High Performance Linear Algebra

Lecture 2: Performance Modeling and the Roofline Model

Ph.D. program in High Performance Scientific Computing

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

November 14, 2025 — 9.00:11.00

Dipartimento
di Matematica
Università di Pisa

Summary of previous lecture

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

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

Table of Contents

2 Continuous Integration and Deployment (CI/CD)

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

An example of CI/CD with GitHub Actions

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

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

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

Makefile basics

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

```
target: dependencies
    command
```

Example:

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

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

Creating a workflow on GitHub Actions

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

Key components:

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

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

An example of CI/CD with GitHub Actions

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

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

An example of CI/CD with GitHub Actions


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

Explanation:

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

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

- Runs on the latest Ubuntu environment

An example of CI/CD with GitHub Actions

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

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

  - Install `gfortran`

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

  - Build the project using `make`

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

An example of CI/CD with GitHub Actions

— Run tests using `make test`

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

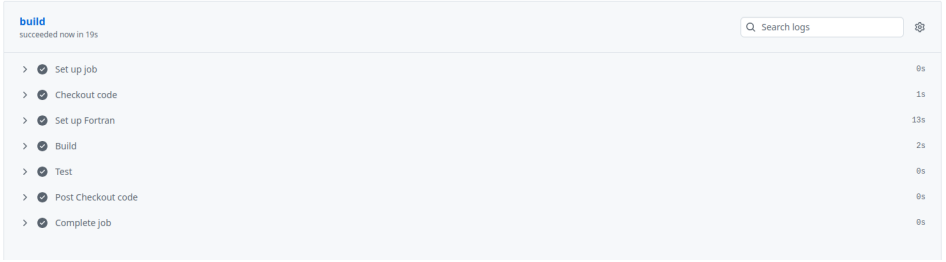
You can change the “manual installation” of `gfortran` with a pre-built action from the GitHub Marketplace, such as setup-fortran:

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

If everything has gone well:

2 Continuous Integration and Deployment (CI/CD)

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



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

Learning by doing:

2 Continuous Integration and Deployment (CI/CD)

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

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

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

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

Further informations on the GitHub Actions

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

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

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

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

Table of Contents

3 General Parallel Programming Issues

General Parallel Programming Issues

3 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:

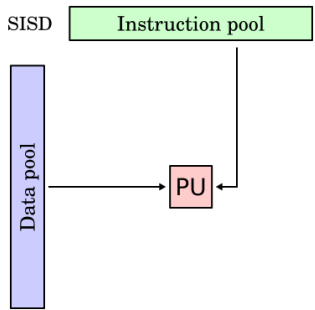
SISD	**S**ingle **I**nstruction **S**ingle **D**ata
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
MISD	**M**ultiple **I**nstruction **S**ingle **D**ata
MIMD	**M**ultiple **I**nstruction **M**ultiple **D**ata

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

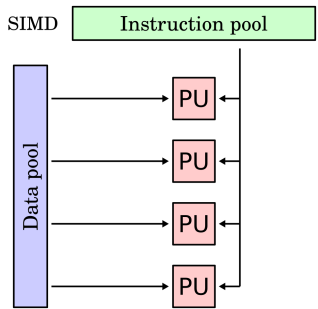


Flynn's Taxonomy: SIMD

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

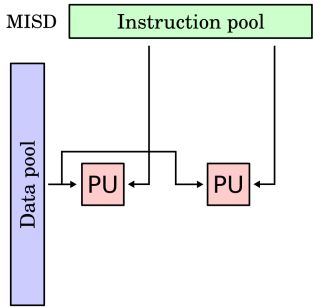


Flynn's Taxonomy: MISD

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

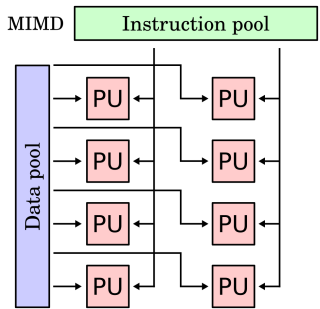


Flynn's Taxonomy: MIMD

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.

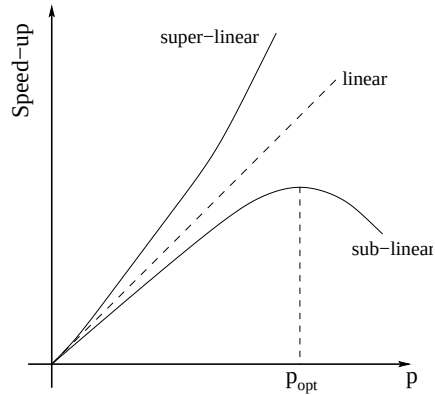
When super-linear speed-up may occur

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

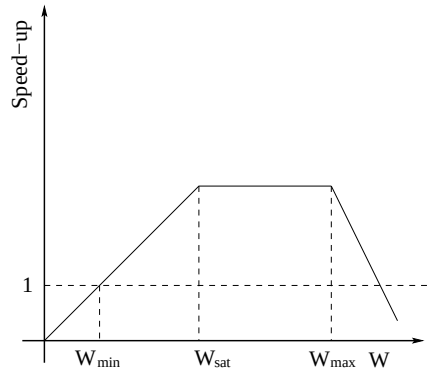
- Linear, sub-linear, and occasional super-linear trends.
- For fixed W, the gap from linear typically widens as p increases due to rising overheads.



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

Throughput vs parallel applications

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

Paradigms, models, and tools

Paradigm Logical structure imposed on a parallel algorithm.

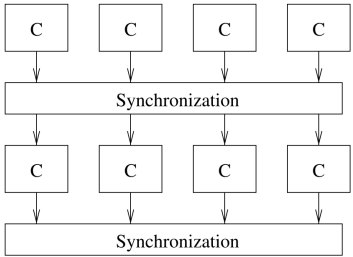
Model How parallelism is expressed in code.

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

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

Algorithmic paradigms (1/2)

Phase parallel Alternate independent compute phases and synchronization.



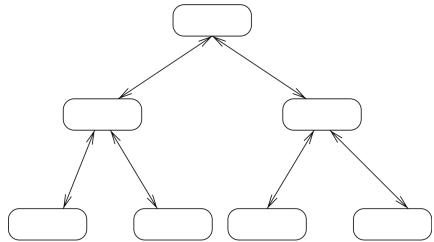
Phase parallel

Phase parallel

Algorithmic paradigms (1/2)

Phase parallel Alternate independent compute phases and synchronization.

Divide and conquer Recursively split into subproblems; combine results.



Divide and Conquer

Divide and conquer

Algorithmic paradigms (1/2)

Phase parallel Alternate independent compute phases and synchronization.

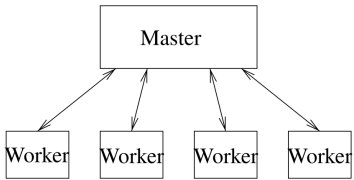
Divide and conquer Recursively split into subproblems; combine results.

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

Algorithmic paradigms (2/2)

Master–worker A controller distributes work and collects results.

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



Master–Worker

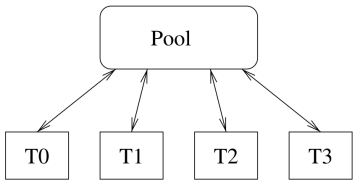
Master–worker

Algorithmic paradigms (2/2)

Master–worker A controller distributes work and collects results.

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

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



Work pool

Work pool

Programming models

Implicit parallelism Compiler extracts parallelism; often low efficiency.

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

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

Programming models

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

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

Programming tools for parallel computing

4 Paradigms, models and tools for parallel programming

Overview

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

Four main tools:

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

MPI: Message Passing Interface

What is MPI?

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

Key features:

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

MPI implementations

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

Note

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

Same code different MPI implementations

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

OpenMP: Shared Memory Parallelism

What is OpenMP?

An API for multi-platform shared memory multiprocessing programming.

Key features:

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

Info: www.openmp.org

OpenACC: Directive-Based Accelerator Programming

What is OpenACC?

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

Key features:

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

Info: www.openacc.org

CUDA: NVIDIA GPU Programming

4 Paradigms, models and tools for parallel programming

What is CUDA?

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

Key features:

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

Info: developer.nvidia.com/cuda-zone

Combining tools

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

Common pattern: MPI+X

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

MPI+X: The current standard

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

Research opportunity

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

Modern Memory Hierarchy

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

Memory Hierarchy Levels

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

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

The Problem

Processor speeds have grown much faster than memory bandwidth improvements

- **Memory wall:** memory latency and bandwidth become the primary bottleneck
- Need tools to understand and visualize this limitation
- Enter: the *Roofline Model* [5]

The Roofline Model: Concept

Definition

A visual performance model relating computational throughput to memory bandwidth

Key hardware characteristics:

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

Key application characteristic:

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

Roofline Model: The Relationship

Fundamental equation

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

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

Roofline Model: Visual Representation

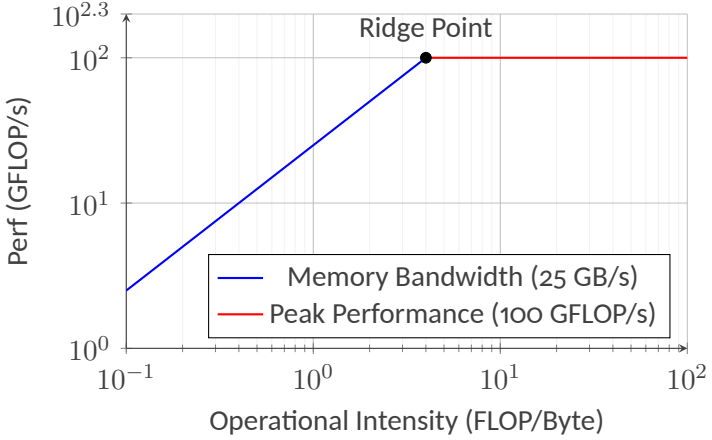


Figure: Roofline model: 100 GFLOP/s peak, 25 GB/s bandwidth

Understanding the Roofline Plot

Two regions:

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

Ridge point:

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

Using the Roofline Model

Applications:

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

Optimization strategy

Compare kernel's OI to ridge point:

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

Roofline and Linear Algebra Evolution

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

More details on BLAS in upcoming lectures

Measuring Memory Bandwidth: STREAM

STREAM Benchmark [3, 4]

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

Four kernels:

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

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

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

Measuring Memory Bandwidth: Example

Let us run try the STREAM benchmark on your machine:

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

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

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

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

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

Example output of STREAM benchmark

4 Paradigms, models and tools for parallel programming

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

Example output of STREAM benchmark


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

How to obtain correct results from STREAM

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

Note

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

How to obtain correct results from STREAM: Example

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

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

On my machine, this returns:

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

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

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

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

How to obtain correct results from STREAM: Modifying the Makefile

Using `awk`

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

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

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

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

Measuring Peak Performance

Estimation formula

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

Example: x86 processor with AVX2

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

Note

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

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

Modern memory hierarchy and Roofline model: for performance analysis

Measuring memory bandwidth: with STREAM benchmark

Next up: 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.

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

References

6 Bibliography

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

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