

High Performance Linear Algebra

Lecture 2: Performance Modeling and the Roofline Model

Ph.D. program in High Performance Scientific Computing

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

November 19, 2026 — 14.00:16.00



Dipartimento
di Matematica
Università di Pisa



Summary of previous lecture

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

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



Table of Contents

2 Continuous Integration and Deployment (CI/CD)

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



What is CI/CD?

2 Continuous Integration and Deployment (CI/CD)

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

Continuous Integration (CI):

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

Continuous Deployment (CD):

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



Benefits of CI/CD

Key benefits:

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

Popular CI/CD tools:

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



An example of CI/CD with GitHub Actions

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

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

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



Makefile basics

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

```
target: dependencies
    command
```

Example:

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

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



Creating a workflow with Forgejo Actions

Forgejo Actions automates build, test and delivery workflows for a repository on Codeberg.

- **Workflows:** YAML files that define triggers and jobs.
- **Jobs:** groups of steps executed by an eligible runner.
- **Steps:** shell commands or reusable actions.
- **Runners:** machines that execute those jobs.



Store workflow files in `.forgejo/workflows/`.

Codeberg already runs Forgejo, what we need to provide is **the runner** and the **build environment**.



The components of our pipeline

Codeberg	Hosts the repository, schedules jobs and displays results.
Linux host	Runs Forgejo Runner as a background service.
Docker job	Provides Git, Node.js, GNU Fortran and Make.
Repository	Supplies the source code, Makefile and workflow.

The runner connects to Codeberg and requests work.

It then creates a **job container** and reports *logs* and the *result*.

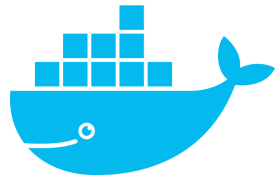
A public IP address or incoming webhook port is unnecessary.



Preparing the host machine

This example assumes a Linux host with systemd.

1. Install and start Docker.
2. Install a current, verified Forgejo Runner binary.
3. Create a dedicated `forgejo-runner` service user.
4. Give that user access to Docker.
5. Create a private configuration file and working directory.



Example working directory: `/var/lib/forgejo-runner`.

In the following slides we will go through the commands needed to setup the various steps.



Installing Docker Engine on Ubuntu

Host: fresh Ubuntu 24.04 (Noble), with `sudo` access.

Step 1 of 5: Install prerequisites

Install the tools needed to add Docker's repository.

```
sudo apt-get update
sudo apt-get install -y ca-certificates curl
```

Docker documentation: installation on Different Linux Distro



Installing Docker Engine on Ubuntu

Host: fresh Ubuntu 24.04 (Noble), with `sudo` access.

Step 2 of 5: Add Docker's signing key

APT uses this key to verify Docker's repository.

```
sudo install -m 0755 -d /etc/apt/keyrings

sudo curl -fsSL \
  https://download.docker.com/linux/ubuntu/gpg \
  -o /etc/apt/keyrings/docker.asc

sudo chmod a+r /etc/apt/keyrings/docker.asc
```

Docker documentation: installation on Different Linux Distro



Installing Docker Engine on Ubuntu

Host: fresh Ubuntu 24.04 (Noble), with `sudo` access.

Step 3 of 5: Add Docker's APT repository

Select the stable packages for Ubuntu 24.04 and your architecture.

```
sudo tee /etc/apt/sources.list.d/docker.sources \
  >/dev/null <<EOF
Types: deb
URIs: https://download.docker.com/linux/ubuntu
Suites: noble
Components: stable
Architectures: $(dpkg --print-architecture)
Signed-By: /etc/apt/keyrings/docker.asc
EOF
```

Docker documentation: installation on Different Linux Distro



Installing Docker Engine on Ubuntu

Host: fresh Ubuntu 24.04 (Noble), with `sudo` access.

Step 4 of 5: Install Docker Engine and plugins

Refresh the package list, then install Docker, Buildx and Compose.

```
sudo apt-get update

sudo apt-get install -y \
  docker-ce docker-ce-cli containerd.io \
  docker-buildx-plugin docker-compose-plugin
```

Docker documentation: installation on Different Linux Distro



Installing Docker Engine on Ubuntu

Host: fresh Ubuntu 24.04 (Noble), with `sudo` access.

Step 5 of 5: Start Docker and verify installation

Start Docker now, enable it at boot, and run a test container.

```
sudo systemctl enable --now docker
```

```
sudo docker run --rm hello-world
```

Expected result: the test container prints `Hello from Docker!` and exits.

Docker documentation: installation on Different Linux Distro



Installing Forgejo Runner

Run these steps in order, in the **same Bash terminal**.

Step 1 of 6: Install prerequisites

Install the download tools, signature checker and Git.

```
sudo apt-get update
sudo apt-get install -y jq gnupg git
```

Forgejo documentation: installing the runner binary



Installing Forgejo Runner

Run these steps in order, in the **same Bash terminal**.

Step 2 of 6: Prepare a download directory

Select the host architecture and create a private working directory.

```
RUNNER_ARCH="$(dpkg --print-architecture)"
case "$RUNNER_ARCH" in
  amd64|arm64) ;;
  *) echo "Unsupported architecture for this example"; exit 1 ;;
esac

RUNNER_DIR="$(mktemp -d)"
cd "$RUNNER_DIR" || exit 1
```

Forgejo documentation: installing the runner binary



Installing Forgejo Runner

Run these steps in order, in the **same Bash terminal**.

Step 3 of 6: Select the current release

Ask Forgejo's release API for the latest version.

```
RUNNER_VERSION="$(
  curl -fsSL \
    https://data.forgejo.org/api/v1/repos/forgejo/runner/releases/latest |
  jq -er '(.tag_name // .name) | ltrimstr("v")'
)" && printf 'Selected version: %s\n' "$RUNNER_VERSION"
```

Continue after this command successfully prints a version number.

Forgejo documentation: installing the runner binary



Installing Forgejo Runner

Run these steps in order, in the **same Bash terminal**.

Step 4 of 6: Download the binary and its signature

Download both files for the selected version and architecture.

```
RUNNER_BASE="https://code.forgejo.org/forgejo/runner"
RUNNER_FILE="forgejo-runner-${RUNNER_VERSION}-linux-${RUNNER_ARCH}"
RUNNER_URL="${RUNNER_BASE}/releases/download/v${RUNNER_VERSION}"

curl -fL "${RUNNER_URL}/${RUNNER_FILE}" \
  -o forgejo-runner
curl -fL "${RUNNER_URL}/${RUNNER_FILE}.asc" \
  -o forgejo-runner.asc
```

Forgejo documentation: installing the runner binary



Installing Forgejo Runner

Run these steps in order, in the **same Bash terminal**.

Step 5 of 6: Import Forgejo's release signing key

Use the full published fingerprint and a separate temporary keyring.

```
RUNNER_KEYRING="${RUNNER_DIR}/gnupg"
mkdir -m 0700 "$RUNNER_KEYRING"
RUNNER_SIGNING_KEY=EB114F5E6C0DC2BCDD183550A4B61A2DC5923710

gpg --homedir "$RUNNER_KEYRING" \
  --keyserver hkps://keys.openpgp.org \
  --recv-keys "$RUNNER_SIGNING_KEY"
```

Forgejo documentation: installing the runner binary



Installing Forgejo Runner

Run these steps in order, in the **same Bash terminal**.

Step 6 of 6: Verify, install and check the version

Installation proceeds only when signature verification succeeds.

```
gpg --homedir "$RUNNER_KEYRING" \
  --verify forgejo-runner.asc forgejo-runner &&
sudo install -m 0755 forgejo-runner \
  /usr/local/bin/forgejo-runner &&
/usr/local/bin/forgejo-runner --version
```

Expected result: GPG reports `Good signature`, then Forgejo Runner prints its installed version.

Forgejo documentation: installing the runner binary



Enabling Actions and creating the runner

In the Codeberg repository:

1. Open **Settings**, then **Units / Overview**.
2. Enable **Actions** and save.
3. Open **Settings / Actions / Runners**.
4. Choose **Create new runner** and give it a name.
5. Copy the displayed connection values to the host configuration.

The registration flow provides a **UUID** and a secret **token** for `server.connections`.

Registering here limits the runner to this repository. **Keep the real token outside Git**.



A reusable Fortran build environment

Build this image once on the runner host. Save the following as `Dockerfile`:

```
FROM node:24-bookworm
RUN apt-get update \
 && apt-get install -y --no-install-recommends \
      gfortran make cmake ninja-build git curl jq \
      ca-certificates \
 && rm -rf /var/lib/apt/lists/*
```

and in the same folder, run

```
sudo docker build --pull -t ci-fortran:1 .
```

Node.js supports JavaScript actions such as checkout. The compiler and build tools are ready before each job starts.



Runner labels select the job environment

Generate defaults with your installed runner:

```
forgejo-runner generate-config > config.yml
sudo mkdir -p /etc/forgejo-runner/
sudo mv config.yml /etc/forgejo-runner/config.yml
```

Use the guide to save and protect `/etc/forgejo-runner/config.yml`. Selected settings:

```
runner:
  capacity: 1
  timeout: 30m
  labels:
    - "fortran-docker:docker://ci-fortran:1"
container:
  docker_host: "-"
  force_pull: false
```

The workflow's `runs-on: fortran-docker` matches this label. A label names an environment. It does not provision a machine.



Connecting the runner to Codeberg

Add the values supplied by the Codeberg runner dialog:

```
server:
  connections:
    codeberg:
      url: https://codeberg.org/
      uuid: "REPLACE_WITH_RUNNER_UUID"
      token: "REPLACE_WITH_RUNNER_TOKEN"
```

Keep the configuration readable only by the service user and root. Complete the Docker and service configuration from the setup guide.

Use the registration instructions for your installed runner version. Older examples use a deprecated registration command.



Running Forgejo Runner as a service

Step 1 of 11: Check the prerequisites

systemd will supervise the long-running runner process and start it automatically at boot when enabled.

```
/usr/local/bin/forgejo-runner --version
sudo systemctl is-active docker
sudo test -s /etc/forgejo-runner/config.yml &&
  echo "Runner configuration found"
```

The configuration must already contain your Codeberg connection and runner labels.

Forgejo Runner: service setup systemctl documentation



Running Forgejo Runner as a service

Step 2 of 11: Create or reuse the service account

The runner uses a dedicated account with access to Docker.

```
getent group forgejo-runner >/dev/null ||
  sudo groupadd --system forgejo-runner

id -u forgejo-runner >/dev/null 2>&1 ||
  sudo useradd --system --gid forgejo-runner \
    --home-dir /var/lib/forgejo-runner \
    --shell /usr/sbin/nologin forgejo-runner

sudo usermod -aG docker forgejo-runner
```

The account and group are created only if they do not already exist.

Forgejo Runner: service setup systemctl documentation



Running Forgejo Runner as a service

Step 3 of 11: Prepare directories and file permissions

Create the working directory and give the service read access to the existing configuration.

```
sudo install -d -m 0750 \
  -o forgejo-runner -g forgejo-runner /var/lib/forgejo-runner
sudo install -d -m 0750 \
  -o root -g forgejo-runner /etc/forgejo-runner
sudo chown root:forgejo-runner /etc/forgejo-runner/config.yml
sudo chmod 0640 /etc/forgejo-runner/config.yml
sudo -u forgejo-runner test -r /etc/forgejo-runner/config.yml &&
  echo "Runner can read configuration"
```

These commands preserve the configuration contents and credentials.

Forgejo Runner: service setup systemctl documentation



Running Forgejo Runner as a service

Step 4 of 11: Create the unit file

The `[Unit]` section declares dependencies and startup order.

```
sudo tee /etc/systemd/system/forgejo-runner.service >/dev/null <<'EOF'
[Unit]
Description=Forgejo Runner for Codeberg
Wants=network-online.target
After=network-online.target docker.service
Requires=docker.service
EOF
```

Steps 4–7 write **one file**. Run them in order. To repeat file creation, begin again at step 4.

Forgejo Runner: service setup systemctl documentation



Running Forgejo Runner as a service

Step 5 of 11: Append the execution settings

Use `tee -a` to append to the same unit file.

```
sudo tee -a /etc/systemd/system/forgejo-runner.service >/dev/null <<'EOF'
[Service]
Type=simple
User=forgejo-runner
Group=forgejo-runner
SupplementaryGroups=docker
WorkingDirectory=/var/lib/forgejo-runner
ExecStart=/usr/local/bin/forgejo-runner daemon \
  -c /etc/forgejo-runner/config.yml
EOF
```

`ExecStart` launches the daemon using an explicit configuration path.



Running Forgejo Runner as a service

Step 6 of 11: Append the restart and shutdown policy

```
sudo tee -a /etc/systemd/system/forgejo-runner.service >/dev/null <<'EOF'
Restart=always
RestartSec=5
TimeoutStopSec=60
UMask=0077
EOF
```

- Restart after an exit, with a five-second delay.
- Allow up to 60 seconds for the service to stop.
- Create new files with private permissions.

Use `shutdown_timeout:` 30s in the existing `runner:` section of the configuration.

Forgejo Runner: service setup systemctl documentation



Running Forgejo Runner as a service

Step 7 of 11: Append the boot configuration

The `[Install]` section tells systemd how to enable this service.

```
sudo tee -a /etc/systemd/system/forgejo-runner.service >/dev/null <<'EOF'
[Install]
WantedBy=multi-user.target
EOF

sudo chown root:root /etc/systemd/system/forgejo-runner.service
sudo chmod 0644 /etc/systemd/system/forgejo-runner.service
```

The complete unit file now contains `[Unit]`, `[Service]` and `[Install]`.

Forgejo Runner: service setup systemctl documentation



Running Forgejo Runner as a service

Step 8 of 11: Load, enable and start the service

```
sudo systemctl daemon-reload
sudo systemctl enable forgejo-runner.service
sudo systemctl start forgejo-runner.service
```

- `daemon-reload`: read the new unit file.
- `enable`: arrange automatic startup at boot.
- `start`: launch the runner now.

`systemctl enable --now` combines enabling and starting.

Forgejo Runner: service setup systemctl documentation



Running Forgejo Runner as a service

Step 9 of 11: Verify the running service

```
sudo systemctl status forgejo-runner.service --no-pager
sudo systemctl is-enabled forgejo-runner.service
```

Expected results:

- `active (running)` for the process.
- `enabled` for startup at boot.

In Codeberg, open the repository's **Settings / Actions / Runners**. Confirm that the runner is **online** and has the expected label.

Forgejo Runner: service setup systemctl documentation



Running Forgejo Runner as a service

Step 10 of 11: Inspect the daemon logs

Read recent messages:

```
sudo journalctl -u forgejo-runner.service -n 50 --no-pager
```

Follow new messages as they arrive:

```
sudo journalctl -u forgejo-runner.service -f
```

`Ctrl+C` closes the log viewer. The service keeps running.

Forgejo Runner: service setup systemctl documentation



Running Forgejo Runner as a service

Step 11 of 11: Apply later configuration changes

After editing the `.service` file:

```
sudo systemctl daemon-reload
sudo systemctl restart forgejo-runner.service
```

After editing only `/etc/forgejo-runner/config.yml`, restart the service directly.

Perform restarts while the runner is idle, since a restart may interrupt a job.

Forgejo Runner: service setup systemctl documentation



The CI workflow: trigger and runner

Create `.forgejo/workflows/ci.yml`:

```yaml
name: Fortran CI
on:
  push:
    branches: [main]
  workflow_dispatch:

jobs:
  build-and-test:
    runs-on: fortran-docker
    env:
      FC: gfortran
    steps:
      # Add the steps from the next slide here.
```

Pushes to `main` start the workflow. `workflow_dispatch` also allows a manual run.



The CI workflow: checkout, build and test

Under the job's `steps:` key:

```
- name: Checkout code
  uses: https://data.forgejo.org/actions/checkout@v7
  with:
    persist-credentials: false
- name: Compiler version
  run: gfortran --version
- name: Build
  run: make FC="$FC"
- name: Test
  run: make test FC="$FC"
```

Use a complete action URL to select its source explicitly. No compiler installation is needed inside this workflow.



Verifying the first run

Commit and push the workflow from your project checkout:

```
git add .forgejo/workflows/ci.yml
git commit -m "Add Forgejo Fortran CI"
git push origin main
```

Open the repository's **Actions** tab and select the run. Check the checkout, compiler, build and test logs.

- A queued job often indicates an offline runner or a label mismatch.
- A failed step shows the command and its output in the log.
- A green result requires every required command to succeed.



Delivery: a tested source release

On a version tag, build and test the source, then package it in `dist/release/` and publish it:

```yaml
      - name: Publish release
        uses: https://data.forgejo.org/actions/forgejo-release@v2.13.4
        with:
          direction: upload
          url: https://codeberg.org
          repo: ${{ forgejo.repository }}
          tag: ${{ forgejo.ref_name }}
          sha: ${{ forgejo.sha }}
          token: ${{ forgejo.token }}
          release-dir: dist/release
```

The temporary job token publishes the release. Keep the runner token on the host. The guide includes the tag trigger and packaging commands.



Learning by doing: compiler selection

Question: How can the Makefile use the compiler selected by CI?

```
ifeq ($(origin FC),default)
FC = gfortran
endif
FC ?= gfortran
FFLAGS ?= -O2

build/hello: src/hello.f90
    mkdir -p build
    $(FC) $(FFLAGS) -o $@ $<
```

GNU Make has a built-in `FC` value, so `FC ?= gfortran` alone can leave `f77` selected.

```
make FC=gfortran
FC=gfortran make
```

GNU Make: variable origins



Moving to CMake

A minimal `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.16)
project(HelloFortran LANGUAGES Fortran)
add_executable(hello src/hello.f90)
enable_testing()
add_test(NAME hello-output COMMAND hello)
set_tests_properties(hello-output PROPERTIES
  PASS_REGULAR_EXPRESSION "Hello, Fortran!")
```

Replace the Make steps with:

```
FC=gfortran cmake -S . -B build-gfortran -G Ninja
cmake --build build-gfortran
ctest --test-dir build-gfortran --output-on-failure
```

CMake remembers the compiler in the build directory. Use a fresh directory when selecting a different compiler. CMake: the FC environment variable



Learning by doing: extending the pipeline to CMake

- Move the CI to use CMake instead of the Makefile.
- Can we extend the Docker file to use more than one compiler?
- Add tests that verify program output or numerical accuracy.

Windows and macOS jobs require suitable runners and supported execution environments. Changing `runs-on` alone does not create those machines.

Allow workflows on your host only from contributors you trust. Docker job containers reduce accidental interference but share the host kernel.



Table of Contents

3 General Parallel Programming Issues

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



General Parallel Programming Issues

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

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



What do we mean by parallelism?

Definition

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

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



What do we mean by parallelism?

Keywords: FLOP and FLOP/s

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



Levels of parallelism in a computing system

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

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



Programmer's perspective

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



Parallel systems: memory configurations

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

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



Flynn's Taxonomy

Overview

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

Four categories:

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

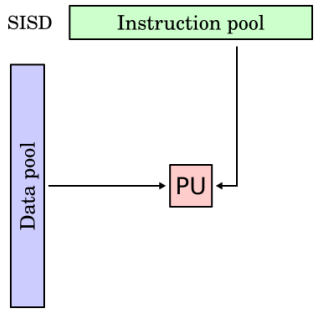


Flynn's Taxonomy: SISD

Single Instruction Single Data

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

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



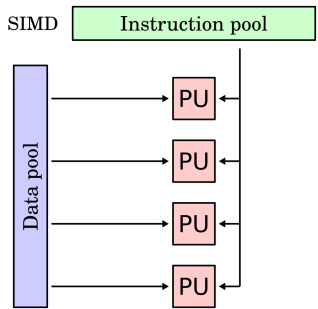


Flynn's Taxonomy: SIMD

Single Instruction Multiple Data

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

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



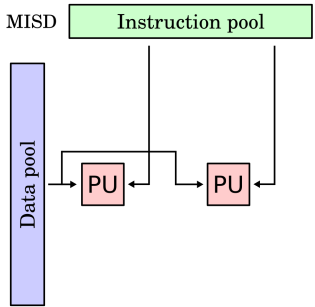


Flynn's Taxonomy: MISD

Multiple Instruction Single Data

Multiple instruction streams operating on the same data.

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



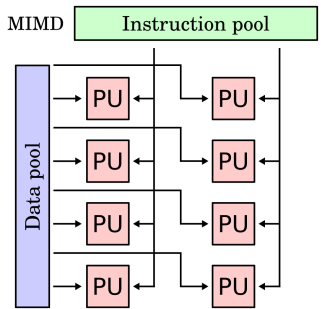


Flynn's Taxonomy: MIMD

Multiple Instruction Multiple Data

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

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





Modern Supercomputers: Beyond Flynn

3 General Parallel Programming Issues

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



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



Anatomy of a Modern Supercomputer

A typical modern supercomputer consists of:

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



The Challenge

Exploiting the computational power of modern supercomputers demands:

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

Coming up: Basic concepts of parallel programming and performance evaluation



Performance metrics in parallel computing

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



Performance metrics in parallel computing

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

No single criterion fits all

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



Beware of benchmarks

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



What do we mean by scalability?

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



Questions a metric should answer

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

Note

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



Definitions of scalability

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



Workload, execution times, and I/O

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

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



Workload, execution times, and I/O

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

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

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

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



Workload, execution times, and I/O

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

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



Execution-time models and benchmarking

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



Definition

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

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

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



Why speed-up is often sub-linear

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

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

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

The speed-up is then:

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

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



When super-linear speed-up may occur

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

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



When super-linear speed-up may occur

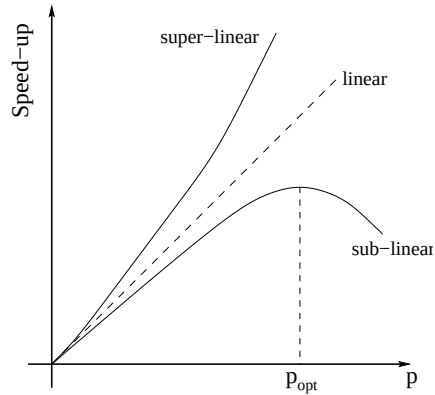
1. Better use of **memory hierarchy**: partitioning reduces working set per process, improving cache and memory behavior (coming back to this in a few slides).
2. **Search problems:** parallel decomposition changes exploration order, finding solutions earlier; sometimes even $S > 1$ on a single core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

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



Speed-up vs number of processors

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

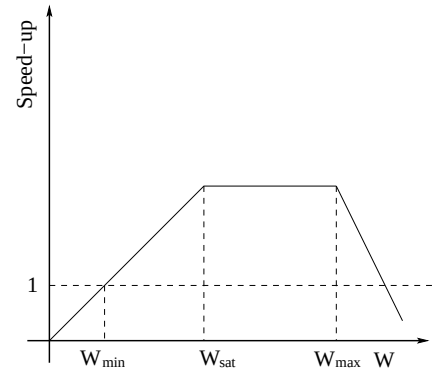


Speed-up vs processors



Speed-up vs problem size

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



Speed-up vs problem size



Efficiency

Definition

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

or, substituting the definition of speed-up

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

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



Two models of scalability

We can think of two complementary models of scalability:

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

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

Strong-scaling vs weak-scaling

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



Amdahl's model

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

Parallel time

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



Amdahl's speed-up and limit

Speed-up

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

Asymptotic limit

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

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



Implications of Amdahl's law

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



Gustafson's law: scaled-speed perspective

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

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

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

Putting it together

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



Table of Contents

4 Paradigms, models and tools for parallel programming

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



Throughput vs parallel applications

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



Paradigms, models, and tools

Paradigm Logical structure imposed on a parallel algorithm.

Model How parallelism is expressed in code.

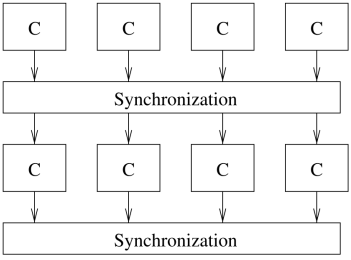
Tool Software instruments (compiler, libraries, etc.).

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



Algorithmic paradigms (1/2)

Phase parallel Alternate independent compute phases and synchronization.



Phase parallel

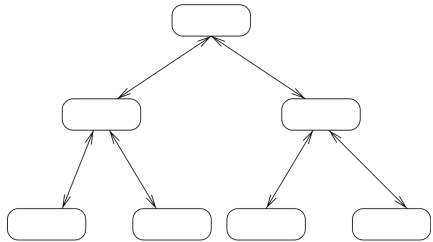
Phase parallel



Algorithmic paradigms (1/2)

Phase parallel Alternate independent compute phases and synchronization.

Divide and conquer Recursively split into subproblems; combine results.



Divide and Conquer

Divide and conquer



Algorithmic paradigms (1/2)

Phase parallel Alternate independent compute phases and synchronization.

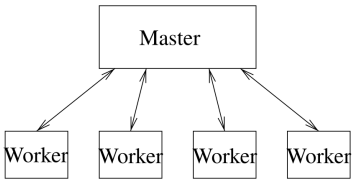
Divide and conquer Recursively split into subproblems; combine results.

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



Algorithmic paradigms (2/2)

Master–worker A controller distributes work and collects results.



Master–Worker

Master–worker

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

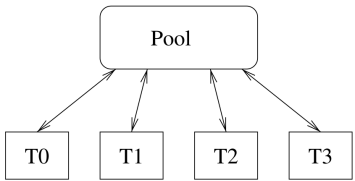


Algorithmic paradigms (2/2)

Master–worker A controller distributes work and collects results.

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

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



Work pool



Programming models

Implicit parallelism Compiler extracts parallelism; often low efficiency.

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

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



Programming models

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

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



Overview

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

Four main tools:

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



MPI: Message Passing Interface

What is MPI?

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

Key features:

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



MPI implementations

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

Note

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

Same code different MPI implementations

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



OpenMP: Shared Memory Parallelism

What is OpenMP?

An API for multi-platform shared memory multiprocessing programming.

Key features:

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

Info: www.openmp.org



OpenACC: Directive-Based Accelerator Programming

What is OpenACC?

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

Key features:

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

Info: www.openacc.org



CUDA: NVIDIA GPU Programming

What is CUDA?

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

Key features:

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

Info: developer.nvidia.com/cuda-zone



The MPI+X framework

Combining tools

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

Common pattern: MPI+X

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



MPI+X: The current standard

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

Research opportunity

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



Summary of Lecture 2

5 Conclusion and summary

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

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

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

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

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



References

6 Bibliography

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

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