2 (Feb. 1985), pp. 9–25 (cited on page 148).

[73] IEEE Computer Society Standards Committee. Working group of the Microprocessor Standards Subcommittee and American National Standards Institute. *IEEE standard for binary floating-point arithmetic*. ANSI/IEEE Std 754-1985. 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA: IEEE Computer Society Press, 1985, p. 18 (cited on page 148).

[74] Shlomo Waser and Michael J. Flynn. *Introduction to Arithmetic for Digital Systems Designers*. 1982, pp. xvii + 308 (cited on page 148).

[75] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. A preliminary draft was published in the January 1980 issue of IEEE Computer, together with several companion articles [65–71]. Available from the IEEE Service Center, Piscataway, NJ, USA. IEEE, New York. Aug. 1985 (cited on page 148).