191.114 Basics of Parallel Computing

Author:

Max Tiessler

2025-04-25

Contents

1 Introduction

This summary is for the course "Basics of Parallel Computing" (course code: 191.114) at the Vienna University of Technology (TU Wien) and was created for the Summer Semester 2025 in preparation for the exam. The document covers most of the essential topics discussed in the course, focusing on key concepts and practical applications in parallel computing. While it aims to provide a comprehensive review of the material, some advanced topics might not be explored in full depth.

Almost all the content has been done following the subject's slides. Also personal examples and explanations from previous courses, personal and professional experiences have been considered while doing this handbook.

This summary is available on GitHub. If you find typos, errors, want to add content (such as additional explanations, links, figures, or examples), or wish to contribute, you can do so at the following repository:
`https://github.com/mtiessler/191.114-Basics-of-Parallel-Computing`.

If the URL does not work, you can locate it on GitHub using the following details:

- **User**: mtiessler
- **Repo-Name**: 191.114-Basics-of-Parallel-Computing

2 Introduction to Parallel Hardware & Basic Notation

2.1 Hardware Fundamentals

2.1.1 Basic Definitions: CPU / Core / Socket

CPU (Central Processing Unit) The CPU, or processor, is the **central unit** that executes instructions stored in main memory. It features **several registers** (i.e., small, fast storage units with **specific bit widths**) for temporarily holding data during computations. A typical CPU is composed of:

- An **Arithmetic Logic Unit (ALU)** for performing arithmetic and logical operations.
- A **Control Unit** that orchestrates the execution sequence.
- Specialized units such as the **Floating Point Unit (FPU)**, which is optimized for floating-point arithmetic.

The CPU performs fundamental operations like load, store, operate, and jump commands.

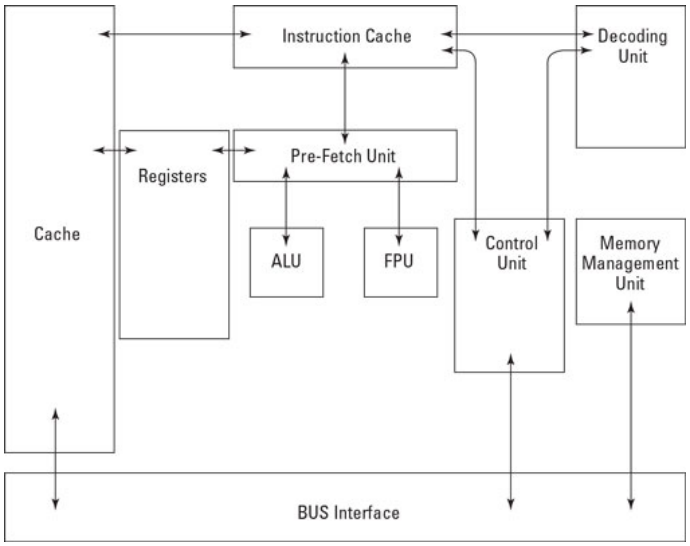


Figure 1: CPU Architecture Components and Bus Connections[1]

Core Modern systems commonly use **multi-core CPUs**, meaning that several cores are integrated onto a single chip. Each core contains its own set of registers, an ALU, an FPU, and other essential components—essentially functioning as an **independent processor**. In contemporary architectures (e.g., those by Intel and AMD), cores typically share a common **Level 3 (L3) cache** (also known as the **Last Level Cache** or LLC), which is the largest and slowest cache in the hierarchy, while each core maintains its own private **Level 1 (L1)** and **Level 2 (L2) caches**. Cores communicate with each other via an **interconnection network** designed with a specific topology to ensure efficient data exchange.

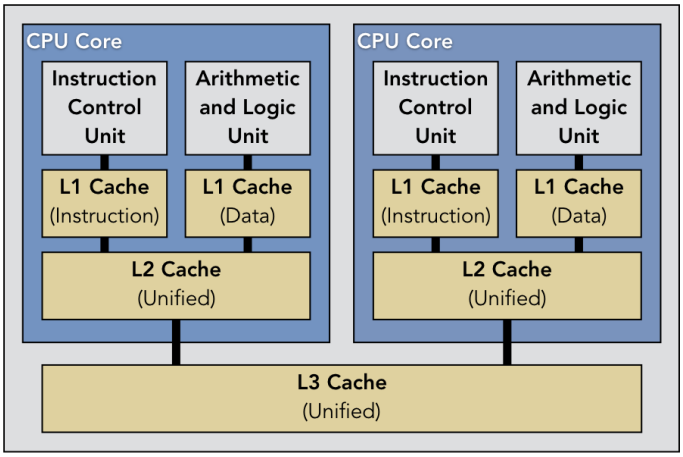


Figure 2: Example of dual-core processor with a shared L3 cache and dedicated L1/L2 caches [1].

Socket A **socket** (or CPU slot) is the physical connector on the motherboard that houses the CPU. Some motherboards provide **multiple sockets**, allowing for multi-processor configurations; an example is a four-socket system arranged in a crossbar configuration.

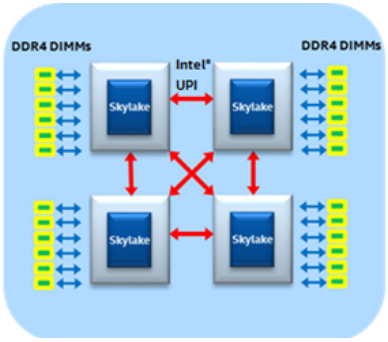


Figure 3: Four-socket crossbar configuration [2]

Altogether, a system may contain **multiple CPUs**, where each CPU resides in its own socket and each CPU can have **multiple cores**, as illustrated below:

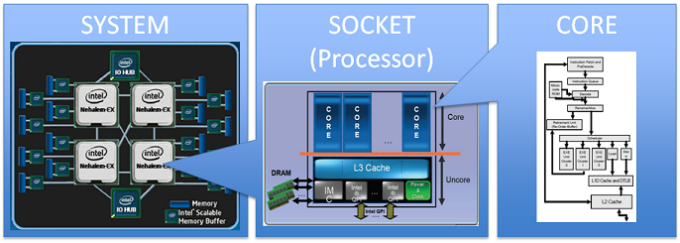


Figure 4: A modern multi-processor, multi-core system [3].

2.1.2 Additional Hardware Definitions

FPU (Floating Point Unit): A **dedicated component** within the CPU for performing floating-point arithmetic operations, crucial for handling real-number calculations.

Registers: High-speed storage locations within the CPU that temporarily hold data and instructions. The size of a register (its bit width) directly influences the amount of data it can process at once.

Cache Memory: A small, **fast memory** component used to store frequently accessed data, reducing access time from the main memory. Modern CPUs usually implement a **multi-level cache hierarchy** (L1, L2, and L3) to optimize performance.

2.2 Advanced Architectures

2.2.1 SIMD Instructions

Modern processors provide **SIMD (Single Instruction, Multiple Data)** instructions, which allow a single instruction to operate on multiple data elements simultaneously. This capability is key for accelerating multimedia processing, scientific computations, and other **data-parallel tasks**.

x86 Architectures x86 is a family of complex instruction set computing (CISC) architectures used primarily in personal computers and servers. SIMD on x86 is implemented through instruction sets such as:

- **SSE (Streaming SIMD Extensions):** Introduced in 1999, supports **128-bit vector operations**.
- **AVX (Advanced Vector Extensions):** Offers vector widths of **256 and 512 bits**, enhancing performance for floating-point intensive applications.

ARM Architectures ARM is a family of reduced instruction set computing (RISC) architectures designed for low power consumption and widely used in mobile and embedded devices. SIMD functionality in ARM is provided by:

- **Neon:** The primary SIMD extension for ARM, supporting **128-bit vector operations**, commonly used in multimedia and signal processing.
- **Helium (M-Profile Vector Extension, MVE):** A more advanced SIMD extension supporting additional vector operations, beneficial for embedded applications.

Additional SIMD Definitions

Instruction Set: A collection of commands that a processor can execute. **SIMD instruction sets** extend these commands to process multiple data elements in parallel.

Vector Operations: Operations that apply a **single instruction simultaneously** to a group of data elements.

Scalar Mode: The mode in which instructions are executed on **individual data elements** one at a time.

2.2.2 UMA & NUMA Architectures

Shared memory parallel computers are systems in which all processors have access to a common, **global address space**. In these systems, the main memory (DRAM) is shared among all processors,

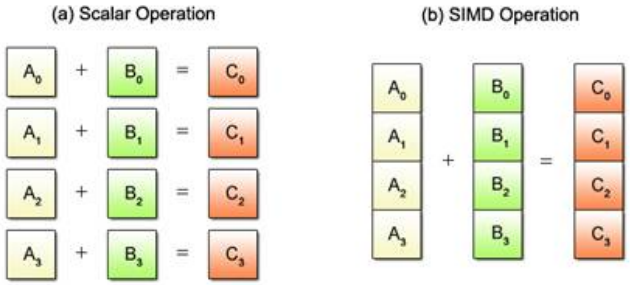
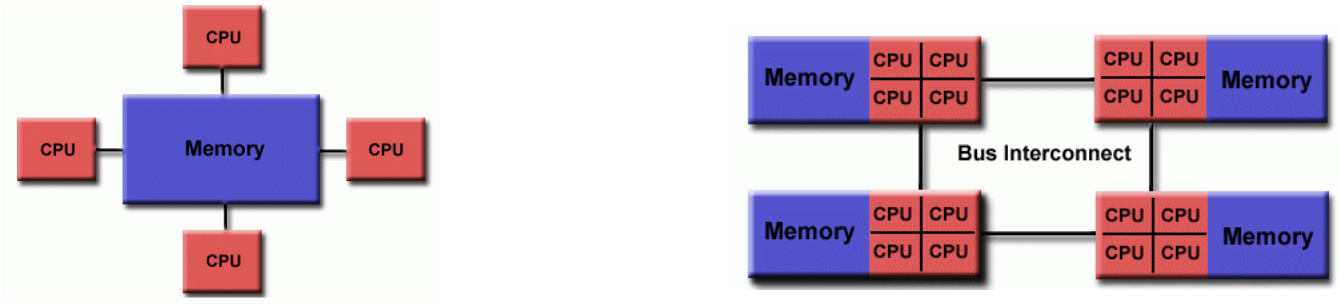


Figure 5: Comparison of SIMD vs. scalar execution.

and any change made by one processor is visible to all others. However, the efficiency of this model depends on how the main memory is physically placed relative to the processors. Multi-processor systems are generally classified into two types based on this placement:

Uniform Memory Access (UMA) UMA systems, typically represented by **Symmetric Multiprocessor (SMP)** machines, feature identical processors with **equal access times** to memory. Often referred to as **CC-UMA (Cache Coherent UMA)**, these systems ensure that cache updates by one processor are immediately visible to others.

Non-Uniform Memory Access (NUMA) In NUMA systems, often formed by linking two or more SMPs, one SMP can directly access the memory of another. This results in different processors having **varying access times**; memory access over inter-SMP links is slower. When cache coherency is maintained, these systems are called **CC-NUMA**.



UMA Diagram

NUMA Diagram

Figure 6: Comparison of UMA and NUMA architectures [4]

General Characteristics Shared memory systems provide the advantage of a global address space that simplifies programming and ensures fast, uniform data sharing due to the proximity of memory to CPUs. However, as more CPUs are added, **scalability issues** can arise. Increased traffic on the memory-CPU path and overhead from maintaining **cache coherency** may limit performance, requiring careful design of **synchronization constructs** to manage access correctly.

2.2.3 GPUs

Graphics Processing Units (GPUs) can be found as dedicated cards (e.g., via PCIe) or integrated into the CPU. The key idea behind GPUs is the use of **many small processing cores**; for example, an RTX 3080 features 8704 cores. These scalar processors execute threads optimized for **data-parallel, throughput computations**, and hide latency by overlapping execution from other threads.

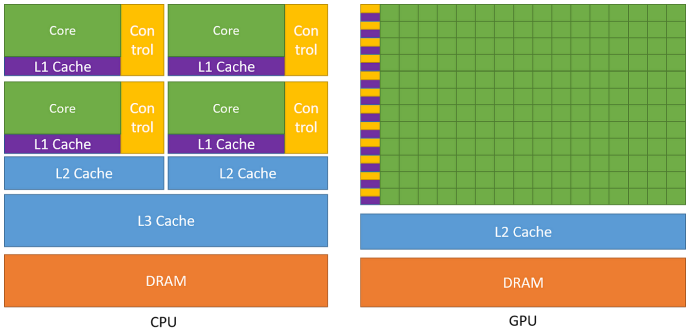


Figure 7: Comparison CPU & GPU [5]

2.3 Parallel Computing Concepts

2.3.1 Parallel Computing

Parallel computing is the **simultaneous use of multiple compute resources** to solve a computational problem. The problem is divided into **discrete parts** that can be solved concurrently, with instructions from each part executing simultaneously on different processors.

2.3.2 Distributed Computing

Distributed computing arises when a problem is solved by a set of **distributed entities** (e.g., processors, nodes, processes, actors, agents, sensors, or peers), where each entity has only **partial knowledge** of the entire problem. Communication among these entities is required to coordinate the solution.

2.3.3 High-Performance Computing (HPC)

High-Performance Computing (HPC) refers to **aggregating computing power** to deliver performance far exceeding that of a typical desktop or workstation. HPC is used to solve large problems in science, engineering, or business.

2.3.4 Supercomputers

A **supercomputer** is a large-scale computer system made up of many smaller computers (**nodes**), **each containing multiple processors or cores**. These nodes communicate over a **high-speed network** (e.g., InfiniBand or Omni-Path). An example is the SuperMUC-NG, which features:

- A total core count of 311,040 (peak performance of 26.9 PetaFlop/s).
- 719 TB of main memory.

- 6,336 Thin compute nodes (each with 48 cores and 96 GB memory) and 144 Fat compute nodes (each with 48 cores and 768 GB memory).
- Intel OmniPath networking in a "fat tree" configuration.

2.3.5 Processes & Threads

Both **processes** and **threads** can execute **independent instructions** and work with **different data**. Within a process, multiple threads can be created.

Processes: Heavier units that have their **own memory space** (address space). Communication between processes requires **inter-process communication (IPC)**, which is generally slower. Context switches are more expensive.

Threads: Lighter units that **share the same memory space**, allowing direct access to shared variables. Context switches between threads are generally **cheaper** than between processes.

Processes	**Threads**
Heavier	Lightweight
Own memory space (address space)	Shared address space
Require inter-process communication (often slow)	Direct access to shared variables
Context switches more expensive	Cheaper context switches

Table 1: Comparison between Processes and Threads

Unique to each **thread** are:

- Thread ID
- Saved registers, stack pointer, and instruction pointer
- Stack (for local variables, temporary variables, and return addresses)
- Signal mask and priority (scheduling information)

Items **shared among threads** within a process include:

- Text segment (instructions)
- Data segment (static and global data)
- BSS segment (uninitialized data)
- Open file descriptors, signals, current working directory, user and group IDs

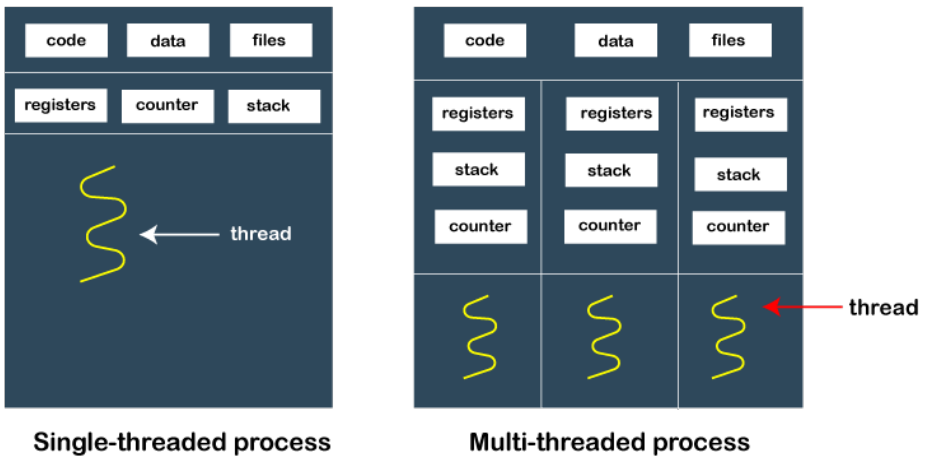


Figure 8: Processes vs Threads [6]

2.3.6 Parallel Computing Hierarchy

Parallelism can be exploited at several levels:

1. **Core Level:** Utilize **SIMD** instructions, **pipelining**, instruction-level parallelism (ILP) and **multiple functional units** (arithmetic logical units (ALUs), floating point units (FPUs)) within a single core.
2. **CPU/GPU Level:** Employ multiple **control flows** using **threads or processes** across **different cores** or GPU multiprocessors.
3. **Cluster Level:** Use several **compute nodes** (each with multiple CPUs) in parallel.

2.3.7 Hyper-Threading

Hyper-Threading (an Intel term for **simultaneous multithreading, SMT**) allows a single CPU core to run **two software threads concurrently** by duplicating parts of the CPU (mainly the registers) while **sharing execution units** (ALUs, FPUs). This makes the CPU appear to have twice as many cores (**logical cores**) as physical ones. It typically increases performance by around 30%, however, **it does not double performance**.

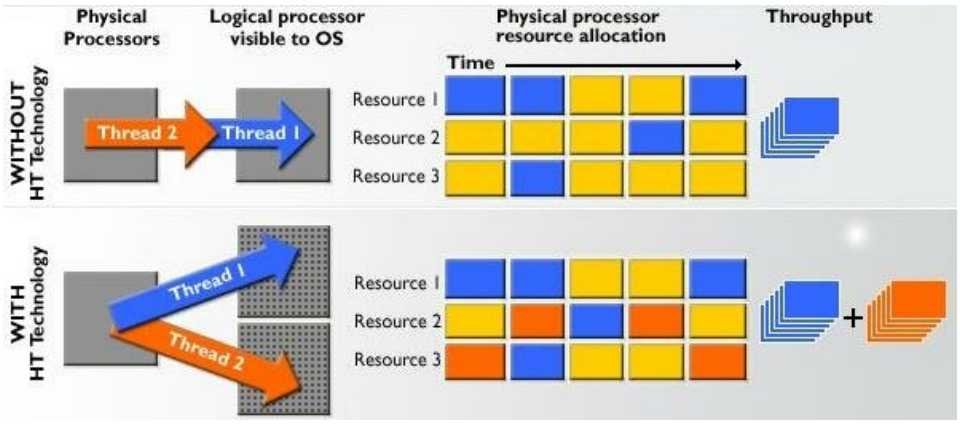


Figure 9: How Hyper-Threading works [7]

2.4 Speedup and Efficiency

2.4.1 Parallel Runtime

The **parallel runtime** is the time that goes from the moment a parallel computation starts to the moment the **last processing element finishes** execution.

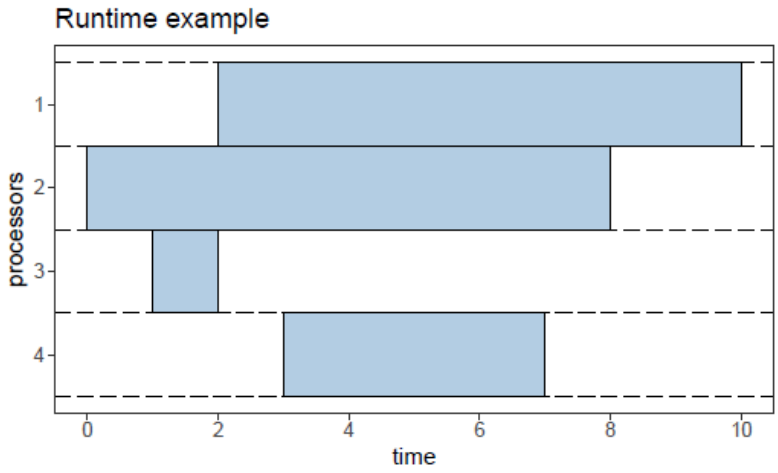


Figure 10: **Parallel runtime example**: Processor 2 starts at time step 0, and processor 1 ends at time step 10. Thus, the total runtime is 10 time steps [8]

2.4.2 Speed-Up

Let:

- p: Number of processors
- n: Input size
- $T_{\text{seq}}(n)$: Sequential running time
- $T_{\text{par}}(n, p)$: Parallel running time with p processors

Absolute Speed-Up
The **absolute speed-up** $S_a(n, p)$ is defined as the ratio of the **best sequential running time** $T_{\text{seq}}(n)$ to the parallel running time on p processors $T_{\text{par}}(n, p)$:

$$S_a(n, p) = \frac{T_{\text{seq}}(n)}{T_{\text{par}}(n, p)}$$

Relative Speed-Up
In many cases, **the sequential code might not be available or executable on a single processor**. In such scenarios, we use the **relative speed-up** $S_r(n, p)$, defined as the ratio of the running time $T_{\text{par}}(n, 1)$ of the parallel code with 1 processor to the parallel running time on p processors $T_{\text{par}}(n, p)$:

$$S_r(n, p) = \frac{T_{\text{par}}(n, 1)}{T_{\text{par}}(n, p)}$$

2.4.3 Parallel Efficiency

While speed-up tells us how much faster a program runs with multiple processors, **it doesn't tell us how efficiently those processors are being used**. This is where **parallel efficiency** comes in.

- A speed-up of 10 using 32 processors is quite good—each processor contributes meaningfully to the performance gain.
- A speed-up of 10 using 1024 processors is far less efficient—most of those processors are underutilized.

Efficiency Definition
Parallel efficiency $E(n,p)$ quantifies how effectively we are using p processors. It is defined as:

$$E(n,p) = \frac{T_{\text{seq}}(n)}{p \cdot T_{\text{par}}(n,p)} = \frac{S_a(n,p)}{p}$$

Interpretation:

- It is the ratio of the actual speed-up to the ideal speed-up (which would be p if all processors were used perfectly).
- An efficiency of 1 (or 100%) means **perfect utilization**—every processor contributes fully.
- An efficiency below 1 indicates that there are **overheads** such as communication, synchronization, or idle time.
- As the number of processors increases, efficiency tends to **decrease** due to these overheads.

Goal: Maximize performance while maintaining high efficiency. Adding more processors doesn't always help if the overhead outweighs the benefit.

2.5 Amdahl's Law

2.5.1 Amdahl's Law Definition

Amdahl's Law describes the **theoretical limit of speed-up** in parallel computing. It states that the **potential speed-up** of **a parallel application** is bounded by the **sequential fraction** of the code.

Let:

- s, where $0 \leq s \leq 1$, be the **sequential fraction** of the code.
- $T^*_{\text{seq}}(n)$ be the **best possible sequential runtime** for input size n.

Then the attainable speed-up using p processors is:

$$S(n,p) = \frac{T^*_{\text{seq}}(n)}{sT^*_{\text{seq}}(n) + \frac{1-s}{p}T^*_{\text{seq}}(n)} = \frac{1}{s + \frac{1-s}{p}}$$

As $p \to \infty$, the speed-up approaches:

$$\lim_{p\to\infty} S(n,p) = \frac{1}{s}$$

This means that no matter how many processors are used, the **maximum speed-up** is bounded by the inverse of the sequential portion.

2.5.2 Modern Hardware Implications

Modern processors do not always run all cores at the same frequency. In fact, the more cores are active, the lower the frequency per core tends to be due to **power and thermal constraints**. This introduces a new constraint on the achievable speed-up.

Breakeven Efficiency The **breakeven efficiency** tells us how well we must utilize multiple cores to match or exceed the performance of a single core running at its maximum frequency.

Active Cores	Max Frequency (GHz)	Breakeven Efficiency
4	2.4	34%
3	2.8	39%
2	3.2	52%
1	3.3	100%

Table 2: Breakeven efficiency for various core counts and frequencies

This table tells us, for example, that if two cores are running at 3.2 GHz, we must achieve at least **52% parallel efficiency** to match the performance of a single core running at 3.3 GHz.

How is this computed? We compare the total processing power of p cores (each running at frequency f_p) to a single core running at its peak frequency f_1:

$$\text{Breakeven Efficiency} = \frac{f_1}{p \cdot f_p}$$

Where:

- f_1 is the frequency of the single fastest core (e.g., 3.3 GHz),
- p is the number of active cores,
- f_p is the max frequency per core when p cores are active.

For example, with $p = 2$ cores at $f_p = 3.2$ GHz:

$$\text{Efficiency} = \frac{3.3}{2 \cdot 3.2} \approx 0.52 \text{ or } 52\%$$

This means that unless the 2-core parallel program is at least 52% efficient, it won't outperform the single-core version running at full speed.

2.5.3 Processor Example: Intel Xeon Gold 6130F

Let's consider the Intel Xeon Gold 6130F processor (used in the Hydra cluster, launched Q3'17):

- 16 physical cores (hyper-threading disabled)
- Base Frequency: 2.10 GHz

- Max Turbo Frequency: 3.70 GHz (for fewer active cores)

As more cores become active, the **maximum turbo frequency per core drops** (e.g., 3.7 GHz for 1-2 active cores, 3.1 GHz for 9-12 active cores, 2.8 GHz for 13-16 active cores). This affects how much speed-up is actually possible even if parallelization is perfect. The true upper bound on performance is not just Amdahl's Law, it's also limited by hardware characteristics like **clock frequency scaling**.

Even with perfect parallel code, hardware constraints such as reduced per-core frequencies mean there's a **hard ceiling on achievable speed-up**. For example, achieving a speed-up of 12 on this CPU would be excellent, as it's likely near the architectural limit imposed by frequency scaling.

2.6 Relative Speed-up in Practice

2.6.1 Scaling, A Matter of Your Reference Point

Consider the task of summing numbers. first from 1 to 1, then 1 to 2, and so on, up to 1 to 10000. When running this simple R code on a machine with 2 physical cores and 4 hyperthreads, some performance improvement using parallelism (via `mclapply` library) is observed.

In this context, the R code seemed to "scale well" when using 2 cores (achieving a relative speed-up of 1.88). However, this observation is based purely on the **relative speed-up**, which compares the performance of the parallel R code with different numbers of threads using the single-threaded R code as a baseline.

2.6.2 Improving the Sequential Baseline

To evaluate the performance more rigorously, the same functionality is implemented in Julia, a language known for high-performance execution.

- Mean time of the Julia sequential program: **85 μs**
- Parallel R code runtime on 4 cores (using 4 threads): **0.59 s** (Note: Slightly different from 0.66s in, likely due to variance or a different run; text uses 0.59s)
- Relative speed-up (R parallel code on 4 cores vs. R single-threaded code): **1.78** (Value from text, slightly different from 2.4 in)

At first glance, a relative speed-up of 1.78 might appear decent. But when we switch the baseline to a truly optimized sequential implementation (in Julia), the picture changes dramatically.

2.6.3 Absolute Speed-up

Using the Julia runtime as the true baseline ($T_{\text{seq}}(n) = 85\mu s$), we can compute the **absolute speed-up** $S_a(n, p = 4)$ of the R parallel version ($T_{\text{par}}(n, p = 4) = 0.59s$):

$$S_a(n, 4) = \frac{T_{\text{seq}}^{\text{Julia}}}{T_{\text{par}}^{\text{R}}} = \frac{85 \times 10^{-6} \text{ s}}{0.59 \text{ s}} \approx 0.00014$$

This extremely small value shows that, in reality, the R parallel code is **much slower** than an optimized sequential alternative.

From this example we can get the following take away message: **relative speed-up can be completely misleading**. Always compare against the **best known sequential algorithm/implementation** when evaluating parallel performance if possible.

2.7 NUMA Performance in Practice

2.7.1 NUMA Topologies and Cache Sharing

The layout of NUMA nodes varies across CPU architectures:

- **AMD EPYC**: Typically features **many NUMA domains** (e.g., 8 domains for 7001 series, potentially fewer like 2 or 4 for 7002 series depending on configuration), with cores grouped into chiplets.
- **Intel Xeon (Hydra Cluster Example - Gold 6130F)**: Often has **fewer NUMA nodes**, typically one per socket (e.g., 2 sockets = 2 NUMA nodes).

It's important to understand how cores and caches are grouped within a NUMA node:

- Cores within a NUMA node typically share caches (e.g., L3).
- Example (AMD EPYC 7551 on 'nebulac'): Cores 0, 16, 32, and 48 share one L3 cache within NUMA node 0.
- The 'nebulac' test system shown has a total of 8 NUMA nodes (2 sockets $\times$ 4 NUMA nodes per socket).
- The 'hydra01' test system shown has 2 NUMA nodes (one per socket).

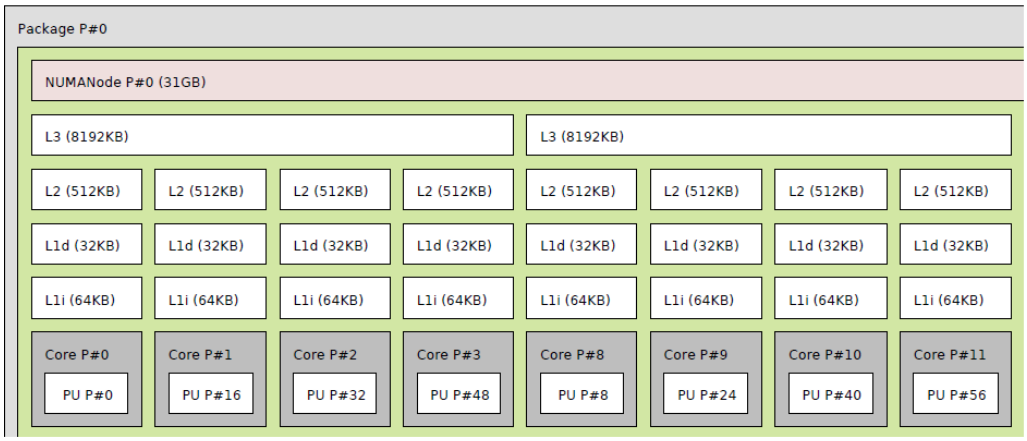


Figure 11: One NUMA node in detail – L3 cache shared between cores (e.g., 0, 16, 32, 48 on AMD EPYC) [8].

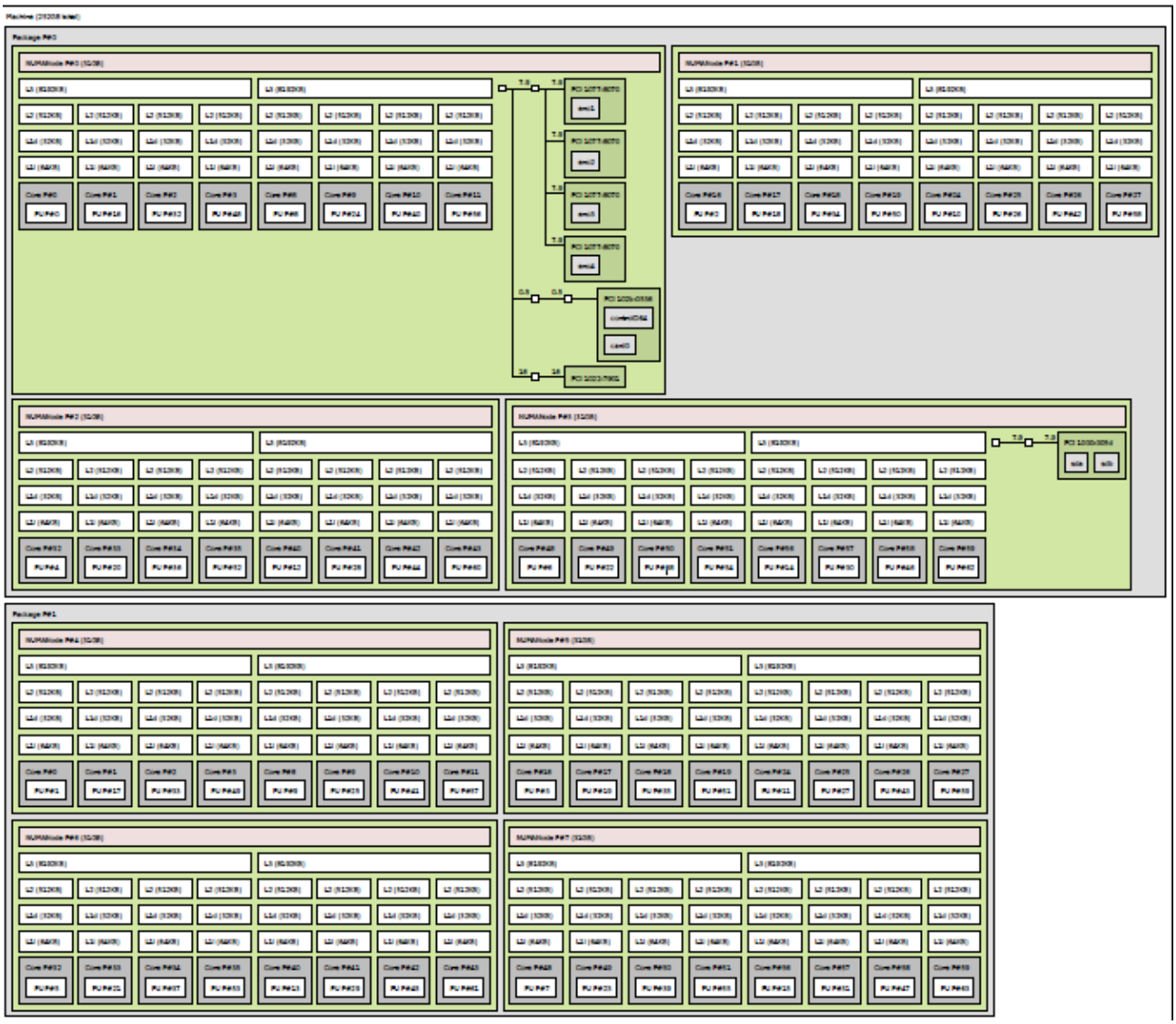


Figure 12: NUMA Topology Example (e.g., Intel Xeon Hydra node where each socket is a NUMA node) [8]. (Image Placeholder)

2.7.2 Memory Bandwidth Experiment (AMD EPYC)

To evaluate how NUMA node placement affects **memory bandwidth**, the `stream_omp` benchmark was run on an AMD EPYC-based system (`nebulac`). The test measures memory throughput (practical bandwidth) under two configurations:

Config 1: Both threads pinned to the **same NUMA node** (cores 0, 16).

Config 2: Each thread pinned to a **different NUMA node** (cores 0, 2).

Hypothesis: Config 2 should show higher throughput, as each thread can potentially access a separate memory controller.

Results (Config 1 – Same NUMA Node):
Using `numactl --physcpubind=0,16 ./stream_omp`:

- Triad throughput: ∼22.7 GB/s.

Results (Config 2 – Different NUMA Nodes):
Using `numactl --physcpubind=0,2 ./stream_omp`:

- Triad throughput: ∼39 GB/s.

Conclusion: Pinning threads to **different NUMA nodes** significantly increased memory throughput (∼23 GB/s to ∼39 GB/s), validating the hypothesis. This highlights the benefit of leveraging multiple memory controllers available across NUMA domains, especially on architectures like AMD EPYC with numerous memory channels.

2.7.3 Memory Latency Between NUMA Nodes (AMD EPYC and Intel Xeon)

The command `numactl -H` was used to investigate memory **access distances** (latency scores) between NUMA nodes.

On the AMD EPYC system ('nebulac'):

- The system has 8 NUMA nodes (0-7).
- Node-to-node distances (latency scores) vary: 10 (local), 16 (neighboring), 22 (intermediate), 28 (remote). Accessing memory on node 7 from node 0 has a higher latency score (28) than accessing memory on node 1 from node 0 (16).

On the Intel Xeon system ('hydra01'):

- The system has 2 NUMA nodes (0-1), one per socket.
- Intra-node distance (local access, e.g., node 0 to 0): 10.
- Inter-node distance (remote access, e.g., node 0 to 1): 21.

Conclusion: Accessing memory from a **remote NUMA node** incurs significantly higher latency. On the Xeon system, remote access latency is roughly **twice as high** as local memory access (score 21 vs 10).

2.7.4 Does Inter-Core Latency Matter? (Intel Xeon / Hydra)

To assess how NUMA node placement impacts **inter-process communication latency**, the **OSU MPI Microbenchmarks** (`osu_latency`) were used on an Intel Xeon Gold 6130F processor (Hydra cluster). This test performs a simple ping-pong message exchange between two MPI processes.

Config 1 – Same Socket (block:block mapping): Both MPI processes are mapped to cores on the **same socket** (same NUMA node).

- Latency for 1-byte message: 0.41 μs.
- Latency remains low up to 128 bytes: ∼0.58 μs.

Config 2 – Different Sockets (block:cyclic mapping): Processes are mapped to cores on **separate sockets** (different NUMA nodes).

- Latency for 1-byte message: 0.65 μs.
- Latency increases more significantly, reaching $\sim$1.01 μs for 128 bytes.

Conclusion: These results confirm that **intra-socket** (local NUMA) communication is significantly **faster** than inter-socket (remote NUMA) communication, especially for small message sizes (0.41 μs vs 0.65 μs for 1 byte).

Takeaway: Placement of threads and processes **matters**. NUMA-aware scheduling and memory allocation can dramatically improve performance. For latency-sensitive applications, **co-locating communicating entities** on the same NUMA node is critical.

2.8 Gustafson-Barsis's Law

Gustafson ([9]) proposed a re-evaluation of Amdahl's law, often referred to as Gustafson's Law or Gustafson-Barsis's Law.

Observation: More powerful computer systems are often used to solve **larger problems**, rather than solving the same size problem in less time. Amdahl's law assumes a **fixed problem size**.

Gustafson's Argument:

- Assume a program has a serial fraction s and a parallel fraction $1 - s$.
- The parallel portion $(1 - s)$ scales perfectly with the number of processors p.
- If the entire workload (serial part s + parallel part $p(1 - s)$ scaled for p processors) were run on a single serial processor, the time taken would be $T_{seq} = s + p(1 - s)$ (assuming the parallel runtime on p processors is normalized to $s + (1 - s) = 1$).
- The parallel runtime on p processors is $T_{par} = s + (1 - s) = 1$.

The **scaled speed-up** (measuring how much more work can be done in the same time) is then:

$$S_{\text{scaled}}(p) = \frac{T_{seq}}{T_{par}} = \frac{s + p(1 - s)}{s + (1 - s)} = s + p(1 - s)$$

This formulation shows that the speed-up is not limited by $1/s$ if the **problem size (workload) scales** with the number of processors. It addresses the question: How much larger a problem can be solved in the same fixed time using p processors compared to 1?

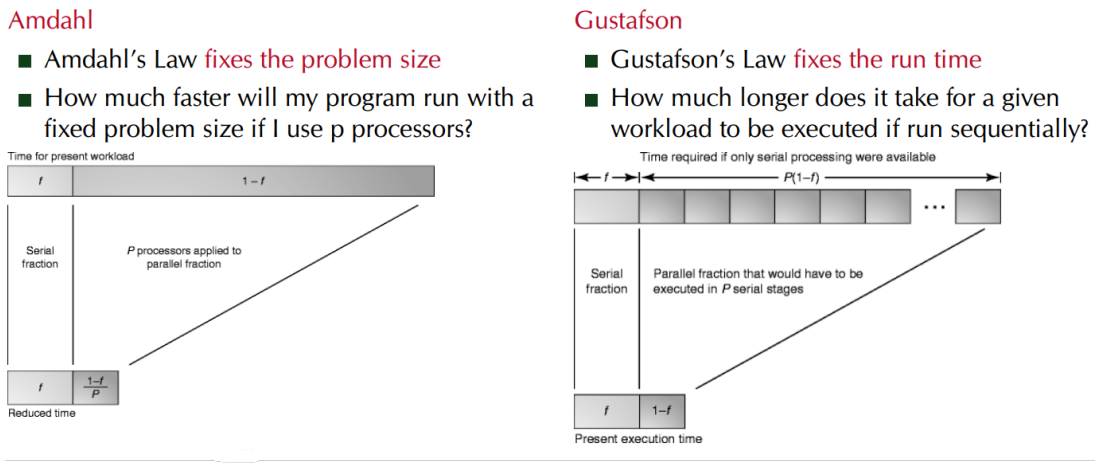


Figure 13: Amdahl vs Gustafson Visualization [10]

2.9 Strong and Weak Scaling

In parallel computing performance analysis, two types of scaling are commonly examined:

Strong Scaling: Based on **Amdahl's Law**, it measures how the execution time varies with the number of processors for a **fixed total problem size**. The goal is to solve the same problem faster. Speedup is typically calculated as $S_p = T_1/T_p$.

Weak Scaling: Based on **Gustafson's Law**, it measures how the execution time varies with the number of processors when the **problem size per processor** is kept **fixed**. The total problem size grows with p. The goal is to solve a larger problem in the same amount of time. Ideally, the execution time T_p remains constant as p increases. Scaled speed-up is often calculated as $S_{\text{scaled}}(p) = p \times (T_1/T_p)$ where T_1 is the time for the base problem size on 1 processor and T_p is the time for the p-times larger problem on p processors, or simply by observing if T_p remains constant.

2.9.1 Scaling Example (R dummy_count)

Using the R 'dummy_count' example on the Hydra cluster (32 cores available per node):

Strong Scaling Experiment

- **Fixed problem size**: `input <- 1:10000`.
- Increase number of cores p from 1 to 32.
- Measure time $T_{\text{par}}(n, p)$ ('tstrong' in results).
- Calculate relative speed-up $S_r(n, p) = T_{\text{par}}(n, 1)/T_{\text{par}}(n, p)$ ('sustrong').
- Results show time decreases (e.g., 1.57s for 1 core to 0.1s for 32 cores) and speed-up increases (e.g., up to 16.24 for 32 cores), but deviates from ideal linear speed-up.

Weak Scaling Experiment

- **Problem size per core fixed**: Base size is `input <- 1:10000` for $p = 1$.
- For p cores, use `rep(input, p)` as the input, so total work grows linearly with p.

- Measure time $T_{\text{par}}(pn, p)$ (`tweak` in results).
- Ideal result: `tweak` remains constant.
- Calculate scaled speed-up $S_{\text{scaled}} = (p \times T_{\text{par}}(n, 1))/T_{\text{par}}(pn, p)$ ('suscaled').
- Results show `tweak` stays relatively constant (e.g., 1.57s for 1 core, 1.7s for 32 cores), indicating good weak scaling. The scaled speed-up approaches linear (e.g., 29.54 for 32 cores).

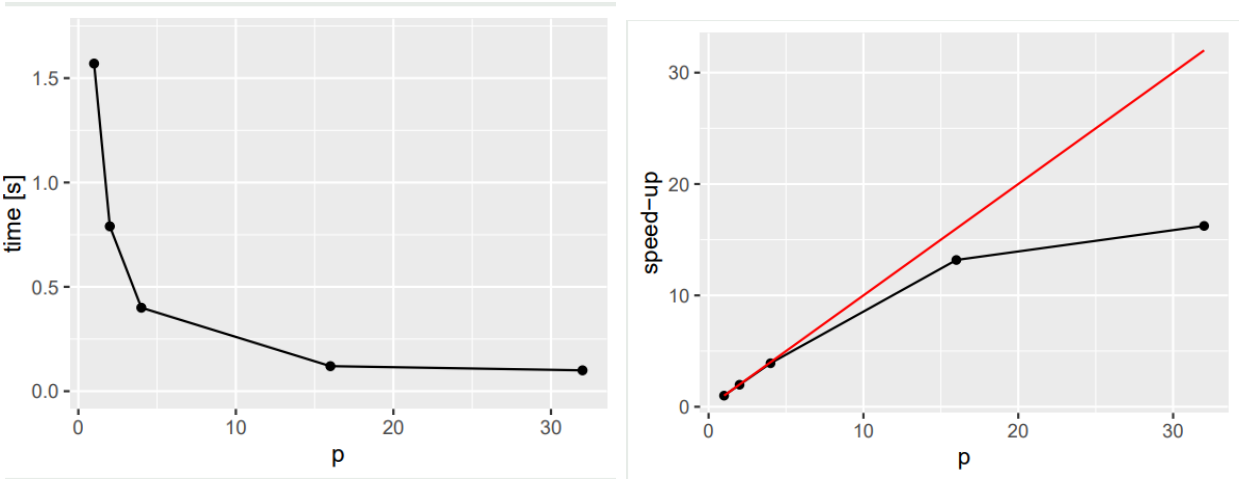


Figure 14: Strong Scaling Results (Time and Speedup)

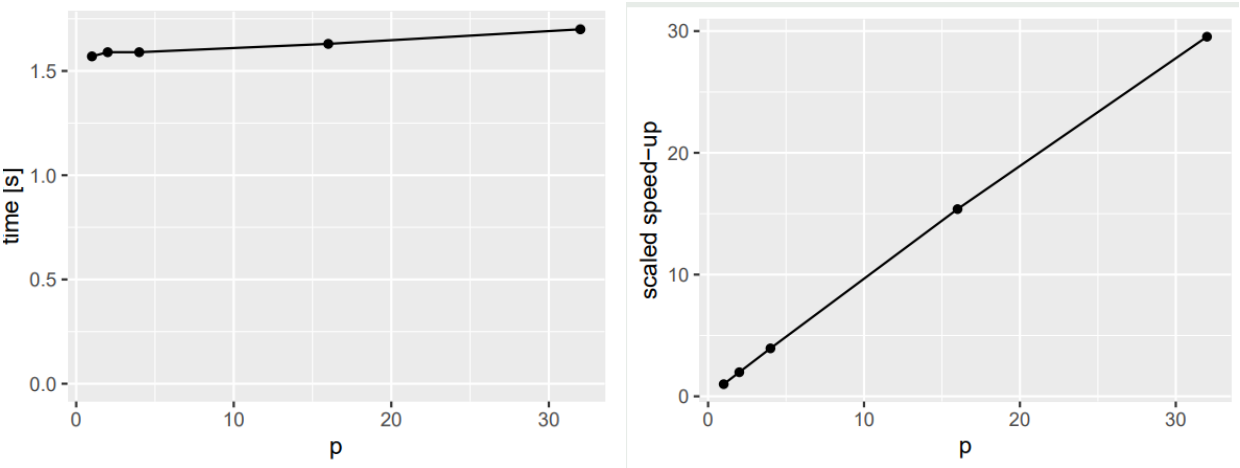


Figure 15: Weak Scaling Results (Time and Scaled Speedup)

2.9.2 Note on Scaling Problem Size

Correctly scaling the problem size for weak scaling is crucial. It depends on the algorithm's complexity.

- For the 'dummy_count' example where work per item j is $O(j)$, the total work for '1:n' is $O(n^2)$. Linearly replicating the input list 'p' times approximately scales the work linearly with p (though not exactly, as the sum depends on the values).

- For matrix-vector multiplication ($n \times n$ matrix), complexity is $O(n^2)$. If work per processor $w_p = 2n^2/p$ is to remain constant ($w_p = w_1 = 2n_1^2$), then the problem size n must scale as $n = n_1\sqrt{p}$. So, for $p = 1, 2, 4, 8...$, n should be $n_1, 1.41n_1, 2n_1, 2.83n_1....$

2.9.3 Embarrassing Parallelism

A problem or workload is called **embarrassingly parallel** (or perfectly parallel) if it requires **little or no effort** to separate the problem into a number of parallel tasks. This often occurs when there is **no dependency or communication** required between the parallel tasks.

- **Characteristics:** Obvious parallelization strategy, minimal communication overhead, easy work splitting.
- **Example:** Parameter sweeps, where the same program is run independently with many different input parameters. Each run is an independent task.

2.10 Cost and Work of Parallel Programs

2.10.1 Cost Definition

The **cost** of a parallel algorithm is defined as the **product** of the number of processors p and the parallel runtime $T_{par}(p, n)$:

$$C(p, n) = p \times T_{par}(p, n)$$

It represents the total time invested across all processors.

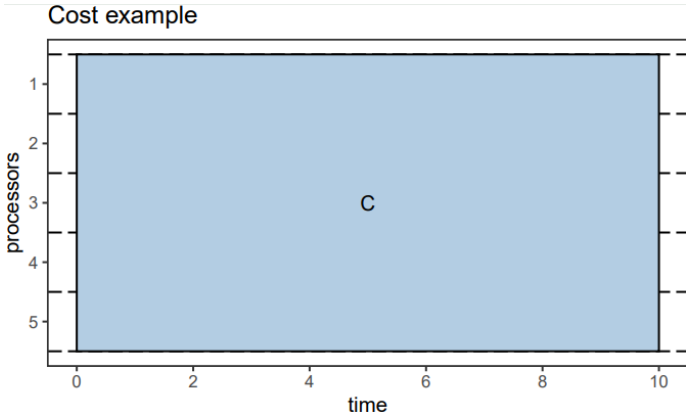


Figure 16: Cost Example Diagram

2.10.2 Cost Optimality

A parallel algorithm is **cost-optimal** if its cost has the **same asymptotic growth** (in terms of input size n) as the runtime of the **fastest known sequential algorithm** $T_{seq}(n)$ [11]:

$$C(p, n) = p \times T_{par}(p, n) \in O(T_{seq}(n))$$

Cost optimality implies that the parallel algorithm does not perform asymptotically more total work than necessary compared to the sequential version. Note that $p \times T_{par} \geq T_{seq}$ usually holds, so cost-optimality means the overhead doesn't dominate asymptotically.

2.10.3 Work Definition

The **work** W of a parallel algorithm is the **total number of operations** performed by all processors during the execution. Work is related to cost but counts operations rather than processor-time product.

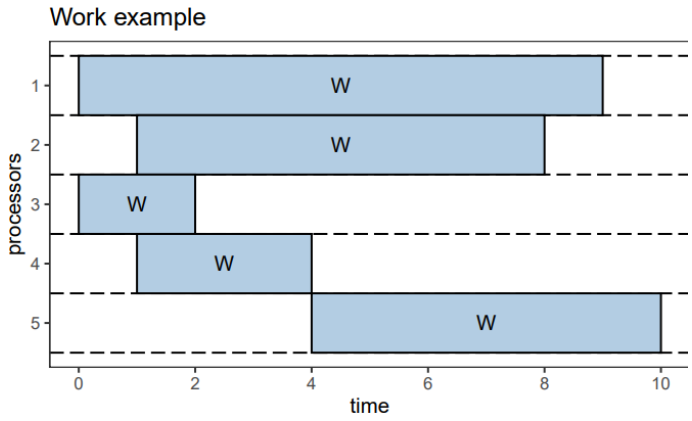


Figure 17: Work Example Diagram

2.10.4 Work Optimality

A parallel algorithm is **work-optimal** if its total work W has the **same asymptotic growth** as the runtime (or number of operations) of the **fastest known sequential algorithm** $T_{seq}(n)$:

$$W \in O(T_{seq}(n))$$

Work optimality focuses purely on the total number of operations performed, ignoring idle time. An algorithm can be work-optimal but not cost-optimal if processors are idle for significant periods. A cost-optimal algorithm is usually also work-optimal.

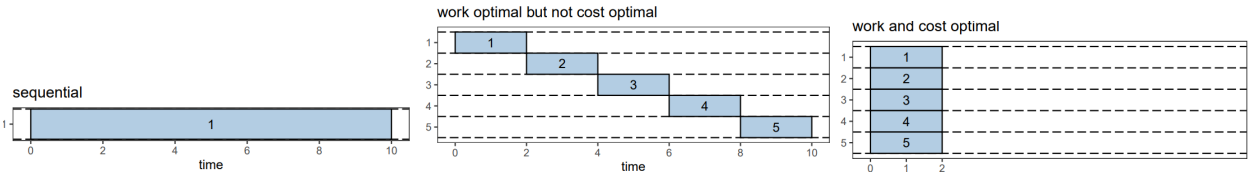


Figure 18: Sequential vs Work/Cost Optimality Examples

2.11 Cost Analysis / Example

2.11.1 Problem: Parallel Summation

Consider adding n numbers using p processors. The best sequential algorithm takes $T_{seq}(n) = \Theta(n)$ time.

Attempt 1: Tree-Based Reduction (n processors)

- Assume n processors, where n is a power of two. Processor i initially holds the i-th number.
- Use a **binary tree structure** to perform pairwise sums in parallel [11].

- In each step, half the active processors send their value to another processor, which computes a partial sum.
- This takes $\log_2 n$ parallel steps (communication + computation).
- Parallel Runtime: $T_{par}(n, n) = \Theta(\log n)$.
- Speed-up: $S_n = T_{seq}/T_{par} = \Theta(n/\log n)$.
- Cost: $C = p \times T_{par} = n \times \Theta(\log n) = \Theta(n \log n)$.
- Is it cost-optimal? Compare $C = \Theta(n \log n)$ with $T_{seq} = \Theta(n)$. Since $n \log n$ grows faster than n, this algorithm is **not cost-optimal**. It uses too many processors relative to the sequential work.

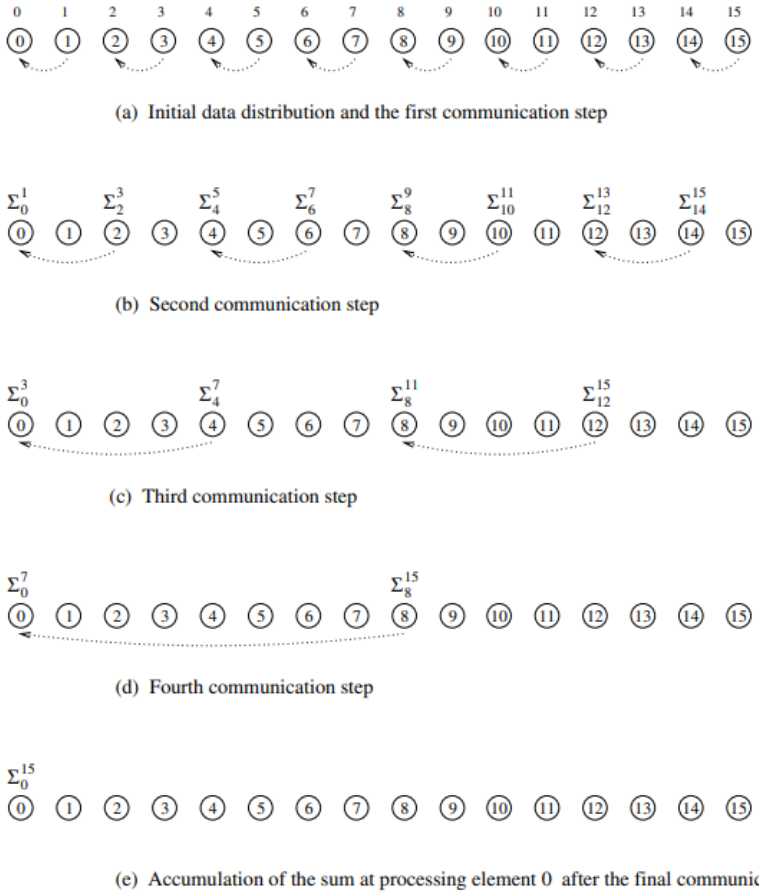


Figure 19: Parallel Summation using Tree Reduction (n processors) [11].

Attempt 2: Simulating n Processors on p Processors

- Assume $p < n$, both powers of two.
- Map the work of the original n virtual processors to the p physical processors. Virtual processor i is simulated by physical processor $i \pmod p$.
- Each physical processor initially handles n/p numbers.
- Simulate the tree-based reduction:

  - The first $\log p$ steps of the original algorithm involve communication between different physical processors. Each step requires n/p work locally plus communication. Total time for these steps might be around $O((n/p)\log p)$.
  - The remaining $\log n - \log p = \log(n/p)$ steps involve communication only between virtual processors mapped to the *same* physical processor. This becomes local computation on each physical processor, adding n/p numbers, taking $O(n/p)$ time.
- Overall Runtime (derived differently in source): The source analyzes the simulation step-by-step, suggesting communication takes $O((n/p)\log p)$ and local computation takes $O(n/p)$, leading to $T_{par}(n,p) = O(n/p + (n/p)\log p) = O((n/p)\log p)$ if $p > 1$.
- Cost: $C = p \times T_{par} = p \times O((n/p)\log p) = O(n \log p)$.
- Is it cost-optimal? Compare $C = O(n \log p)$ with $T_{seq} = \Theta(n)$. Since $\log p > 1$ (for $p > 2$), this is generally **not cost-optimal**.

Attempt 3: Local Summation First

- **Step 1:** Each of the p processors locally sums its assigned n/p numbers. This takes $T_{\text{local}} = O(n/p)$ time, performed in parallel.
- **Step 2:** Now there are p partial sums, one on each processor.
- **Step 3:** Use the tree-based reduction (like Attempt 1, but with p elements/processors) to sum these p partial sums. This takes $T_{\text{reduction}} = O(\log p)$ time.
- Total Parallel Runtime: $T_{par}(n,p) = T_{\text{local}} + T_{\text{reduction}} = O(n/p + \log p)$.
- Cost: $C = p \times T_{par} = p \times O(n/p + \log p) = O(n + p \log p)$.
- Is it cost-optimal? Compare $C = O(n + p\log p)$ with $T_{seq} = \Theta(n)$. The cost is $O(n)$ if n dominates $p \log p$, i.e., if $n = \Omega(p \log p)$. Under this condition, the algorithm **is cost-optimal**. This approach effectively balances local computation with parallel reduction overhead.

3 Introduction to Parallel Programming with R and Python

3.1 Parallel Programming with R & `mclapply`

The `parallel` package in R provides functions for parallel computation. A key function is `mclapply`, which is the parallel version of the base R function `lapply`. `lapply` applies a function to each element of a list sequentially. `mclapply` aims to do this concurrently using multiple cores.

Example of base R `lapply`:

```r
# squaring all elements in this list
squared <- lapply(c(1,2,3,4), function(x) x^2)
print(squared)
# [[1]]
# [1] 1
#
# [[2]]
# [1] 4
#
# [[3]]
# [1] 9
#
# [[4]]
# [1] 16

# Combine results (e.g., sum)
cat("sum:")
Reduce(f="+", squared)
# sum:[1] 30
```

Listing 1: Base R lapply Example

3.1.1 Load Balancing and Scheduling

When applying a function in parallel to a list of inputs (tasks), two key questions arise:

1. How many **workers** (processes or threads) should be created? Typically, this matches the number of available cores.
2. How should work (list elements/tasks) be assigned to workers? The goal is **efficient load balancing**.

Scheduling Strategies in `mclapply`: `mclapply` uses **processes** for parallelism and primarily employs two scheduling strategies controlled by the `mc.preschedule` argument:

- **Static Scheduling (`mc.preschedule = TRUE`, default):** Work is divided among the workers *before* execution starts, typically in a **round-robin** fashion.
  - **Pros:** Fast schedule building, potentially lower overhead for tasks with uniform runtimes.
  - **Cons:** Can lead to poor load balancing for **heterogeneous workloads** (tasks with varying runtimes), as one worker might get stuck with all the long tasks.

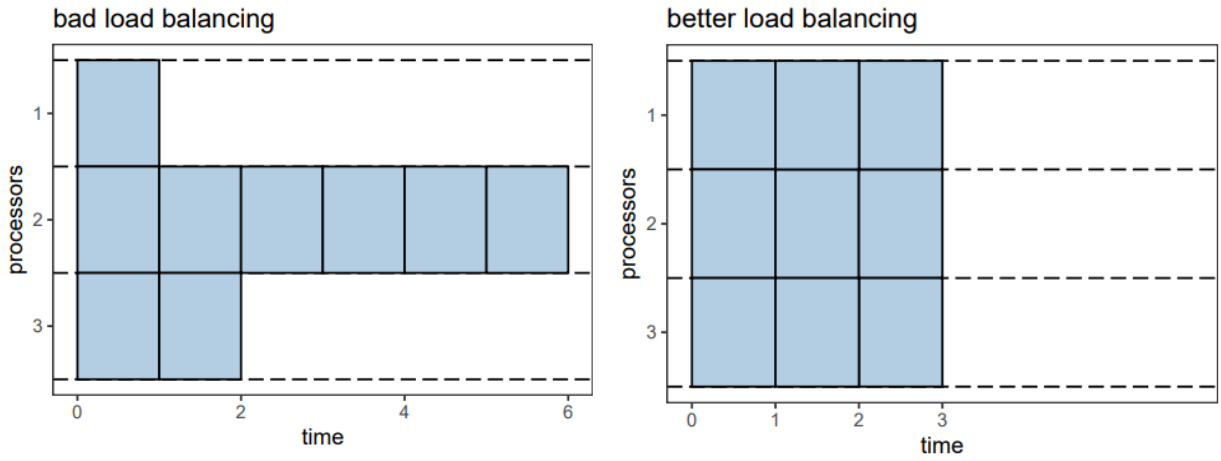


Figure 20: Illustration of Bad vs. Better Load Balancing

Example with heterogeneous tasks (4x 100ms, 4x 1000ms) and static scheduling (2 cores):

```r
# Define a dummy task that waits and returns process ID
dummy_task <- function(wait_ms) {
  Sys.sleep(wait_ms/1e3) # sleep for some time
  Sys.getpid()           # return the current process ID (pid)
}

wait_times <- c(rep(100,4), rep(1000,4)) # Heterogeneous workload
# mc.cores = 2, mc.preschedule = TRUE (default)
res <- mclapply(wait_times, dummy_task, mc.cores = 2)
print(unlist(res) - min(unlist(res))) # Normalized PIDs
# [1] 0 1 0 1 0 1 0 1  <- Round-robin assignment
# Execution time: ~2.2s (dominated by the 4x 1000ms tasks split on 2 cores)

# Unfortunate input order with static scheduling:
wait_times_bad <- rep(c(100,1000), 4)
res_bad <- mclapply(wait_times_bad, dummy_task, mc.cores = 2)
# Execution time: ~4.0s (Process 1 gets all 4x 1000ms tasks)
```

Listing 2: mclapply Static Scheduling Example

- **Dynamic Scheduling** (`mc.preschedule = FALSE`)**:** Tasks are assigned to workers dynamically as they become available. Workers request new tasks when they finish their current one.
  - **Pros:** Better load balancing for heterogeneous tasks.
  - **Cons:** Can have significantly higher overhead, especially for **many small tasks**, as the implementation might involve spawning new processes frequently. In tests with many small tasks, dynamic scheduling was much slower than static scheduling and even sequential execution.

Example with heterogeneous tasks and dynamic scheduling:

```
# (dummy_task defined above)
wait_times_bad <- rep(c(100,1000), 4) # Same unfortunate order
# mc.cores = 2, mc.preschedule = FALSE
res_dyn <- mclapply(wait_times_bad, dummy_task, mc.cores = 2,
                    mc.preschedule = FALSE)
# Execution time: ~2.4s (Better load balancing than static for this input)
# Note: PIDs might be different for each task, indicating process spawning.
```

Listing 3: mclapply Dynamic Scheduling Example

Example comparing scheduling for many small tasks:

```
small_task <- function(a) a**2
n_tasks <- 1000

# Dynamic parallel (mc.preschedule = FALSE)
# Elapsed time: ~4.0s (Very high overhead)

# Static parallel (mc.preschedule = TRUE)
# Elapsed time: ~0.013s

# Sequential (lapply)
# Elapsed time: ~0.004s
```

Listing 4: mclapply Scheduling Comparison (Small Tasks)

Performance Considerations For tasks with very short execution times (low granularity), the overhead of parallelization (process creation, scheduling, data transfer) can outweigh the benefits of parallel execution, potentially leading to parallel execution being slower than sequential execution. However, for larger task granularities or larger overall problem sizes, static scheduling with `mclapply` can provide speedups, although often less than ideal (e.g., speedups ranging from 0.7 to 1.4 observed in an example squaring task, depending on input size).

3.2 Parallel Computing Concepts in Python

Python is widely used, particularly in data science and machine learning, and supports parallel computation. However, understanding the **Global Interpreter Lock (GIL)** is crucial when using threads for parallelism in the standard CPython interpreter.

3.2.1 The Global Interpreter Lock (GIL)

CPython, the default interpreter, uses a **GIL**, which is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecode at the same time within a single process. Even on a multi-core processor, only the thread that holds the GIL can execute Python code. Other threads must wait for the GIL to be released.

- **Consequence:** For CPU-bound tasks (like pure Python calculations), using the `threading` module in CPython typically does **not** achieve true parallelism and can even result in slower execution compared to a sequential version due to locking overhead. Threads are more suitable for I/O-bound tasks where threads can release the GIL while waiting for I/O operations.

- **Workarounds:** Using different interpreters (like Jython, IronPython which don't have a GIL), using the `multiprocessing` module, or using C extensions (e.g., with Cython or libraries like NumPy which release the GIL during computation).

Example demonstrating GIL effect with `threading` for a CPU-bound task:

```python
import time
from threading import Thread

def reduction(n): # Simple CPU-bound task
    sum_val = 0
    for i in range(n):
        sum_val += i
    return sum_val

N = 10**7

# --- Run with 1 thread ---
t1 = Thread(target=reduction, args=(N,))
start = time.time()
t1.start()
t1.join()
end = time.time()
print(f'Time (1 thread): {end - start:.4f}')
# Example Output: Time (1 thread): 0.5602

# --- Run with 2 threads ---
t1 = Thread(target=reduction, args=(N,))
t2 = Thread(target=reduction, args=(N,))
start = time.time()
t1.start()
t2.start()
t1.join()
t2.join()
end = time.time()
print(f'Time (2 threads): {end - start:.4f}')
# Example Output: Time (2 threads): 1.0967 (Roughly double!)
```

Listing 5: Threading GIL Example

3.2.2 Parallelism with `multiprocessing`

The `multiprocessing` package bypasses the GIL limitation by using separate **processes** instead of threads. Each process has its own Python interpreter and memory space.

- **Benefit:** Achieves true parallelism for CPU-bound tasks on multi-core systems, as demonstrated by running two reductions concurrently in roughly the same time as one.
- **Note:** The `multiprocess` package (a fork) offers better compatibility with interactive shells while providing similar functionality.

Example using `multiprocessing` for the same task:

```python
import time
import multiprocessing # or import multiprocess as multiprocessing

# (reduction function defined as above)

N = 10**7

# --- Run with 1 process ---
p1 = multiprocessing.Process(target=reduction, args=(N,))
start = time.time()
p1.start()
p1.join()
end = time.time()
print(f'Time (1 process): {end - start:.4f}')
# Example Output: Time (1 process): 0.6343

# --- Run with 2 processes ---
p1 = multiprocessing.Process(target=reduction, args=(N,))
p2 = multiprocessing.Process(target=reduction, args=(N,))
start = time.time()
p1.start()
p2.start()
p1.join()
p2.join()
end = time.time()
print(f'Time (2 processes): {end - start:.4f}')
# Example Output: Time (2 processes): 0.6777 (Similar to 1 process!)
```

Listing 6: Multiprocessing Example

3.2.3 Common Parallel Paradigms

Worker Pool / Thread Pool A common paradigm involves creating a fixed number of **worker** processes or threads (often matching the number of cores).

- Tasks are placed in a shared **queue** (or channel).
- Idle workers fetch tasks from the queue, execute them, and potentially place results in another queue.
- Python's `multiprocessing.Pool` implements this pattern. It manages a pool of worker processes and provides methods like `map`, `imap`, `apply_async` etc., to distribute tasks.
- **Chunking:** Methods like `map` often divide the input iterable into chunks to reduce scheduling overhead. The **chunk size** can significantly impact performance; too small leads to high overhead, too large leads to poor load balancing.
- **Potential Bottlenecks:** The shared task queue can become a bottleneck if tasks are very small (high contention). Memory usage can be high if `map` needs to materialize the entire input list (`imap` is often preferred for memory efficiency).

Example using `Pool.map`:

```
import os
import multiprocess as mp # Using multiprocess fork

def square(n):
    # print(f'pid {os.getpid()} gets n={n}') # Optional: trace process assignment
    result = n*n
    return result

inputs = list(range(8))
pool = mp.Pool(processes=4)
p_outputs = pool.map(square, inputs) # Default chunking often occurs
pool.close() # No more tasks will be submitted
pool.join()  # Wait for worker processes to exit
print('Outputs:', p_outputs)
# Outputs: [0, 1, 4, 9, 16, 25, 36, 49]
```

Listing 7: Multiprocessing Pool.map Example

Example using `Pool.apply_async` for dynamic task submission:

```
# (square function defined as above)
import multiprocess as mp
pool = mp.Pool(processes=4)
task_handles = []
for i in range(4):
    print(f"Spawning task: n={i+1}")
    t = pool.apply_async(square, (i+1,)) # Submit tasks asynchronously
    task_handles.append(t)

pool.close()
# Get results (blocks until each result is ready)
results = [handle.get() for handle in task_handles]
pool.join()
print('Async Results:', results)
# Async Results: [1, 4, 9, 16]
```

Listing 8: Multiprocessing Pool.apply_async Example

Shared Resources & Synchronization When parallel processes/threads need to access or modify shared data (variables, files, etc.), synchronization is crucial to prevent race conditions and ensure correctness.

- **Critical Section:** A part of the program where shared resources are accessed/modified.
- **Mutual Exclusion:** Ensuring that only one process/thread can execute a critical section at any given time.
- **Locks:** A common mechanism to achieve mutual exclusion. A lock has `acquire()` and `release()` methods. A thread must acquire the lock before entering the critical section and release it upon exiting. Python's `multiprocessing` provides `Lock` objects and synchronized wrappers like `Value` and `Array`. Using locks correctly prevents data corruption but serializes access to the critical section. Incorrect use can lead to errors or deadlocks.

Example of race condition with shared `multiprocessing.Value` without a lock:

```
import multiprocess as mp

def worker_no_lock(n, r, global_val):
    local_sum = 0
    for i in range(r):
        # Simulate some work and calculate a local result
        randnums = [1] * n
        local_sum += sum(randnums)
    # Unprotected update - RACE CONDITION!
    global_val.value += local_sum # Reads, adds, writes - not atomic

globsum_unsafe = mp.Value('i', 0) # 'i' for integer
n = 10
r = 10000
p0 = mp.Process(target=worker_no_lock, args=(n, r, globsum_unsafe))
p1 = mp.Process(target=worker_no_lock, args=(n, r, globsum_unsafe))
p0.start(); p1.start()
p0.join(); p1.join()
print(f"Expected sum: {n*r*2}")
print(f"Unsafe global sum: {globsum_unsafe.value}") # Likely incorrect
# Example Output: Unsafe global sum: 104080 (Incorrect!)
```

Listing 9: Race Condition Example

Example fixing the race condition using the lock associated with `Value`:

```
import multiprocess as mp

def worker_with_lock(n, r, global_val):
    local_sum = 0
    for i in range(r):
        randnums = [1] * n
        local_sum += sum(randnums)
    # Protected update using context manager ('with')
    with global_val.get_lock():
        global_val.value += local_sum

globsum_safe = mp.Value('i', 0) # Default lock=True
n = 10
r = 10000
p0 = mp.Process(target=worker_with_lock, args=(n, r, globsum_safe))
p1 = mp.Process(target=worker_with_lock, args=(n, r, globsum_safe))
p0.start(); p1.start()
p0.join(); p1.join()
print(f"Expected sum: {n*r*2}")
print(f"Safe global sum: {globsum_safe.value}") # Correct
# Example Output: Safe global sum: 200000
```

Listing 10: Using Lock to Prevent Race Condition

Message Passing An alternative to shared memory and locks is **message passing**, where processes communicate by explicitly sending and receiving messages, often via **queues** or **pipes**.

- **Queues (`multiprocessing.Queue`, `JoinableQueue`):** Allow multiple producers and consumers. `JoinableQueue` provides ways to track task completion.
- **Pipes (`multiprocessing.Pipe`):** Typically provide a connection between two processes.
- **Benefit:** Avoids shared state issues but requires explicit communication logic.

Example using `JoinableQueue` for tasks and `Queue` for results:

```
import multiprocess as mp

def worker_queue(taskq, resq):
    while True:
        task_data = taskq.get() # Blocks if queue is empty
        if task_data == 'EXIT':
            taskq.task_done() # Mark exit signal as done
            break
        # Process the task (e.g., unpack data, compute)
        n, r = task_data
        local_sum = 0
        for _ in range(r):
            local_sum += sum([1]*n)
        resq.put(local_sum) # Put result onto result queue
        taskq.task_done()   # Signal that this task is complete

n_cores = 4
n = 10
r = 10000
task_queue = mp.JoinableQueue()
result_queue = mp.Queue()

# Start workers
workers = []
for i in range(n_cores):
    w = mp.Process(target=worker_queue, args=(task_queue, result_queue))
    w.start()
    workers.append(w)

# Put tasks onto the queue
for i in range(n_cores): # Example: one task per worker
    task_queue.put((n, r))

# Wait for all tasks to be processed
task_queue.join()

# Send exit signals and wait for workers to terminate
for i in range(n_cores):
    task_queue.put('EXIT')
for w in workers:
    w.join()

# Collect results
total_sum = 0
while not result_queue.empty():
    total_sum += result_queue.get()

print(f"Expected sum: {n*r*n_cores}")
```

```
print(f"Queue-based sum: {total_sum}")
# Example Output: Queue-based sum: 400000
```

Listing 11: Message Passing with Queues Example

3.2.4 Deadlocks

A **deadlock** occurs when two or more processes/threads are blocked indefinitely, each waiting for a resource held by another process/thread in the cycle [12].

- **Common Causes:**
  - Acquiring multiple locks in different orders across threads [13].
  - Incorrect use of blocking communication calls (e.g., `queue.get()` without proper handling of empty queues or termination signals).
- **Prevention/Handling:** Consistent lock ordering, using timeouts for blocking calls, careful design of process termination and communication protocols.

Example of potential deadlock due to lock ordering:

```
import multiprocess as mp
import time

def worker1(lock1, lock2, n):
    for _ in range(n):
        lock1.acquire()
        # time.sleep(0.001) # Small delays increase deadlock chance
        lock2.acquire()
        # print("w1", end="") # Critical section
        lock2.release()
        lock1.release()

def worker2(lock1, lock2, n):
    for _ in range(n):
        lock2.acquire() # Opposite order!
        # time.sleep(0.001)
        lock1.acquire()
        # print("w2", end="") # Critical section
        lock1.release()
        lock2.release()

if __name__ == "__main__":
    l1 = mp.Lock()
    l2 = mp.Lock()
    n_loops = 1000 # Larger n increases deadlock probability

    p0 = mp.Process(target=worker1, args=(l1, l2, n_loops))
    p1 = mp.Process(target=worker2, args=(l1, l2, n_loops))
    p0.start(); p1.start()
    p0.join(); p1.join() # This might hang if deadlock occurs
    print("\nDone (if no deadlock)")
```

Listing 12: Potential Deadlock with Locks Example

4 Introduction to OpenMP

Open Multi-Processing (OpenMP) is a portable standard Application Programming Interface (API) for programming **shared-memory systems** using threads. It extends sequential languages like C, C++, and Fortran using compiler **directives** (`pragmas`).

OpenMP utilizes a **fork-join parallelism** model. A program starts with a single **master thread**. When a parallel region is encountered, the master thread creates (forks) a **team of threads**. These threads execute the code within the parallel region concurrently. At the end of the parallel region, the threads synchronize (often via an implicit barrier), and only the master thread continues execution until the next parallel region or the program ends.

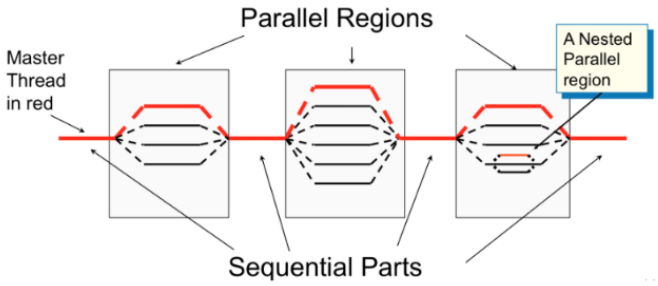


Figure 21: OpenMP Fork-Join Parallelism Model (own resource)

The primary way to specify parallel execution is using the `#pragma omp` directive followed by a construct name and optional clauses.

4.0.1 Core Concepts (Parallel Regions, Loops)

Parallel Regions (`parallel`) The `parallel` directive creates a team of threads and executes the following structured block in parallel by each thread.

```c
#include <stdio.h>
#include <omp.h> // Include OpenMP header

int main() {
    // Creates a team of threads (number determined by environment or clause)
    #pragma omp parallel
    {
        // This block is executed by every thread in the team
        int thread_id = omp_get_thread_num(); // Get unique thread ID
        int num_threads = omp_get_num_threads(); // Get total threads in team
        printf("Hello from thread %d of %d\n", thread_id, num_threads);
    } // Implicit barrier: threads wait here until all have finished
    printf("Back to sequential execution.\n");
    return 0;
}
```

Listing 13: Basic OpenMP Parallel Region

The number of threads can be controlled using the `num_threads` clause or the `OMP_NUM_THREADS` environment variable.

Work-Sharing Loop Construct (`for`) The `for` directive (often combined as `#pragma omp parallel for`) is used to distribute the iterations of a loop across the threads in the team.

- Each thread executes a portion of the loop iterations.
- There is an **implicit barrier** at the end of the `for` loop by default (can be removed with `nowait`).
- The loop must be in **canonical form**:
  - Integer control variable.
  - Control variable not modified inside the loop body (except by the increment).
  - Loop bounds (start, end, step) must be loop invariant.
  - Simple linear increment/decrement (e.g., ‘i++‘, ‘i–‘, ‘i+=step‘).
  - No jumps out of the loop (e.g., ‘break‘, ‘goto‘ to outside label).

```c
#include <stdio.h>
#include <omp.h>

#define N 10

int main() {
    int i;
    #pragma omp parallel for
    for (i = 0; i < N; i++) {
        printf("Iteration %d executed by thread %d\n", i, omp_get_thread_num());
        // Do work for iteration i
    }
    // Implicit barrier here
    return 0;
}
```

Listing 14: OpenMP Parallel For Loop

Using ‘#pragma omp parallel‘ followed by ‘#pragma omp for‘ inside the block is equivalent to the combined ‘#pragma omp parallel for‘.

Nested Loops and Collapse For nested loops, parallelizing only the outer loop might be inefficient if its iteration count is small. If the loops are **perfectly nested** (no code between the ‘for‘ statements), the ‘collapse(n)‘ clause can be used to merge the iteration spaces of the outer ‘n‘ loops and distribute the combined iterations among threads.

```c
#include <stdio.h>
#include <omp.h>

int main() {
    int i, j;
    int n = 2;
    int m = 10;

    // Collapses the two loops into a single iteration space of n*m
    #pragma omp parallel for collapse(2)
    for (i = 0; i < n; i++) {
```

```
        for (j = 0; j < m; j++) {
            printf("Thread %d handles (%d, %d)\n", omp_get_thread_num(), i, j);
        }
    }
    return 0;
}
```

Listing 15: OpenMP Loop Collapse

4.0.2 Data Scoping (Shared, Private, Firstprivate, Lastprivate)

Variables within parallel regions can have different data-sharing attributes:

- **Shared (default for most variables):** Only one instance of the variable exists, accessible by all threads. Access must be synchronized if multiple threads write to it to avoid **data races**.
- **Private:** Each thread gets its own local, uninitialized copy of the variable. Modifications are local to the thread. Loop iteration variables in ‘omp for‘ loops are implicitly private.
- **Firstprivate:** Each thread gets its own local copy, initialized with the value of the original variable from before the parallel region.
- **Lastprivate:** The value of the private variable from the thread that executes the *sequentially last* iteration (for loops) or section is copied back to the original variable after the region.

These attributes are specified using clauses like ‘shared(var1)‘, ‘private(var2)‘, ‘firstprivate(var3)‘, ‘lastprivate(var4)‘.

Example showing default shared behavior (potential race condition):

```
#include <stdio.h>
#include <omp.h>

int main() {
    int a = 2; // a is shared by default
    #pragma omp parallel
    {
        printf("Thread %d sees a = %d\n", omp_get_thread_num(), a);
        // Potential race condition if multiple threads execute this:
        a++;
    }
    // Final value of 'a' is non-deterministic
    printf("Final a = %d (result may vary)\n", a);
    return 0;
}
```

Listing 16: Shared Variable Race Condition

Example using ‘firstprivate‘:

```c
#include <stdio.h>
#include <omp.h>

int main() {
    int a = 2;
    #pragma omp parallel firstprivate(a)
    {
        // Each thread starts with its own 'a' initialized to 2
        printf("Thread %d initial a = %d\n", omp_get_thread_num(), a);
        a++; // Modify local copy
        printf("Thread %d final local a = %d\n", omp_get_thread_num(), a);
    }
    // Original 'a' remains unchanged
    printf("Original a after parallel region = %d\n", a); // Output: 2
    return 0;
}
```

Listing 17: Firstprivate Variable Example

4.0.3 Synchronization Primitives (Barrier, Master, Single, Critical, Atomic)

OpenMP provides directives for coordinating threads:

- `barrier:` ‘#pragma omp barrier‘ imposes an explicit barrier. All threads in the team must reach this point before any can proceed beyond it. Implicit barriers exist at the end of ‘parallel‘, ‘for‘, ‘sections‘, ‘single‘ constructs unless ‘nowait‘ is specified.
- `master:` ‘#pragma omp master‘ specifies a structured block that is executed **only by the master thread** (thread 0). Other threads skip the block. There is **no implicit barrier**.
- `single:` ‘#pragma omp single‘ specifies a structured block executed by **only one arbitrary thread** from the team. Threads not executing the block wait at an implicit barrier at the end, unless ‘nowait‘ is specified.
- `critical:` ‘#pragma omp critical [(name)]‘ defines a **critical section**. Only one thread at a time can execute this block (globally, or for a given optional name). Ensures mutual exclusion for sensitive code updating shared resources.
- `atomic:` ‘#pragma omp atomic [clause]‘ provides mutual exclusion for a single, simple update statement (like increment, decrement, simple binary operations) on a shared memory location. It is often more efficient than ‘critical‘ for simple updates as it may use hardware atomic instructions. Clauses like ‘read‘, ‘write‘, ‘update‘, ‘capture‘ specify the operation type.

Example using ‘critical‘:

```c
#include <stdio.h>
#include <omp.h>

int main() {
    int shared_counter = 0;
    #pragma omp parallel num_threads(4)
    {
        // Simulate some work
```

```
        int thread_id = omp_get_thread_num();

        // Critical section to update shared counter safely
        #pragma omp critical
        {
            shared_counter++;
            printf("Thread %d incremented counter to %d\n", thread_id, shared_
                counter);
        }
    }
    printf("Final counter value: %d\n", shared_counter); // Should be 4
    return 0;
}
```

Listing 18: Critical Section Example

Example using ‘atomic‘:

```
#include <stdio.h>
#include <omp.h>

int main() {
    long long total_sum = 0;
    int N = 1000;

    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        int local_val = i % 5; // Some local computation
        // Atomic update to the shared sum
        #pragma omp atomic update
        total_sum += local_val;
    }
    printf("Final atomic sum: %lld\n", total_sum);
    return 0;
}
```

Listing 19: Atomic Update Example

4.0.4 Scheduling Clauses (`schedule`)

The ‘schedule‘ clause, used with ‘omp for‘, controls how loop iterations are mapped to threads. This impacts **load balancing** and **overhead**.

- `schedule(static [, chunk_size])`: Iterations are divided into chunks (of ‘chunk_size‘ if specified, or roughly ‘num_iterations / num_threads‘ otherwise) and assigned to threads in a fixed, round-robin manner before the loop starts. Low overhead, but potentially poor load balancing for heterogeneous iteration times.
- `schedule(dynamic [, chunk_size])`: Iterations are divided into chunks (default size 1). Threads request and execute one chunk at a time. Better load balancing for heterogeneous iterations, but higher overhead due to dynamic dispatch.
- `schedule(guided [, chunk_size])`: Similar to dynamic, but chunk sizes start large and decrease over time to reduce overhead while still adapting to load imbalance. Minimum chunk size can be set.

- `schedule(auto)`: The compiler/runtime chooses the schedule.
- `schedule(runtime)`: The schedule is determined at runtime via the ‘OMP_SCHEDULE‘ environment variable.

The choice of schedule significantly impacts performance. Static is often good for uniform iterations, while dynamic/guided are better for non-uniform iteration times, but the chunk size needs tuning.

4.0.5 Reductions (`reduction`)

Performing reductions (e.g., sum, product, max, min over elements processed in parallel) safely and efficiently requires special handling. Using ‘critical‘ or ‘atomic‘ on every update is correct but serializes updates and performs poorly. The ‘reduction(operator:list)‘ clause provides an efficient way to perform reductions.

- OpenMP creates a private copy of each variable in the ‘list‘ for each thread.
- Each thread accumulates its partial result into its private copy.
- At the end of the region, the private copies are combined (using the specified ‘operator‘) into the original shared variable.
- Supported operators include ‘+‘, ‘*‘, ‘-‘, ‘‘, ‘—‘, ‘^‘, ‘‘, ‘——‘, ‘min‘, ‘max‘.

```c
#include <stdio.h>
#include <omp.h>

#define N 10000
#define NUM_THREADS 4

int main() {
    int i;
    double sum = 0.0;
    double a[N];

    // Initialize array
    for (i = 0; i < N; i++) a[i] = (double)i;

    // Parallel sum using reduction
    #pragma omp parallel for num_threads(NUM_THREADS) reduction(+:sum)
    for (i = 0; i < N; i++) {
        sum += a[i]; // Each thread adds to its private 'sum'
    }
    // At the end, private sums are added to the original 'sum'

    printf("Parallel reduction sum = %f\n", sum);

    // Verification (Sequential sum)
    double seq_sum = 0.0;
    for (i = 0; i < N; i++) seq_sum += a[i];
    printf("Sequential sum         = %f\n", seq_sum);

    return 0;
}
```

Listing 20: OpenMP Reduction Example

4.0.6 Tasks and Dependencies (`task`, `taskwait`, `depend`)

OpenMP provides task-based parallelism, suitable for irregular or recursive parallelism where work units (tasks) are generated dynamically.

- **`task`:** '#pragma omp task [clauses]' defines a unit of work (code block) that can be executed immediately or deferred for execution by any thread in the team. Data scoping clauses ('shared', 'private', 'firstprivate') apply.
- **`taskwait`:** '#pragma omp taskwait' acts as a barrier for tasks generated within the current task. The current task suspends until all its direct child tasks are complete.
- **`depend`:** The 'depend(dependency-type: list)' clause on a 'task' specifies dependencies between sibling tasks based on shared data, creating a task graph.
  - 'depend(in: var)': Task reads 'var'. Depends on prior tasks with 'out' or 'inout' dependencies on 'var'.
  - 'depend(out: var)': Task writes 'var'. Depends on prior tasks with 'in', 'out', or 'inout' dependencies on 'var'.
  - 'depend(inout: var)': Task reads and writes 'var'. Depends on prior tasks with 'in', 'out', or 'inout' dependencies on 'var'.
- Tasks are typically generated within a 'single' or 'master' region to avoid redundant task creation by all threads.

Example using tasks for Fibonacci (recursive parallelism):

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int fib(int n) {
    int i, j;
    if (n < 2) return n;

    // Create task for fib(n-1) result stored in shared 'i'
    #pragma omp task shared(i)
    i = fib(n - 1);

    // Create task for fib(n-2) result stored in shared 'j'
    #pragma omp task shared(j)
    j = fib(n - 2);

    // Wait for child tasks computing i and j to complete
    #pragma omp taskwait
    return i + j;
}

int main(int argc, char *argv[]) {
    int n = (argc > 1) ? atoi(argv[1]) : 10;
    int fib_res = -1;

    #pragma omp parallel // Start a team of threads
    {
```

```
28          #pragma omp single // Only one thread creates the initial task
29          {
30              fib_res = fib(n);
31          }
32      } // Implicit barrier
33      printf("fib(%d) = %d\n", n, fib_res);
34      return 0;
35  }
```

Listing 21: Fibonacci using OpenMP Tasks

4.0.7 Loop Dependencies

Parallelizing loops requires that iterations are independent or that dependencies are handled correctly.

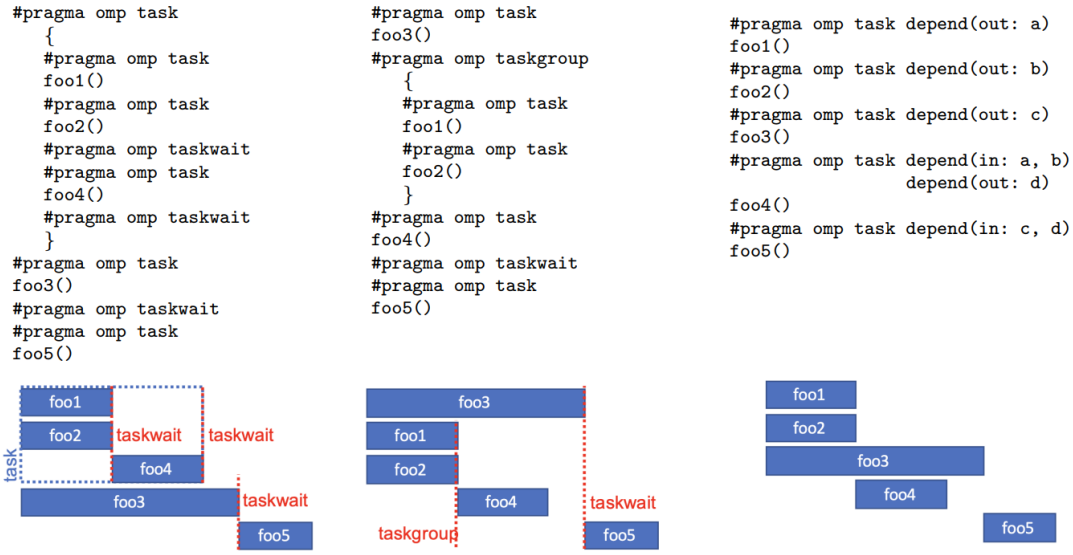


Figure 22: Dependency examples (own resource)

Common dependencies:

- **Flow Dependence (RAW - Read After Write):** Iteration j reads a value written by iteration i ($i < j$). This is a true dependence that often prevents parallelization unless handled (e.g., by privatization or restructuring).
- **Anti Dependence (WAR - Write After Read):** Iteration j writes a value read by iteration i ($i < j$). Can often be resolved by using private variables.
- **Output Dependence (WAW - Write After Write):** Iteration j writes a value also written by iteration i ($i < j$). Can often be resolved by using private variables.
- **Input Dependence (RAR - Read After Read):** Iteration j reads a value also read by iteration i ($i < j$). Does not prevent parallelization.

Loop-carried dependencies (where dependence crosses iterations) are the main concern. Reductions are a common pattern involving loop-carried flow dependence, handled by the 'reduction'

clause. Induction variables (like ‘v‘ in the example below) can sometimes be made private or calculated directly using the loop index [12].

Example with dependencies and resolution:

```
// Original loop with dependencies
double v = start;
double sum = 0.0;
for (int i = 0; i < n; i++) {
    sum = sum + f(v); // Loop-carried flow dep on sum; RAW dep on v
    v = v + step;     // Loop-carried flow dep on v (induction variable)
}

// Parallel version resolving dependencies
double parallel_sum = 0.0;
#pragma omp parallel for reduction(+:parallel_sum) private(v) shared(step, start)
for (int i = 0; i < n; i++) {
    // Calculate 'v' directly based on iteration index 'i'
    v = start + i * step;
    // Reduction handles 'sum' dependency
    parallel_sum = parallel_sum + f(v);
}
// 'parallel_sum' now holds the final result
```

Listing 22: Handling Loop Dependencies

4.0.8 False Sharing

A performance issue in cache-coherent shared-memory systems. It occurs when:

- Different threads access and modify *different* variables.
- These variables happen to reside on the **same cache line** (typically 64 bytes).
- When one thread modifies its variable, the cache coherency protocol invalidates the cache line for other threads, even though they weren't using the specific modified variable.
- This forces other threads to re-fetch the cache line (e.g., from L3 or main memory) upon their next access, causing performance degradation due to unnecessary cache misses and coherency traffic.

This commonly happens with arrays where elements accessed by different threads are close together (e.g., ‘counter[thread_id]‘).

Solution: Pad data structures or allocate data such that variables accessed by different threads are on different cache lines (e.g., ensuring elements are 64 bytes apart on systems with 64-byte cache lines, possibly using aligned memory allocation).

Example demonstrating potential false sharing:

```
#include <stdlib.h>
#include <omp.h>

int main(int argc, char* argv[]) {
    int N = 10000000;
    int num_threads = 4;
```

```
    // Array where each thread increments its own element
    int* counts = calloc(num_threads, sizeof(int)); // Likely fits in one cache
        line

    #pragma omp parallel for num_threads(num_threads) schedule(static)
    for(int i = 0; i < N; i++) {
        int tid = omp_get_thread_num();
        counts[tid]++; // Threads write to adjacent elements -> potential false
            sharing
    }
    // ... print results ...
    free(counts);
    return 0;
}
```

Listing 23: Potential False Sharing Example

Fix using padding (conceptual - requires careful allocation):

```
#include <stdlib.h>
#include <omp.h>
// For posix_memalign:
#define _POSIX_C_SOURCE 200112L
#include <malloc.h> // Or specific header for aligned allocation

#define CACHE_LINE_SIZE 64 // Example cache line size

int main(int argc, char* argv[]) {
    int N = 10000000;
    int num_threads = 4;
    int padding_elements = CACHE_LINE_SIZE / sizeof(int); // Elements per cache
        line

    // Allocate padded array (ensure alignment if possible e.g., posix_memalign)
    int* counts_padded;
    // Allocate memory aligned to cache line size
    // Total size needed: num_threads * elements_per_line * size_of_element
    // For simplicity, let's use padding_elements as stride directly.
    // Allocate enough space for num_threads each potentially padded
    size_t total_size = num_threads * CACHE_LINE_SIZE;
    int err = posix_memalign((void**)&counts_padded, CACHE_LINE_SIZE, total_size)
        ;
    if (err != 0 || counts_padded == NULL) { perror("posix_memalign failed");
        exit(EXIT_FAILURE); }
    // Initialize relevant elements to 0 if needed (calloc doesn't work directly
        with posix_memalign)
    for (int t = 0; t < num_threads; ++t) {
        counts_padded[t * padding_elements] = 0;
    }


    #pragma omp parallel for num_threads(num_threads) schedule(static)
    for(int i = 0; i < N; i++) {
        int tid = omp_get_thread_num();
        // Access element spaced by padding_elements to ensure different cache
            lines
        counts_padded[tid * padding_elements]++;
```

```
    }
    // ... print results from counts_padded[tid * padding_elements] ...
    free(counts_padded);
    return 0;
}
```

Listing 24: Avoiding False Sharing with Padding

4.1 Introduction to the Roofline Model

Performance models are crucial tools in high-performance computing. They help answer the central question: **"Is my program fast enough?**"

Models allow us to:

- Identify performance **bottlenecks**.
- Motivate software **optimizations** or algorithmic changes.
- Determine theoretical performance **limits** on given hardware.
- Predict performance on different architectures.

The **Roofline Model**, introduced by Williams, Waterman, and Patterson [14], is a **visual performance model** designed to provide insight into the performance limitations of computational kernels on multi-core architectures.

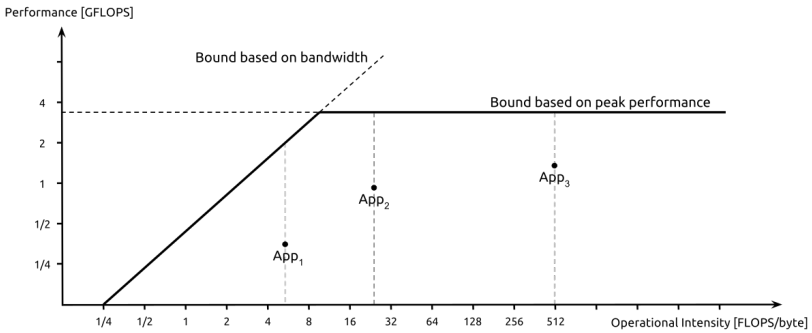


Figure 23: Roofline Model [14]

4.1.1 Motivation and Basic Concepts

The model simplifies the architecture into:

- **Computational Elements** (cores) with a peak computational rate (e.g., GFLOPs/s - Giga Floating Point Operations per Second).
- **Memory Elements** (DRAM) connected via an interconnect with a maximum **memory bandwidth** (e.g., GB/s - Gigabytes per Second).

Programs are viewed as sequences of **computational kernels** (e.g., matrix operations, FFTs, stencil computations). The performance of these kernels is often limited by either the processor's peak computational speed or the memory bandwidth.

4.1.2 Arithmetic Intensity (AI)

A central concept in the Roofline model is **Arithmetic Intensity (AI)**.

- **Definition:** AI is the ratio of total floating-point operations performed by a kernel to the total bytes transferred between memory (DRAM) and the processor (caches).
- **Units:** FLOPs / Byte.
- **Significance:** It quantifies how much computation a kernel performs per byte of data moved. Kernels with high AI are computationally intensive, while those with low AI are memory-intensive.

Calculating AI: Examples

- **Example 1: Vector Dot Product Component** (‘temp += a[i] * a[i]‘)

```
double temp = 0.0;
for (long i = 0; i < N; i++) {
    // Assume a[i] is double (8 bytes)
    temp += a[i] * a[i]; // 1 ADD + 1 MUL = 2 FLOPs
}
```

Listing 25: AI Example 1 Code

  - Operations per iteration: 2 FLOPs (1 ADD, 1 MUL).
  - Memory access per iteration: 1 read of ‘a[i]‘ (8 bytes).
  - AI = (2N FLOPs) / (8N Bytes) = **1/4 FLOPs/Byte**.

- **Example 2: Stencil Computation** (‘C[i][j] = a*A[i][j] + b*(...)‘)

```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        // Assume all are doubles (8 bytes)
        C[i][j] = a * A[i][j] +        // 2 FLOPs
                  b * ( A[i][j-1] +    // 1 FLOP
                        A[i-1][j] +    // 1 FLOP
                        A[i+1][j] +    // 1 FLOP
                        A[i][j+1] );   // 1 FLOP -> Total 6 FLOPs
    }
}
```

Listing 26: AI Example 2 Code

  - Operations per iteration: 6 FLOPs (2 MUL, 4 ADD).
  - Memory access per iteration (simplified, assumes A read once, C written once with write-allocate): Read ‘A[i][j]‘ (8B), Read ‘A[i][j-1]‘ etc. (often cached, but let’s consider boundary case / main memory traffic for simplicity here: assume 1 read for A), Write ‘C[i][j]‘ (8B), Read for ‘C[i][j]‘ due to write-allocate (8B). Total simplified: 24 Bytes. (Note: PDF analysis yields 1/4 based on specific assumptions, possibly considering cache reuse not shown here). For consistency with PDF:
  - PDF Analysis Result: AI = **1/4 FLOPs/Byte**.

- **Example 3: Matrix Multiplication** (‘C[i][j] += A[i][k] * B[k][j]‘)

```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        for (k = 0; k < N; k++) {
            // Assume doubles (8 bytes)
            // 1 ADD + 1 MUL = 2 FLOPs
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Listing 27: AI Example 3 Code

  - Total Operations: $2N^3$ FLOPs.
  - Total Memory access (naive, assumes no cache reuse): Read A ($N^2 \times 8$), Read B ($N^2 \times 8$), Read C ($N^2 \times 8$), Write C ($N^2 \times 8$). Total = $4N^2 \times 8 = 32N^2$ Bytes.
  - AI = ($2N^3$ FLOPs) / ($32N^2$ Bytes) = **N/16 FLOPs/Byte**. AI increases with problem size N.

Determining AI in Practice Analytical estimation is often difficult for complex codes. Hardware **performance counters** can be used to measure:

- Total floating-point operations executed (FLOPs).
- Total bytes transferred between DRAM and CPU (Communication Volume).

Tools like **LIKWID**, **PAPI**, or Linux **perf** can access these counters. AI is then calculated as Measured FLOPs / Measured Bytes.

4.1.3 Constructing the Roofline Plot

The Roofline model plots **Attainable Performance** (GFLOPs/s) on the y-axis (log scale) against **Arithmetic Intensity** (FLOPs/Byte) on the x-axis (log scale).

The attainable performance is limited by the minimum of two factors: the machine’s peak computational rate and the performance achievable given the memory bandwidth. The relationship is derived as follows:

- Time is bound by computation or memory access: $Time = \max\left(\frac{\#FLOPs}{PeakGFLOPs/s}, \frac{\#Bytes}{PeakGB/s}\right)$
- Performance (FLOPs/Time):
  $GFLOPs/s = \frac{\#FLOPs}{Time} = \min\left(PeakGFLOPs/s, \frac{\#FLOPs}{\#Bytes} \times PeakGB/s\right)$
- Substituting AI = $\#FLOPs/\#Bytes$: **Attainable GFLOPs/s** = min(**Peak GFLOPs/s**, **AI**× **Peak GB/s**)

This equation defines the ”roofline”:

- A flat horizontal line (”roof”) representing the **Peak Computational Performance** (π) of the machine.

- A diagonal line with a slope of 1 representing the **Peak Memory Bandwidth** (β) limit (Performance = AI $\times\beta$).
- The actual performance ceiling is the lower of these two lines.
- The point where these lines intersect is called the **ridge point**. Kernels with AI lower than the ridge point are typically **bandwidth-bound**; those with AI higher are potentially **compute-bound**.

Obtaining Machine Parameters

- **Peak GFLOPs/s (π):** Calculated from processor specifications: (Cores $\times$ Frequency $\times$ FLOPs/cycle/core). Requires knowing SIMD width (e.g., AVX2=256-bit), number of Floating Point Units per core, and if FMA (Fused Multiply-Add) is supported.
- **Peak Memory Bandwidth (β):** Measured using benchmarks like **STREAM** (Triad benchmark is often used). Theoretical bandwidth from specifications (Channels $\times$ Frequency $\times$ Bytes/transfer) is often not achievable in practice. Measured STREAM bandwidth provides a more realistic ceiling.

4.1.4 Interpreting the Model and Ceilings

The basic roofline represents theoretical maximums. Actual performance often falls below this due to various bottlenecks, which can be represented as lower, intermediate "ceilings":

- **SIMD Ceiling:** Performance limit if code doesn't utilize SIMD vector instructions effectively (e.g., performance might be halved or quartered depending on SIMD width if only scalar operations are used).
- **Instruction Level Parallelism (ILP) Ceiling:** Limit due to instruction latencies, dependencies, and the processor's ability to hide latency (e.g., if dependent instructions have a 4-cycle latency, peak might be reduced). Loop unrolling can sometimes improve ILP.
- **NUMA Ceiling:** Limit due to inefficient data placement in NUMA systems, causing remote memory accesses and reduced effective bandwidth if code/data is not NUMA-aware (e.g., using first-touch policy for allocation).
- **Other factors:** Cache capacity/bandwidth limitations, thread synchronization overhead, etc.

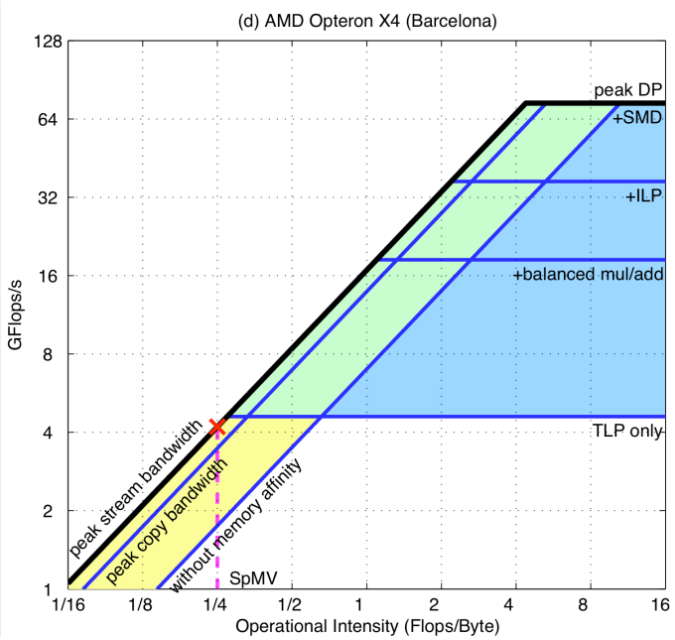


Figure 24: Roofline Ceilings

By plotting a kernel's measured performance (GFLOPs/s) and its calculated/measured AI onto the Roofline chart for a specific machine, one can:

- See if the kernel is bandwidth-bound or compute-bound.
- Identify the performance gap between the current performance and the relevant ceiling (bandwidth or compute).
- Determine which optimizations (e.g., improving memory access patterns, vectorization, increasing ILP, better thread management) might yield the most benefit by moving the kernel's performance point "up" towards the ceiling or potentially "right" by increasing AI (e.g., through caching/data reuse).

5 Introduction to CUDA

Modern computing often utilizes **accelerators** alongside traditional CPUs to boost performance for specific workloads. Common accelerators include **GPUs (Graphics Processing Units)** from vendors like NVIDIA (using CUDA) and AMD (using ROCm), specialized hardware like FPGAs, and historically, co-processors like Intel Xeon Phi.

Programming these accelerators requires specific models and APIs:

- **OpenACC / OpenMP Offloading:** Pragma-based approaches aiming for portability, extending CPU code to offload regions to accelerators. Support and maturity vary.
- **OpenCL (Open Computing Language):** A lower-level, verbose standard aiming for cross-platform execution on CPUs, GPUs, FPGAs, etc.
- **CUDA (Compute Unified Device Architecture):** NVIDIA's proprietary platform and API for programming their GPUs for general-purpose processing (**GPGPU**). It uses a kernel-based approach with C/C++/Fortran language extensions and bindings.

5.0.1 CPU vs GPU Architecture

CPUs and GPUs have fundamentally different design philosophies:

- **CPU (Host):** Designed for low latency execution of sequential or few parallel threads. Uses large caches and complex control logic (branch prediction, out-of-order execution) to make individual threads run fast.
- **GPU (Device):** Designed for high throughput execution of **thousands of parallel threads**. Uses smaller caches and simpler cores, relying on massive parallelism to hide memory latency. Optimized for **data-parallel** computations.

5.0.2 CUDA Programming Model

The CUDA model distinguishes between the CPU (**host**) and the GPU (**device**), each with its own memory space (though Unified Memory aims to simplify this). A typical CUDA application flow involves:

1. Allocating memory on the device.
2. Copying input data from host memory to device memory.
3. Launching a **kernel** (a function executed on the GPU) on the device, specifying the execution configuration (grid and block dimensions).
4. Copying results from device memory back to host memory.
5. Freeing device memory.

Kernels Functions intended to run on the GPU are marked with the '__global__' specifier. They are called from the host using a special triple-chevron syntax: `kernel_name<<<grid_size, block_size, shared_mem_bytes, stream>>>(argument_list);`

- 'grid_size': Specifies the dimensions (1D, 2D, or 3D) of the grid of thread blocks.

- ‘block_size‘: Specifies the dimensions (1D, 2D, or 3D) of threads within each block.
- ‘shared_mem_bytes‘ (optional): Bytes of dynamic shared memory per block.
- ‘stream‘ (optional): Associates the kernel launch with a specific CUDA stream for asynchronous execution.

Kernel launches are asynchronous by default from the host’s perspective. ‘cudaDeviceSynchronize()‘ can be used to make the host wait for device operations to complete.

```
#include <stdio.h>

// Kernel function executed on the GPU device
__global__ void i_say_hello() {
    printf("hello from thread\n");
    // Note: printf from device has limitations and ordering is not guaranteed
}

int main() {
    printf("Call <<<1, 5>>> (1 Block, 5 Threads)\n");
    // Launch 1 block, each containing 5 threads
    i_say_hello<<<1, 5>>>();
    // Wait for the kernel to finish before proceeding
    cudaDeviceSynchronize();

    printf("\nCall <<<2, 1>>> (2 Blocks, 1 Thread each)\n");
    // Launch 2 blocks, each containing 1 thread
    i_say_hello<<<2, 1>>>();
    cudaDeviceSynchronize();

    return 0;
}
```

Listing 28: Simple CUDA Kernel Launch

5.0.3 CUDA Architecture and Thread Hierarchy

CUDA employs a **Single Instruction, Multiple Thread (SIMT)** execution model.

- **Streaming Multiprocessor (SM):** The core processing unit on the GPU. Contains multiple CUDA cores (e.g., 64-128), scheduling units, registers, and shared memory. Modern GPUs contain many SMs (e.g., 80 in RTX 3080 Ti).
- **Warp:** The fundamental unit of scheduling on an SM, typically consisting of **32 threads**. Threads in a warp execute the same instruction concurrently. If threads within a warp diverge (e.g., due to ‘if-else‘ statements based on thread ID), the warp executes each branch path serially, disabling threads not on the current path (**warp divergence**).
- **Thread Block:** A group of threads (up to 1024 on modern GPUs) that execute on the **same SM**. Threads within a block can cooperate using fast **shared memory** and block-level synchronization (‘__syncthreads()‘). Block dimensions can be 1D, 2D, or 3D.
- **Grid:** A collection of thread blocks (potentially 1D, 2D, or 3D) that constitute a single kernel launch. Blocks within a grid can execute independently and potentially concurrently on

different SMs. There is no built-in synchronization between blocks, except implicitly at kernel completion or via special mechanisms like Cooperative Groups.

Figure 25: Mapping of CUDA Threads, Blocks, and Grids to GPU Hardware (Placeholder)

Thread Indexing CUDA provides built-in variables within kernels to identify threads and blocks:

- 'threadIdx.x, y, z': The 1D, 2D, or 3D index of the current thread within its block.
- 'blockIdx.x, y, z': The 1D, 2D, or 3D index of the current block within the grid.
- 'blockDim.x, y, z': The dimensions (number of threads) of the current block.
- 'gridDim.x, y, z': The dimensions (number of blocks) of the grid.

These are used to calculate a unique global index for each thread, typically used to map threads to data elements. For a 1D grid of 1D blocks, a common mapping is: `int global_idx = blockIdx.x * blockDim.x + threadIdx.x;`

It's crucial to include boundary checks `'if (global_idx < N)` in kernels to prevent threads from accessing memory out of bounds, especially when the total number of threads launched ('gridDim.x * blockDim.x') might exceed the actual data size 'N'. Hardware limits exist for block and grid dimensions (e.g., max 1024 threads/block).

5.0.4 CUDA Memory Hierarchy

GPUs have a complex memory hierarchy:

- **Registers:** Fastest memory, private to each thread, limited amount per SM.
- **Local Memory:** Used for register spills and large automatic variables; resides in off-chip device memory (DRAM) but is cached (L1/L2); private to each thread, slow access.
- **Shared Memory:** Low-latency on-chip memory shared by all threads within a **thread block**. Explicitly managed by the programmer. Crucial for performance by enabling data reuse and cooperation within a block.
- **L1/L2 Caches:** Hardware-managed caches for local and global memory accesses. L1 is typically per-SM, L2 is shared across the GPU.
- **Global Memory:** Large off-chip device memory (DRAM), accessible by all threads in the grid and the host (via copies). Highest latency, highest capacity. Performance depends heavily on access patterns (coalescing, alignment).
- **Constant Memory:** Read-only memory cached on-chip, efficient for broadcasting values read by many threads.
- **Texture Memory:** Optimized for spatial locality, read-only, cached on-chip.

5.0.5 Memory Management

- **Explicit Management:** Requires manual allocation on the device (‘cudaMalloc‘), copying data between host and device (‘cudaMemcpy‘ with ‘cudaMemcpyHostToDevice‘, ‘cudaMemcpyDeviceToHost‘), and freeing device memory (‘cudaFree‘). Gives fine-grained control but is verbose.
- **Unified Memory (UM):** Introduced in CUDA 6.0, simplifies memory management. Uses ‘cudaMallocManaged‘ to allocate memory accessible by both host and device using the same pointer/address. The CUDA driver automatically migrates data pages between host and device memory on demand (typically upon access fault). Simplifies programming but may introduce implicit migration overhead. Requires devices supporting Unified Virtual Addressing (UVA).

5.0.6 Global Memory Access Optimization

Achieving high bandwidth from global memory requires careful access patterns:

- **Coalescing:** Memory requests from threads within a warp are combined (coalesced) into fewer, larger transactions if the threads access contiguous memory locations. Ideally, a warp accesses a contiguous 128-byte segment (for 4-byte accesses).
- **Alignment:** Accesses should align with memory segment boundaries (typically 32, 64, or 128 bytes). Misaligned accesses require extra memory transactions, reducing effective bandwidth.

For example, if threads in a warp access elements ‘A[threadIdx.x]‘, the accesses are likely coalesced. If they access ‘A[threadIdx.x * Stride]‘ where Stride is large, accesses are not coalesced. Copying matrix rows (contiguous in row-major layout) often leads to coalesced access, while copying columns does not.

5.0.7 CUDA Beyond C/C++

While CUDA C/C++ is the primary interface, wrappers and libraries exist for other languages:

- **Julia (`CUDA.jl`):** Provides high-level integration, allowing writing GPU kernels directly in Julia. Leverages Julia's compilation capabilities to generate efficient PTX code [15].
- **Python (`PyCUDA`, `Numba`):**
  - ‘PyCUDA‘: Requires writing kernels in C/C++, but provides Python wrappers for compilation, data management, and kernel launch.
  - ‘Numba‘: Uses decorators (‘@cuda.jit‘) to just-in-time compile Python functions (often using NumPy) into CUDA kernels.

Other languages also have varying levels of CUDA support.

6 Introduction to MPI

The **Message Passing Interface (MPI)** is a standardized and portable specification for communication routines, primarily designed for programming **distributed-memory systems** like compute clusters and supercomputers. It defines the semantics of communication functions that are implemented by various MPI libraries (e.g., Open MPI, MPICH, Intel MPI). MPI enables **process-based parallelism**, where independent processes with separate address spaces communicate by explicitly sending and receiving messages.

MPI covers various aspects of parallel programming:

- **Point-to-Point Communication:** Sending/receiving messages between two specific processes.
- **Collective Communication:** Operations involving a group of processes (e.g., broadcast, reduce).
- **Datatypes:** Defining how data is structured and transferred.
- **Communicators and Groups:** Managing subsets of processes for communication.
- **Process Topologies:** Organizing processes logically (e.g., grids).
- **One-Sided Communication (RMA):** Allowing processes to access remote memory without explicit action from the target process.
- Parallel File I/O.
- Tool Support (Profiling, Debugging).

This introduction focuses primarily on point-to-point and collective communication.

6.0.1 Basic MPI Workflow and Core Functions

A typical MPI program follows this workflow:

1. Launch the same compiled program on multiple processes across potentially distributed machines.
2. Each process initializes the MPI environment using `MPI_Init()`. This establishes the initial communication group, `MPI_COMM_WORLD`.
3. Each process determines its unique identifier (**rank**) within the communicator using `MPI_Comm_rank()`. Ranks are numbered $0, 1, ..., p-1$.
4. Each process determines the total number of processes (**size**) in the communicator using `MPI_Comm_size()`.
5. Processes perform computations and communicate using MPI functions (e.g., `MPI_Send`, `MPI_Recv`, `MPI_Bcast`). Communication typically targets specific ranks.
6. Processes finalize the MPI environment using `MPI_Finalize()` before exiting.

A minimal MPI program structure:

```c
#include <stdio.h>
#include <mpi.h> // Include MPI header

int main(int argc, char* argv[]) {
    int rank, size;

    // Initialize MPI Environment
    MPI_Init(&argc, &argv);

    // Get rank and size within MPI_COMM_WORLD
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    printf("This is rank %d of %d\n", rank, size);

    // Perform computations and communication here...

    // Finalize MPI Environment
    MPI_Finalize();

    return 0;
}
```

Listing 29: Basic MPI Program Structure

MPI programs are typically launched using commands like ‘mpirun‘ or ‘mpiexec‘, specifying the number of processes (‘-np‘). For example: ‘mpirun -np 4 ./my$_m pi_p rogram$‘.

6.0.2 Point-to-Point Communication

This involves sending a message from one specific process to another.

Core Functions: `MPI_Send` and `MPI_Recv`

- `MPI_Send(buffer, count, datatype, dest, tag, comm)`: Sends data.
  - ‘buffer‘: Memory address of the data to send.
  - ‘count‘: Number of elements to send.
  - ‘datatype‘: MPI type of the elements (e.g., ‘MPI_INT‘, ‘MPI_FLOAT‘, ‘MPI_DOUBLE‘).
  - ‘dest‘: Rank of the destination process.
  - ‘tag‘: An integer message identifier to differentiate messages.
  - ‘comm‘: The communicator (group of processes) involved (e.g., ‘MPI_COMM_WORLD‘).
- `MPI_Recv(buffer, count, datatype, source, tag, comm, status)`: Receives data.
  - ‘buffer‘: Memory address where received data should be stored.
  - ‘count‘: Maximum number of elements the buffer can hold.
  - ‘datatype‘: MPI type of the elements expected.

- 'source': Rank of the sending process (can be 'MPI_ANY_SOURCE').
- 'tag': Expected message tag (can be 'MPI_ANY_TAG').
- 'comm': The communicator.
- 'status': An 'MPI_Status' object providing details about the received message (actual source, tag, count), or 'MPI_STATUS_IGNORE'.

Example: Rank 0 sends an integer to Rank 1.

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[]) {
    int rank, size;
    int number;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (size < 2) { /* Error handling */ MPI_Finalize(); return 1;}

    if (rank == 0) {
        number = 10;
        printf("Process 0 sending %d to process 1\n", number);
        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received number %d from process 0\n", number);
    }
    // Other ranks do nothing in this example

    MPI_Finalize();
    return 0;
}
```

Listing 30: Simple MPI Send/Receive

Blocking Communication and Deadlocks 'MPI_Send' and 'MPI_Recv' are typically **blocking** communication operations.

- 'MPI_Recv' blocks until a matching message arrives.
- 'MPI_Send' blocks until the message data has been safely stored (either copied to an internal MPI buffer or transferred to the receiver), allowing the send buffer to be reused [16]. For large messages, 'MPI_Send' often blocks until the matching 'MPI_Recv' is posted.

This blocking nature can lead to **deadlocks** if send/receive calls are not ordered correctly. For example, if two processes both call 'MPI_Send' to each other simultaneously with large messages, both might block waiting for the corresponding 'MPI_Recv', which is never posted.

Example of potential deadlock (symmetric ping-pong):

```c
// Assume rank 0 and rank 1, large nb_ints
int partner = 1 - rank; // Calculate partner rank (0->1, 1->0)
```

```
int *send_buffer, *recv_buffer;
// ... allocate buffers ...

// !! DANGEROUS: Both processes send first !!
MPI_Send(send_buffer, nb_ints, MPI_INT, partner, 0, MPI_COMM_WORLD);
MPI_Recv(recv_buffer, nb_ints, MPI_INT, partner, 0, MPI_COMM_WORLD, MPI_STATUS_
    IGNORE);

// ... free buffers ...
```

Listing 31: Potential Deadlock with Blocking Send/Recv

A safe way to implement ping-pong with blocking calls is to use asymmetric logic (one rank sends first, the other receives first) or use ‘MPI_Sendrecv‘.

Non-blocking Communication MPI also provides non-blocking versions: ‘MPI_Isend‘ and ‘MPI_Irecv‘.

- These functions initiate the communication operation and return immediately, allowing the program to perform other work (computation or other communication).
- They return an ‘MPI_Request‘ object, which is used later to check the status (‘MPI_Test‘, ‘MPI_Testall‘) or wait for completion (‘MPI_Wait‘, ‘MPI_Waitall‘) of the operation.
- Buffers used in non-blocking operations must not be modified until the operation is confirmed complete via ‘MPI_Wait‘ or ‘MPI_Test‘.
- Non-blocking operations help avoid deadlocks and enable **overlap of computation and communication**.

Example fixing ping-pong using ‘MPI_Isend‘:

```
MPI_Request send_request;
MPI_Status status; // Or MPI_STATUS_IGNORE
int *pingpongdata; // Assume buffer is allocated and filled
int *pingrecvdata = (int*)calloc(nb_ints, sizeof(int)); // Need separate buffer

if (rank == 0) {
    // Initiate non-blocking send
    MPI_Isend(pingpongdata, nb_ints, MPI_INT, 1, 0, MPI_COMM_WORLD, &send_request
        );
    // Can perform computation here...
    // Now perform the receive (blocking)
    MPI_Recv(pingrecvdata, nb_ints, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
    // Wait for the send to complete before reusing pingpongdata or freeing it
    MPI_Wait(&send_request, &status);
    printf("Rank 0: completed ping-pong\n");
} else if (rank == 1) {
    // Perform blocking send
    MPI_Send(pingpongdata, nb_ints, MPI_INT, 0, 0, MPI_COMM_WORLD);
    // Perform blocking receive
    MPI_Recv(pingrecvdata, nb_ints, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    printf("Rank 1: completed ping-pong\n");
}
free(pingrecvdata); // Free the separate receive buffer
```

Listing 32: Non-blocking Ping-Pong

Send Modes MPI defines different send modes (‘MPI_Ssend‘, ‘MPI_Bsend‘, ‘MPI_Rsend‘, ‘MPI_Send‘) with varying synchronization semantics and buffering requirements. ‘MPI_Send‘ (standard mode) is most common, allowing the MPI library implementation flexibility.

`MPI_Sendrecv` This function performs a send and a receive in a single call, often useful for structured communication patterns (like shifts or exchanges) and avoiding deadlocks associated with separate blocking sends/receives. ‘MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)‘.

6.0.3 Collective Communication

These operations involve communication among a group of processes defined by a communicator (usually ‘MPI_COMM_WORLD‘). All processes in the communicator must call the same collective function with compatible arguments.

`MPI_Barrier` Synchronizes all processes in the communicator. No process returns from ‘MPI_Barrier‘ until all processes have called it. Note that this does not guarantee processes exit the barrier simultaneously.

`MPI_Bcast` Broadcasts data from one process (**root**) to all other processes in the communicator. ‘MPI_Bcast(buffer, count, datatype, root, comm)‘ The ‘buffer‘ contains the data on the root process before the call and contains the received data on all other processes after the call.

Example using ‘MPI_Bcast‘:

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[]) {
    int rank, size;
    int data; // Variable to hold the broadcast data
    int root_rank = 0; // Define which rank broadcasts

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == root_rank) {
        // Root rank sets the data
        data = 99;
    }

    // All ranks call Bcast. Root sends, others receive.
    MPI_Bcast(&data, 1, MPI_INT, root_rank, MPI_COMM_WORLD);

    printf("Rank %d: has data item: %d\n", rank, data);

    MPI_Finalize();
    return 0;
}
```

Listing 33: MPI_Bcast Example

`MPI_Scatter` Distributes chunks of data from the root process's send buffer to the receive buffers of all processes (including the root). ‘MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)‘

`MPI_Gather` Collects data from all processes' send buffers into the root process's receive buffer. The receive buffer on the root must be large enough to hold ‘size * recvcount‘ elements. ‘MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)‘

Example using ‘MPI_Gather‘:

```c
#include <stdio.h>
#include <stdlib.h> // For calloc, free
#include <mpi.h>
#define N 3 // Number of elements each process sends

int main(int argc, char* argv[]) {
    int rank, size;
    int data[N];
    int root_rank = 1; // Example: Rank 1 gathers
    int *recv_data = NULL; // Receive buffer only needed on root

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // All processes prepare their data to send
    for(int i=0; i<N; i++) { data[i] = rank; }

    // Root allocates space for the gathered data
    if (rank == root_rank) {
        recv_data = (int*)calloc(size * N, sizeof(int));
    }

    // All processes call Gather
    MPI_Gather(data, N, MPI_INT, recv_data, N, MPI_INT, root_rank, MPI_COMM_WORLD
        );

    // Root prints the result
    if (rank == root_rank) {
        printf("Rank %d received: ", rank);
        for(int i=0; i < size * N; i++) { printf("%d ", recv_data[i]); }
        printf("\n");
        if (recv_data != NULL) free(recv_data); // Free allocated memory
    }

    MPI_Finalize();
    return 0;
}
```

Listing 34: MPI_Gather Example

`MPI_Reduce` Combines data from all processes using a specified operation (‘MPI_Op‘, e.g., ‘MPI_SUM‘, ‘MPI_MAX‘, ‘MPI_MIN‘, ‘MPI_PROD‘, ‘MPI_LAND‘, ‘MPI_LOR‘) and places the result on the root process. ‘MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm)‘

`MPI_Allreduce` Similar to 'MPI_Reduce', but the final result is distributed to *all* processes in the communicator. 'MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm)'

Example using 'MPI_Allreduce' to find the maximum value across processes:

```c
#include <stdio.h>
#include <mpi.h>
#define N 4

int main(int argc, char* argv[]) {
    int rank, size;
    int sdata[N], rdata[N]; // Send and receive buffers
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Each process creates slightly different data
    for(int i = 0; i < N; i++) { sdata[i] = (10 - rank) * (i + 1); }

    // Find element-wise maximum across all processes
    MPI_Allreduce(sdata, rdata, N, MPI_INT, MPI_MAX, MPI_COMM_WORLD);

    // All processes now have the same max values in rdata
    // Use Barrier for synchronized output (optional, for cleaner printing)
    MPI_Barrier(MPI_COMM_WORLD);
    printf("Rank %d Allreduce Result: ", rank);
    for(int i = 0; i < N; i++) printf("%d ", rdata[i]);
    printf("\n");

    MPI_Finalize();
    return 0;
}
```

Listing 35: MPI_Allreduce Example

Other Collectives

- `MPI_Allgather`: Like 'MPI_Gather', but the result is distributed to all processes.
- `MPI_Alltoall`: Each process sends distinct data to every other process (matrix transpose pattern).
- `MPI_Scan`: Parallel prefix operation.
- `MPI_Reduce_scatter`: Combines reduction and scatter.

References

[1] Paul A. M. S. Kirkpatrick. *Parallelism Versus Concurrency*. https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/ParVConc.html. Accessed: 2025-03-26. 2021.

[2] Intel Corporation. *Intel® Xeon® Processor Scalable Family Technical Overview*. https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html. Accessed: 2025-03-26. 2017.

[3] Intel Corporation. *Intel® Performance Counter Monitor*. https://www.intel.com/content/www/us/en/developer/articles/tool/performance-counter-monitor.html. Accessed: 2025-03-26. 2023.

[4] Lawrence Livermore National Laboratory. *Introduction to Parallel Computing Tutorial*. https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial. Accessed: 2025-03-26. 2023.

[5] University of New England. *COSC330 Lecture 18: Parallel Processing*. https://turing.une.edu.au/~cosc330/lectures/display_notes.php?lecture=18. Accessed: 2025-03-26. 2023.

[6] TPoint Technologies. *Process vs Thread*. https://www.tpointtech.com/process-vs-thread. Accessed: 2025-03-26. 2024.

[7] ResearchGate. *Fonctionnement de l'Hyperthreading sur processeur Intel*. https://www.researchgate.net/figure/Fonctionnement-de-lHyperthreading-sur-processeur-Intel_fig6_278638340. Accessed: 2025-03-26. 2015.

[8] TU Wien. *Course Slides: 191.114 – Basics of Parallel Computing*. Lecture materials, TU Wien. Accessed: 2025-03-26. 2025.

[9] John L. Gustafson. "Reevaluating Amdahl's Law". In: *Communications of the ACM* 31.5 (1988). Specific details inferred; source PDF only mentions Gustafson, 1988., pp. 532–533. DOI: 10.1145/42411.42415.

[10] Source mentioned in PDF. "Amdahl vs. Gustafson diagram source". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Springer, 2011.

[11] Ananth Grama et al. *Introduction to Parallel Computing*. Second. Addison-Wesley, 2003. ISBN: 0201648652.

[12] Georgios Barlas. *Multicore and GPU Programming: An Integrated Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2014. ISBN: 9780124171404.

[13] Thomas Rauber and Gudula Rünger. *Parallel Programming - for Multicore and Cluster Systems*. 2nd. Springer, 2013. ISBN: 978-3-642-37800-3. DOI: 10.1007/978-3-642-37801-0.

[14] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Communications of the ACM* 52.4 (Apr. 2009), pp. 65–76. DOI: 10.1145/1498765.1498785.

[15] Tim Besard, Christophe Foket, and Bjorn De Sutter. "Effective Extensible Programming: Unleashing Julia on GPUs". In: *IEEE Transactions on Parallel and Distributed Systems* 30.4 (Apr. 2019), pp. 827–841. DOI: 10.1109/TPDS.2018.2872064.

[16] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.1*. June 2015. URL: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.