Notes on Parallel Computing

Nihar Shah

August 8, 2025

Contents

Chapter 1

Computer Architecture

Here, it is assumed that the reader has a basic beginner-level understanding of the C programming language.

For a gentler introduction refer to first four YouTube lectures by Prof. Mathew Jacob on High Performance Computing (HPC) NPTEL [2020].

1.1 How is Data Represented?

In a digital computer, data is represented in binary form. This means everything—from numbers to text—is stored using only two symbols: 0 and 1.

The smallest unit of data is called a **bit**. A bit can take only two values: 0 or 1.

Some commonly used units of data are:

- **bit (b)** — smallest unit of data
- **Byte (B)** — 1 Byte = 8 bits
- **Kilobyte (KB)** — 1 KB = 2^{10} Bytes = 1024 Bytes
- **Megabyte (MB)** — 1 MB = 2^{20} Bytes
- **Gigabyte (GB)** — 1 GB = 2^{30} Bytes

These units are based on powers of two because binary is the foundation of all digital computation.

For example, character data is often represented using an 8-bit code. In many systems, a 7-bit code is used to encode the character itself, and 1 extra bit is added as a **parity bit**.

The purpose of the parity bit is to help check whether the character data is correct or not. It acts as a basic form of error detection—if a bit gets flipped accidentally during storage or transmission, the system can sometimes catch the mistake using this bit.

1.1.1 Integer Data Representation

Integer data can be of two types:

- **Signed Integer** — This type of integer can represent both positive and negative values. The most significant bit (MSB) is used to indicate the sign of the number:
  - 0 for positive numbers
  - 1 for negative numbers

  Suppose $x_{n-1}x_{n-2}\ldots x_0$ is an n-bit signed integer. Then its value can be calculated using:

$$x_{n-1}x_{n-2}\ldots x_0 = (-1)^{x_{n-1}} + \sum_{i=0}^{n-2} x_i 2^i \tag{1.1}$$

  Here, x_0 is the least significant bit (LSB) and x_{n-1} is the most significant bit (MSB), which represents the sign.

  Two's Complement
  Another way to represent signed integers is using **Two's Complement**. In this method, negative numbers are represented by inverting all bits of their positive counterpart and then adding 1 to the result.

  The value of a signed integer in two's complement form is computed as:

$$x_{n-1}x_{n-2}\ldots x_0 = -x_{n-1}\cdot 2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i \tag{1.2}$$

  This representation allows the same hardware to perform both signed and unsigned arithmetic, which makes it widely used.

Example

Steps to convert a negative decimal number to its two's complement representation:

1. Convert the absolute value of the number to binary.
2. Add a 0 (or 0s) in front if needed to reach the desired number of bits.
3. Flip all 0s to 1s and 1s to 0s.
4. Add 1 to the result.

Let's take −14 as an example.

- The binary of 14 is 1110.
- To convert this to 5-bit representation, we pad a 0 in front: 01110.
- Now, flip the bits: 10001.
- Add 1: 10001 + 1 = 10010.

So, the two's complement representation of -14 in 5 bits is 10010.

Similarly, to represent -14 in 32 bits:

- Binary of 14: 1110.
- Pad with zeros: $000\ldots0001110$ (total 32 bits).
- Flip all bits: $111\ldots1110001$.
- Add 1: $111\ldots1110010$.

Thus, $111\ldots1110010$ is the two's complement representation of -14 in 32 bits. Thus, the decimal number -14_{10} using two's complement:

- In 5 bits: 10010 (i.e., $-16+2$)
- In 6 bits: 110010 (i.e., $-32+16+2$)
- In 32 bits: $111\ldots1110010$

Note that in Two's complement representation, positive numbers are represented identically to their standard binary form. For example, to represent the number 65 as a 32-bit signed integer using Two's complement, we simply write its binary form padded with leading zeros:

$$65 = 00000000\ 00000000\ 00000010\ 00000001$$

Here, the most significant bit (MSB) is 0, indicating a non-negative number. Since 65 is positive, no bit inversion or addition is needed — the representation remains the same as the unsigned binary version.

- **Unsigned Integer** — This type of integer can only represent non-negative values. It does not use any sign bit, so all bits contribute to the magnitude of the number.

  If $x_{n-1}x_{n-2}\ldots x_0$ is an n-bit unsigned integer, then its value is computed using:

$$x_{n-1}x_{n-2}\ldots x_0 = \sum_{i=0}^{n-1} x_i 2^i \tag{1.3}$$

- **Hexadecimal** — This is a base-16 number system. It uses the following 16 symbols to represent values: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F`.

  Each hexadecimal digit corresponds to a unique 4-bit binary sequence. That is why we need at least 4 bits to represent a single hexadecimal digit. For example:

$$A_{16} = 1010_2$$

Let us take another example. The decimal number 14 in 32-bit binary representation is:

$$0000\ldots00001110$$

In two's complement form (to represent -14), it becomes:

$$1111\ldots11110010$$

This binary sequence corresponds to the hexadecimal value `FFFFFFF2`. In C programming, we often prefix hexadecimal constants with `0x`, so this would be written as `0xFFFFFFF2`.

The table below shows how each hexadecimal digit maps to its binary equivalent:

Hexadecimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

- **Real Data** — Real numbers are usually represented using floating point notation. This means the decimal point can "float" to any position, allowing us to represent both very small and very large numbers efficiently.

  In contrast, **fixed point** representation reserves a fixed number of bits for the integer and fractional parts, and the decimal point is fixed in one place. For example, suppose we allocate (1 bit for sign, 8 bits for integer part and 7 bits for fractional part):

$$\underbrace{0}_{\text{sign bit}}\ \underbrace{00001101}_{\text{integer part}}\cdot\ \underbrace{0110000}_{\text{fractional part}}$$

  This represents 13.25 because $00001101_2 = 13$ and $0110000_2 = 0.25$.

  IEEE 754 Floating Point Standard: Real numbers in modern computers are represented using the IEEE 754 standard. For a 32-bit float, the layout is as follows:

- **Sign Bit (1 bit):** It is the most significant bit. It is 0 for positive numbers and 1 for negative numbers.
- **Exponent (8 bits):** These next 8 bits store the exponent. The value stored here is in *Excess-127* or *Bias-127* format, which means the actual exponent is:

$$\text{Exponent}_{\text{actual}} = \text{Exponent}_{\text{stored}} - 127$$

- **Fraction (23 bits):** These represent the fractional part of the number. The full number is stored in normalized form as $1.f$, where f is the binary fractional part.

The floating point number represented is:

$$(-1)^s \times 1.f \times 2^{e-127}$$

Steps to convert a real number to IEEE 754 format:

1. Convert the number to binary.
2. Normalize it to the form $1.f \times 2^e$.
3. Add 127 to the exponent value.
4. Determine the sign bit (0 for positive, 1 for negative).
5. Write the exponent in 8-bit binary.
6. Write the fraction after the decimal point in 23-bit binary.

Example

Let's convert -23.5_{10} to IEEE 754 format:

- Binary: -10111.1_2
- Normalized: -1.01111×2^4
- Exponent: $4 + 127 = 131_{10} = 10000011_2$
- Fraction: 01111000000000000000000 (pad 0s to make 23 bits)
- Sign bit: 1

So the 32-bit IEEE 754 representation is:

$$\boxed{1\ 10000011\ 01111000000000000000000}$$

and is `0xC1BC0000` in hexadecimal.

Steps to convert IEEE 754 to decimal:

1. Interpret the sign bit.
2. Convert the exponent bits to decimal and subtract 127.
3. Convert the fraction to decimal by treating it as $1.f$.
4. Multiply the result by $(-1)^s$ and 2^{e-127}.

Example

Let us convert the IEEE 754 number:

$$1\ 10000001\ 01100000000000000000000$$

- Sign bit: 1 (so the number is negative)
- Exponent: $129 \Rightarrow 129 - 127 = 2$
- Fraction: 1.011

So the final number is:

$$-1 \times 1.011 \times 2^2 = -101.1_2 = -5.5_{10}$$

Advantages and Limitations: Floating point allows us to represent a very large range of values. It avoids the limitation of fixed decimal positions. However, it can suffer from rounding errors and loss of precision over many operations.

Special Cases (B&O 2.4.2):

- **Zero:** All exponent and fraction bits are 0.
  - +0: Sign bit 0
  - −0: Sign bit 1
- **Infinity:** Exponent bits are all 1, fraction bits are all 0.
  - $+\infty$: Sign bit 0
  - $-\infty$: Sign bit 1

Such special cases are summarised in the following tables 1.1 and 1.2. For more details on this topics refer Rajaraman [2016].

Value	**Sign**	**Exponent (8 bits)**	**Significand (23 bits)**
+0	0	00000000	00...00 (all 23 bits 0)
−0	1	00000000	00...00 (all 23 bits 0)
$+1.f \times 2^{(e-b)}$	0	00000001 to 11111110	$a\,a\,\ldots\,a$ (where a = 0 or 1)
$-1.f \times 2^{(e-b)}$	1	00000001 to 11111110	$a\,a\,\ldots\,a$ (where a = 0 or 1)
$+\infty$	0	11111111	00...00 (all 23 bits 0)
$-\infty$	1	11111111	00...00 (all 23 bits 0)
SNaN (Signaling NaN)	0 or 1	11111111	000...01 to 011...11 (leading bit 0, at least one 1)
QNaN (Quiet NaN)	0 or 1	11111111	1000...10 (leading bit 1)
Positive subnormal $0.f \times 2^{x+1-b}$	0	00000000	000...01 to 111...11 (at least one 1)

Table 1.1: IEEE 754 Representation for Special Floating-Point Values (32-bit)

Operation	**Result**
$n/\pm\infty$	0
$\pm 0/\pm 0$	NaN
$\pm\infty/\pm\infty$	$\pm\infty$
$\infty - \infty$	NaN
$\pm n/0$	$\pm\infty$
$\pm\infty/\pm\infty$	NaN
$\infty + \infty$	∞
$\pm\infty \times 0$	NaN

Table 1.2: Results of Floating Point Operations Involving Infinity and Zero

Note that table 1.1 can be similarly extended for 64-bit representations. Here's a fun question to think about: What is the largest number that can be represented in 32-bit IEEE 754 format (excluding special values like $+\infty$)?
A natural answer might be: all 1s in the exponent bits and all 1s in the significand (fraction) bits. However, this combination represents a NaN (Not a Number), since according to the IEEE 754 standard, an exponent of all 1s and a non-zero significand encodes a NaN.
Therefore, the largest representable finite number in 32-bit IEEE 754 format corresponds to an exponent of 11111110 (i.e., 254 in decimal), and a significand of all 1s (i.e., the largest fraction just below 1). This gives the largest normal floating-point number just before $+\infty$.
Similarly, for the smallest positive number (magnitude-wise) in 64-bit representation, it is not simply all zeros, since that would correspond to $+0$ (or -0 if the sign bit is 1). There are two interpretations of the "smallest number":

- The smallest **positive** subnormal number is represented by: sign bit = 0, exponent = all 0s, and fraction = $000\ldots01$. This is the smallest positive number greater than zero.
- The smallest **(most negative) representable number** is given by: sign bit = 1, exponent = $111\ldots10$, and fraction = all 1s. This is the largest-magnitude negative number (i.e., most negative finite number).

Handling Overflow in Arithmetic:

Sometimes, the result of a floating point operation might not fit in the 23-bit significand. In such cases, we have two options:

- **Truncation:** Discard extra bits.
- **Rounding:**
  - Rounding upwards: For example, if significand is $0.110\ldots01$ and overflow is by 1, then the significand after rounding becomes $0.110\ldots10$, i.e. we add 1 to the LSB.
  - Rounding downwards: The extra bits are ignored.

IEEE 754 for 64-bit (Double Precision):

For higher precision, we use 64-bit representation:

- 1 bit for sign
- 11 bits for exponent
- 52 bits for fraction

1.1.2 Variables and Data Allocation of Different Lifetimes

Understanding these memory regions and their respective allocation strategies is fundamental to writing safe and efficient programs, particularly in low-level languages like C and C++. Mistakes such as forgetting to free heap memory, or accessing memory after it has been deallocated, are common sources of bugs and security vulnerabilities. In a computer program, different variables have different lifetimes, depending on where and how they are declared and allocated. These lifetimes determine how long a variable remains in memory during the program's execution. Broadly, there are three primary types of data allocation:

- **Static Allocation:** Variables that are statically allocated exist for the entire duration of the program. Their lifetime spans from the beginning to the end of program execution. This includes global variables, as well as static local variables declared with the `static` keyword in C. These variables are allocated in the data segment of the memory, which is part of the executable file. Since their memory is reserved at compile-time, the size and type of statically allocated data must be known in advance. Both initialized and uninitialized static variables are handled this way:
  - Initialized static data is stored in the '.data' segment.
  - Uninitialized static data (commonly zero-initialized) is stored in the '.bss' segment.
- **Heap Allocation:** Heap allocation is used for dynamically allocated data, whose size may not be known at compile-time. This type of allocation allows a program to request and release memory at runtime using functions such as `malloc`, `calloc`, or `realloc` in C. The memory must be explicitly freed using `free` when it is no longer needed, to avoid memory leaks. The lifetime of a heap-allocated variable begins when it is allocated and ends when it is explicitly deallocated. All such allocations happen in the memory region known as the heap. This is particularly useful when dealing with:
  - Variable-sized data structures (e.g., linked lists, trees, graphs)
  - Data that must persist across function calls
- **Stack Allocation:** Stack allocation is used for variables that are declared inside functions, including function parameters and local variables. These variables are allocated on the call stack, a special region of memory used to manage function calls and returns. The lifetime of a stack-allocated variable is confined to the duration of the function in which it is declared. That is, memory for such variables is allocated when the function is invoked and deallocated when the function returns. This makes stack allocation highly efficient, as it simply involves moving the stack pointer up and down. However, stack variables are not accessible once the function ends, and returning pointers to such variables leads to undefined behavior.

For example, when a program begins execution, its memory is organized into distinct regions, as shown in Figure 1.1.

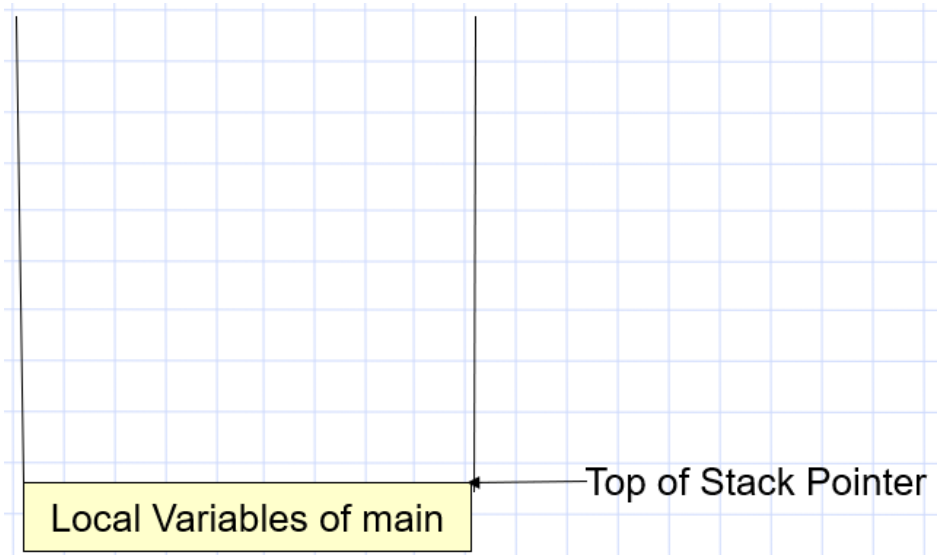


Figure 1.1: Memory Layout of a Program

The stack pointer initially points to the top of the stack, which holds the local variables of the `main()` function. The stack grows downward (towards lower memory addresses), and this region of memory is used to manage function calls and their associated data.

Now suppose the `main()` function makes a call to another function, say `func1()`. In that case, the system allocates additional space on the stack for the function call. This includes space for:

- The return address (i.e., the point in `main()` to resume execution after `func1()` finishes),
- Any arguments passed to `func1()`,
- The local variables declared inside `func1()`.

This updated stack state is illustrated in Figure 1.2. The stack pointer is now moved to point to the new top of the stack, corresponding to the local variables of `func1()`.

When `func1()` completes its execution and returns, the stack frame associated with it is deallocated. This involves:

- Removing the local variables of `func1()`,
- Restoring the return address,
- Adjusting the stack pointer back to its state before the call.

As a result, the stack pointer once again points to the top of the stack containing the local variables of `main()`, restoring the memory layout to its original state shown in Figure 1.1. This mechanism of pushing function call data onto the stack and popping it off during function return is fundamental to how most programming languages, including C, manage nested and recursive function calls. It also underpins the concept of function call stacks and stack-based memory management.

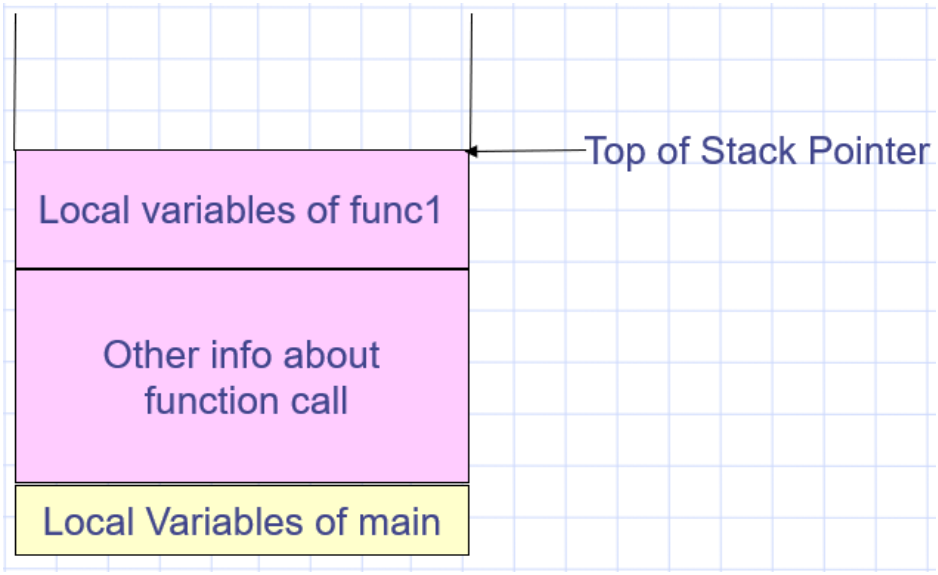


Figure 1.2: Memory Layout after a Function Call to `func1()`

1.1.3 Program and Data

When a program is compiled and executed, its memory layout is broadly divided into several segments, each serving a specific purpose. These include the code segment, data segments (initialized and uninitialized), heap, and stack.

The code is stored in the **text segment**, which contains the compiled machine instructions of the program. This segment is typically read-only and does not change during execution.

The **data segment** holds global and static variables. It is further divided into:

- **Initialized Data Segment:** Contains global and static variables that are initialized by the programmer.
- **Uninitialized Data Segment (also called BSS):** Contains global and static variables that are declared but not explicitly initialized.

Both the code and data segments are fixed in size once the program starts running.

In contrast, two other regions of memory — the **heap** and the **stack** — are dynamic in nature:

- The **heap** is used for dynamically allocated memory (e.g., using `malloc()` in C). The heap grows upwards as new memory is allocated during program execution.
- The **stack** is used for function call information, such as local variables, parameters, and return addresses. It grows downwards and shrinks as functions are called and returned.

This division and organization of memory during program execution is beautifully depicted in Figure 1.3.

A program utilizes the main memory in a structured way, with different sections allocated for different types of data and instructions.

For instance, an instruction such as `add x, y, z` would be stored in the **text segment** (also referred to as the code segment) of the memory. This segment contains

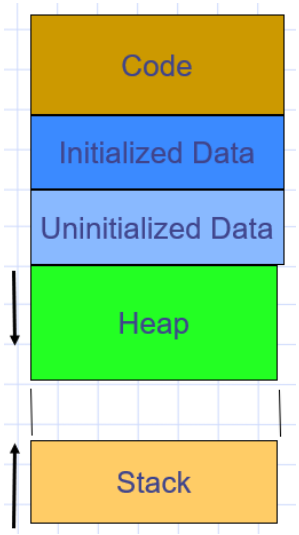


Figure 1.3: Program and Data Memory Layout

the compiled machine instructions that the CPU executes. It is typically read-only to prevent accidental overwriting of program instructions during execution.

The variables used in a program, on the other hand, are allocated in different memory segments depending on their lifetime and declaration style:

- An integer variable like `int x`, if declared within a function, is likely to be **stack-allocated**. This means that memory for `x` is reserved on the call stack, and its lifetime is limited to the duration of the function's execution.
- A variable like `double w`, if allocated dynamically using functions such as `malloc()` or `calloc()`, is stored in the **heap segment**. The programmer must manage the lifetime of such variables explicitly using memory allocation and deallocation functions.
- A variable such as `float t`, if declared globally or as a static local variable, is stored in the **data segment**. This data segment can be further divided into:
  - The **initialized data segment**, where variables with an explicit initial value reside.
  - The **uninitialized data segment** (often called the BSS segment), which holds variables declared but not explicitly initialized.

This organization of memory usage is depicted in Figure 1.4.

1.2 Basic Computer Organization

A computer system is broadly organized into three major components, as illustrated in Figure 1.5:

- **Central Processing Unit (CPU) or Processor:** Often referred to as the brain of the computer, the CPU is responsible for executing instructions and performing all arithmetic and logical operations.

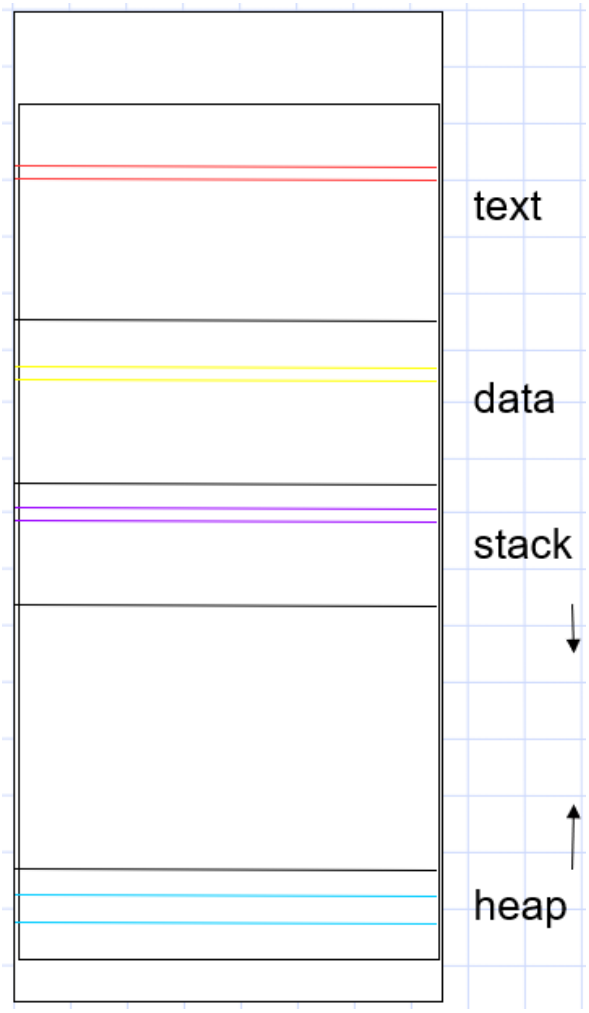


Figure 1.4: Program Memory Layout

- **Main Memory:** This component is used to store both data and instructions temporarily during program execution. Memory is essential for the CPU to retrieve and process data efficiently.

- **Input/Output Devices (I/O Devices):** These devices enable the computer to interact with the external environment. Examples include keyboards, mice, monitors, and printers. Input devices send data to the computer, while output devices display or produce results.

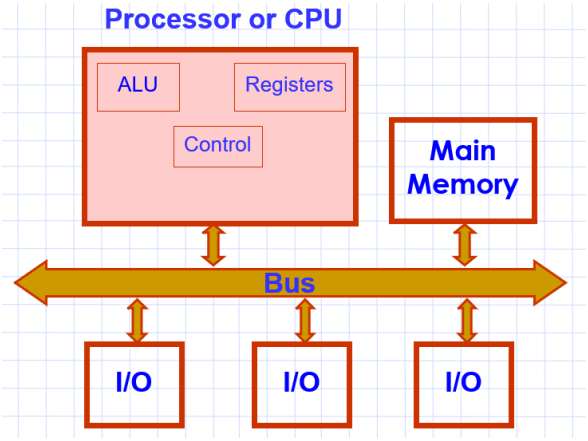


Figure 1.5: Basic Computer Organization

The CPU itself is further divided into the following core components:

- **Control Unit (CU):** The control unit manages and coordinates the operations of the CPU. It directs how data moves between the CPU, memory, and I/O devices. It decodes instructions and ensures that the correct sequence of operations is followed.
- **Arithmetic Logic Unit (ALU):** This unit performs all arithmetic operations (like addition, subtraction) and logical operations (like AND, OR, NOT). It acts as the computational engine of the CPU.
- **Registers:** Registers are small, high-speed storage locations built directly into the CPU. They hold temporary data and intermediate results while instructions are being executed. Registers play a key role in speeding up the processing as accessing them is significantly faster than accessing main memory.

  Registers are generally classified into two categories:

  - **General Purpose Registers (GPRs):** These registers are available to the programmer for storing temporary values, such as operands or intermediate results. They help in reducing memory accesses. However, CPUs typically have a limited number of GPRs. One reason for using them is that there exists a large speed disparity between the CPU and main memory. For example:
    - CPU operates at around 2 GHz, which corresponds to about 0.5 nanoseconds per cycle.
    - Main memory accesses typically take 50–100 nanoseconds.

    Because of this gap, using registers instead of repeatedly accessing memory helps improve performance.
  - **Special Purpose Registers (SPRs):** These are reserved for specific control functions within the CPU. An important example is the **Program Counter (PC)**, which holds the address of the next instruction to be executed. Other SPRs may include status registers, instruction registers, and stack pointers depending on the architecture.

1.2.1 Main Memory and its problems

One of the primary performance bottlenecks in modern computer systems arises due to the significant speed gap between the CPU and the main memory. While CPUs operate at very high frequencies (in the order of gigahertz), main memory (typically DRAM) is much slower—approximately 100× slower. As a result, even though the CPU is capable of executing instructions rapidly, it often ends up idling while waiting for data to be fetched from the main memory.

Another related limitation is the scarcity of CPU registers. These registers are extremely fast and are used to store immediate data required for computations. However, their number is very limited, which means most data must be retrieved from main memory, further aggravating the memory bottleneck.

Example

Let's consider a toy example to understand the main memory organization where the main memory has a size of 128 Bytes, and it is divided into blocks of size 4 Bytes. Since the memory is byte-addressable, each of the 128 Bytes has a unique memory address. To uniquely represent each of these 128 addresses, we require:

$$\log_2 128 = 7 \text{ bits}$$

Hence, memory addresses will range from 0000000_2 to 1111111_2.
The total number of memory blocks is:

$$\frac{128}{4} = 32 \text{ blocks}$$

Thus the number of bits that are used to identify the block number (since there are 32 blocks):

$$\log_2 32 = 5 \text{ bits}$$

Each block contains 4 Bytes, which requires 2 bits to specify the offset within the block:

$$\log_2 4 = 2 \text{ bits}$$

Therefore, any 7-bit memory address can be broken down into:

$$\underbrace{b_6 b_5 b_4 b_3 b_2}_{\text{Block address (5 bits)}} \quad \underbrace{b_1 b_0}_{\text{Offset (2 bits)}}$$

- **Block address (5 bits):** Specifies which of the 32 blocks contains the address.
- **Offset (2 bits):** Specifies which byte within the block is being addressed.

This layout allows the system to efficiently locate a specific byte in memory by first identifying the block and then using the offset to access the byte within that block as shown in figure 1.6.

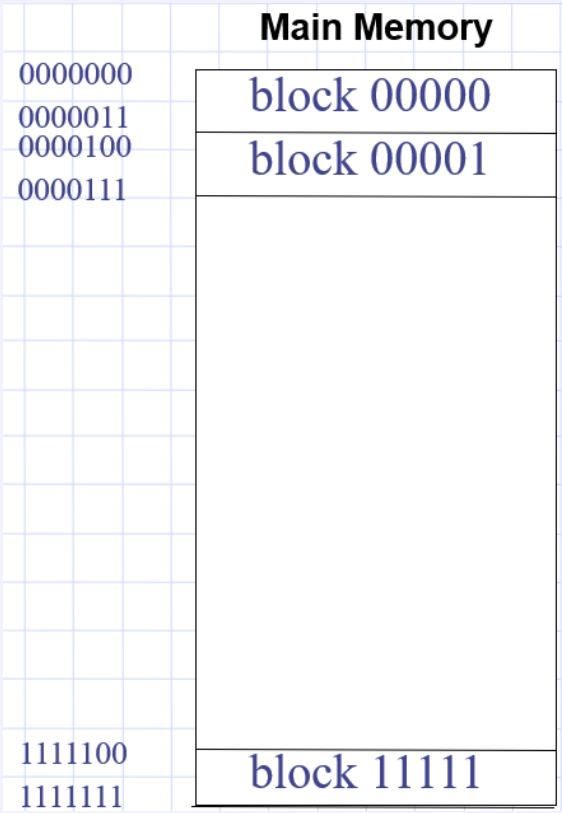


Figure 1.6: Memory Address Breakdown

1.2.2 Cache Memory and Locality of Reference

The solution to this problem is the introduction of cache memory. Cache is a small, high-speed memory that resides very close to or within the CPU itself. It acts as a buffer between the CPU and the main memory, storing frequently accessed data and instructions. If the data the CPU needs is found in the cache (called a *cache hit*), it can be accessed much more quickly than from main memory. Otherwise, a *cache miss* results in fetching data from the slower main memory.

The design of cache memory relies heavily on the **principle of locality of reference**, which exploits patterns in how programs access memory. There are two major types of locality:

- **Temporal Locality:** This principle states that memory locations that have been accessed recently are least likely to be accessed again in the near future. For example, if a program accesses a variable, it is likely to access the same variable again soon. This is often seen in loops or repeated function calls where the same data is used multiple times.

- **Spatial Locality:** This principle states that memory locations near those that have been recently accessed are likely to be accessed soon. For instance, when iterating over an array, accessing one element implies that its neighboring elements will be accessed shortly.

By leveraging these two types of locality, cache memory significantly reduces the

average memory access time and helps ensure that the CPU is not left waiting idly for data.

1.3 Cache Composition

Main memory is much larger than the cache. Hence, it is not possible to keep the entire memory content in the cache at once. So, when the CPU accesses a memory location, a portion of the memory which is a **memory block** is temporarily copied into the cache.

When the CPU needs to access a data item or an instruction, it refers to a specific memory address. A memory address is divided into three fields:

- **Tag:** Used to uniquely identify a block of memory among many possible candidates that could reside in the same cache set.
- **Index:** Identifies the specific cache set to which the memory block maps.
- **Offset:** Specifies the exact word or byte within the block.

It's schematic is as shown in the figure 1.7 .

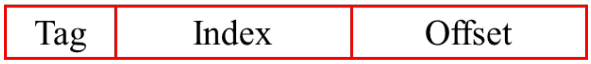


Figure 1.7: Memory Address Breakdown

A cache can be broadly divided into two parts -cache directory which keep track of the address and information about whether the data is valid or not and the cache ram which actually stores the data. The cache is composed of multiple **sets**, where each set contains one or more **cache lines** (also called *slots*). A cache line typically includes the following components:

- **Tag:** A portion of the memory address used to verify if the data stored in the cache line corresponds to the requested address.
- **Valid Bit:** Indicates whether the data in the cache line is valid (i.e., whether it contains up-to-date information from main memory).
- **Data Block:** The actual data fetched from main memory.

It's schematic is as shown in the figure 1.8

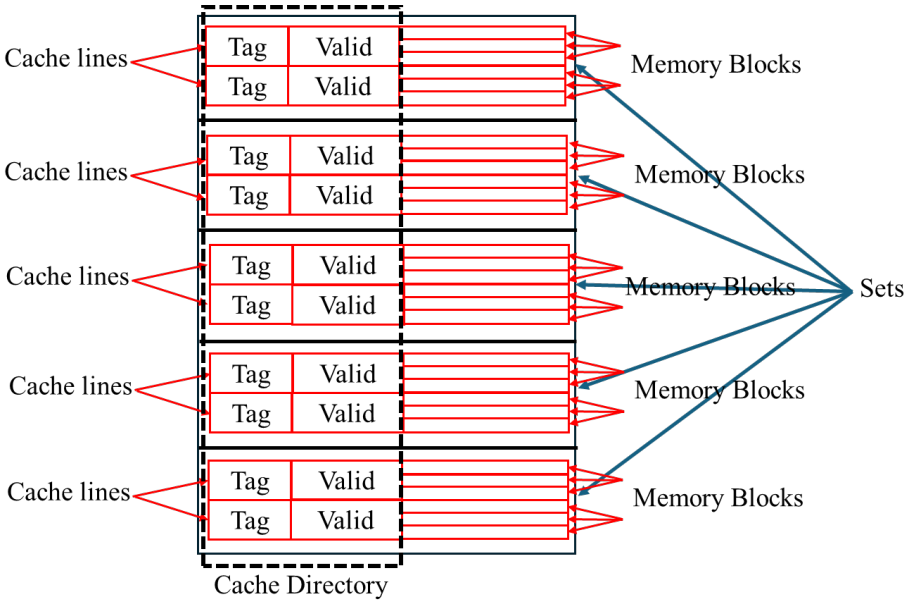


Figure 1.8: Cache Memory

Main memory is organized into blocks or words, and cache memory is used to store a subset of these blocks for faster access. The cache is divided into small storage units called **cache lines** (or slots). Each cache line can hold exactly one block of data from the main memory. The process of deciding which cache line should hold which memory block is called **mapping**. Thus, "mapping" refers to the rule that assigns a memory block to a particular cache line based on the memory address.

When a memory access occurs, the cache operates as follows:

1. The **index** field is used to locate the appropriate cache set.
2. Within the identified set, each cache line's tag is compared to the **tag** field of the memory address.
3. If the tag matches and the **valid bit** is set, it results in a **cache hit**, and the data is fetched using the offset.
4. If the tag does not match any line in the set, it results in a **cache miss**, and the data must be fetched from main memory.

This is as illustrated in figure 1.9

Example

Suppose a memory address is 16 bits long and the cache has 64 lines. Then, we need $\log_2 64 = 6$ bits to index the cache lines. These 6 bits are extracted from the memory address and used to determine which cache line the block maps to. So, memory blocks whose addresses share the same 6 index bits will always map to the same cache line.

The structure and number of sets and lines determine the type of cache organization (e.g., direct-mapped, set-associative, or fully associative, or even m-way set-associative), which will be discussed in subsequent sections.

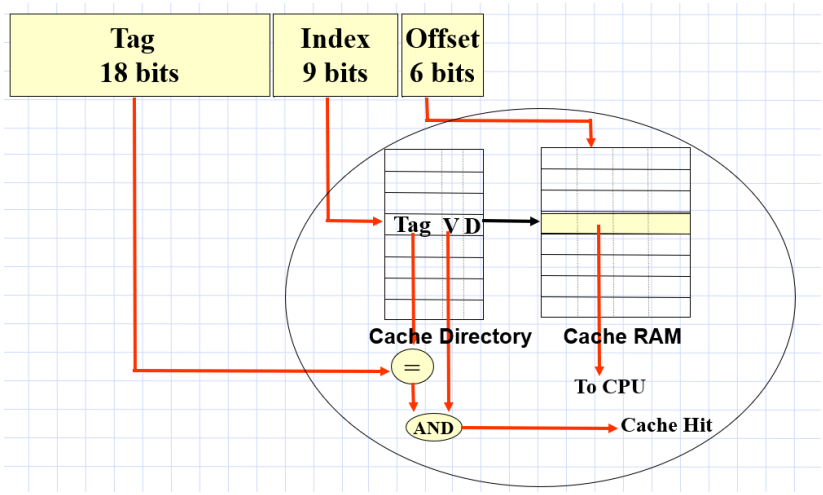


Figure 1.9: Cache Lookup and Access

1.3.1 Direct-Mapped Cache

In a **direct-mapped cache**, each cache set contains exactly **one cache line**.

Cache Access Mechanism

When the processor needs to access a memory word (instruction or data), it sends the corresponding memory address to the cache. The following steps occur:

1. **Indexing:** The index bits of the memory address are used to determine the set (or cache line, in the case of direct-mapped cache) where the memory block might reside. This can be done very fast since the index bits refer to an address which can be located very easily.

2. **Tag Comparison:** The remaining significant bits bits of the memory address, called the *tag*, are compared with the tag stored in the selected cache line. Since there is only one cache line in each set, the index bits map to the cache lines directly.

3. **Valid Bit Check:** Along with the tag, each cache line stores a valid bit indicating whether the data in the cache line is meaningful. If the valid bit is set and the tags match, a **cache hit** occurs.

4. **Cache Hit:** If there is a match and the valid bit is set, the data is fetched directly from the cache, which is very fast compared to main memory access.

5. **Cache Miss:** If the valid bit is not set or the tags do not match, a **cache miss** occurs. In this case:

   - The processor fetches the required memory block from the main memory.
   - In direct-mapped cache since each cache set contains exactly one cache line. This means that every memory block can map to exactly *one* cache line in the cache. The specific line is determined using the *index bits* of the memory address. These index bits identify which cache set the memory block corresponds to. Thus, a given memory block from main memory

always maps to a single fixed location in the cache. If two memory blocks happen to have the same index bits, they will compete for the same cache line. When one is loaded, the other is evicted. This situation is known as a **conflict miss** and is a known limitation of direct-mapped caches. Thus, the fetched block is stored in the cache line corresponding to the index, replacing the existing block if any.

- The tag and valid bit of the cache line are updated.
- The processor resumes execution with the now-available data.

Advantages of Direct-Mapped Cache:

- Very simple to implement in hardware.
- Fast lookup due to one-to-one mapping.
- Less hardware cost compared to other mapping schemes.

Disadvantages of Direct-Mapped Cache:

- High chance of conflict misses. Two memory blocks in main memory that map to the same cache line will replace each other frequently.
- Poor utilization of cache space if many blocks compete for the same line.

Direct-mapped caches are efficient and fast, but the rigid mapping can lead to performance issues if many memory blocks map to the same cache line.

1.3.2 Fully Associative Cache

In a **fully associative cache**, any memory block can be stored in any cache line. It can be thought as only one cache set having all the cache lines. There is no fixed mapping between memory addresses and cache lines. So, there are no index bits in the address.

Cache Access Mechanism

When the processor needs to access a data word or an instruction, it sends the memory address to the cache. The cache performs the following steps:

1. **Tag Comparison – Associative Search:** The memory address is divided into two parts: the **tag** and the **block offset**. Since any block can go into any cache line, there is no need for index bits. The cache controller compares the tag of the incoming address with the tags of all cache lines in parallel. This is called an *associative search*.
2. **Valid Bit Check:** Each cache line has a valid bit. It tells whether the content of the cache line is meaningful. If the valid bit is set and the tag matches, we have a *cache hit*.

3. **Cache Hit:** If there is a tag match and the valid bit is set, the required data is present in the cache. The data is directly sent to the processor. This access is much faster than going to the main memory.

4. **Cache Miss:** If there is no matching tag or the valid bit is not set, it results in a *cache miss*. In this case:

   - The required memory block is fetched from the main memory.
   - The block is then placed into an available cache line.
   - If all lines are full, one of the existing blocks is replaced using a *replacement policy*. Common policies include:
     - **FIFO (First-In-First-Out)**: Replace the oldest block.
     - **LRU (Least Recently Used)**: Replace the block that was not used for the longest time.
     - **Random**: Replace a randomly chosen block.
   - The tag and valid bit of the chosen cache line are updated.
   - The processor resumes execution using the newly fetched data.

Advantages of Fully Associative Cache

- Very flexible. Any memory block can go into any cache line.
- Conflict misses are minimized, as blocks are not restricted to a fixed location.

Disadvantages of Fully Associative Cache

- Hardware is more complex. The cache must compare the incoming tag with all tags in parallel.
- Associative search uses more power and is slower than simple indexing.
- Cache access time is usually longer compared to a direct-mapped cache.

1.3.3 Set Associative Cache

Set associative mapping is a hybrid between direct-mapped and fully associative caches. In this approach, the cache is divided into multiple **sets**. A m-way set associative means each set contains $m(> 1)$ number of cache lines, also called *ways*. A block of main memory maps to exactly one set, but it can go into any line within that set.

For example, consider a 4-way set-associative cache with 128 total cache lines. This means we have:

$$\frac{128}{4} = 32 \text{ sets}$$

Each set has 4 lines. A memory block is first mapped to a set using a portion of its address — typically using a modulo operation as shown in the figure 1.10 for a 2-way set-associative. Then, the block can be stored in any of the 4 lines within that set.

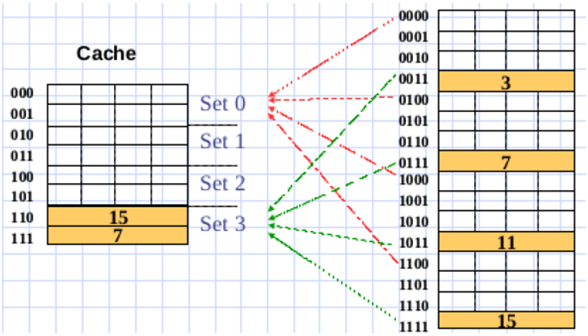


Figure 1.10: Set Associative Mapping

Cache Access Mechanism

When the processor needs to access a data word or an instruction, it sends the memory address to the cache. The following steps occur:

1. **Indexing:** The index bits from the memory address determine which set to look into. This is a fast and simple operation. It is similar to the indexing used in direct-mapped caches.

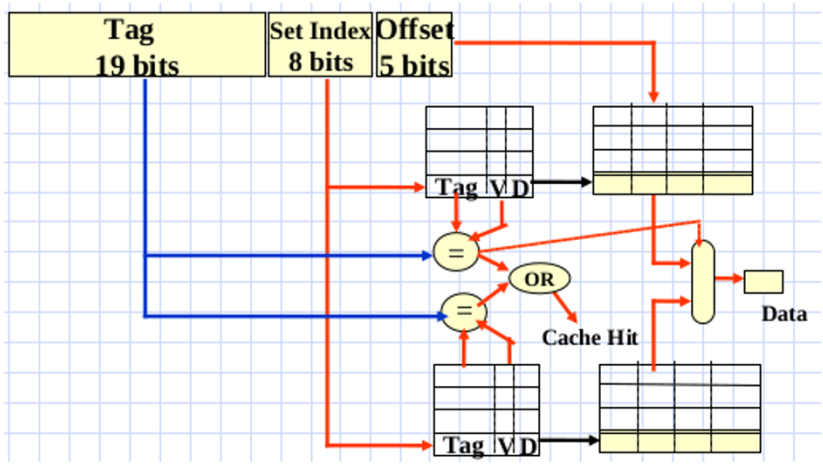
2. **Tag Comparison (Associative Search within Set):** The remaining significant bits of the address form the **tag**. The cache controller compares this tag with the tags of all cache lines inside the selected set. All lines in the set are checked in parallel.

3. **Valid Bit Check:** Each line in the set stores a valid bit. If the valid bit is set and the tag matches, the cache reports a **hit**.

4. **Cache Hit:** If a match is found, the data is fetched directly from the cache. This is much faster than accessing the main memory.

5. **Cache Miss:** If no matching tag is found or the valid bit is not set, a **miss** occurs. In that case:

   - The memory block is fetched from the main memory.
   - The block is placed into the corresponding set.
   - If there is space (a line with invalid bit), the block is placed there.
   - If all lines are occupied, one block is replaced using a replacement policy like:
     - **FIFO (First-In-First-Out)**: Replace the oldest block.
     - **LRU (Least Recently Used)**: Replace the least recently accessed block.
     - **Random**: Replace a randomly chosen block.
   - The tag and valid bit are updated, and the processor resumes using the fetched data.



Advantages

- Reduces conflict misses compared to direct-mapped caches.
- More flexible than direct-mapped, but less complex than fully associative.
- Provides a good balance between speed, flexibility, and hardware cost.

Disadvantages

- Slightly slower than direct-mapped due to the parallel tag comparisons within each set.
- Requires more complex hardware than direct-mapped caches.
- Needs a replacement policy to decide which block to evict on a miss.

Example

Consider an example of an 8 GB main memory (RAM). Let the size of each memory block be 64 bytes and consider a typical cache size of 64 KB, and assume a **2-way set-associative** mapping.
Since 8 GB $\approx 2^{33}$ bytes, we need 33 bits to uniquely identify each byte in memory (because $\log_2 2^{33} = 33$). Since the size of each memory block is 2^{64} bytes, which is 2^6 bytes. Therefore, the total number of memory blocks in main memory is:

$$\frac{2^{33}}{2^6} = 2^{27}$$

So, there are approximately 125×10^6 memory blocks.
To address these blocks, we need 27 bits (since $\log_2 2^{27} = 27$). The remaining 6 bits are used to specify the offset (i.e., the byte position) within a block.
Thus, each 33-bit memory address can be divided as follows:

- 27 bits for the **block number**
- 6 bits for the **offset** within the block

Since each memory block is 64 bytes, each cache block is also 64 bytes. In a 2-way set-associative cache, each **set** contains 2 blocks, meaning each set occupies $2 \times 64 = 128$ bytes.

So, the total number of sets in the cache is:

$$\frac{64 \text{ KB}}{128 \text{ bytes}} = \frac{2^{16}}{2^7} = 2^9 = 512 \text{ sets}$$

Hence, we require $\log_2 512 = 9$ bits to index the cache sets.
Now recall that we need 27 bits to address a memory block (from earlier). In set-associative mapping, the **least significant** 9 bits of the block number are used as the **index** into the cache. The **remaining** 18 bits serve as the **tag**, which is stored in the tag directory to check if a block is present in a set.
Finally, we still need the 6 offset bits to identify the specific byte within the block. Therefore, the 33-bit memory address is divided into:

- **18 tag bits** (for block identification)
- **9 index bits** (for set selection)
- **6 offset bits** (for byte within block)

When the processor wants to access some data or an instruction, it sends a 33-bit memory address. This address is first searched in the cache. This also depends on the type of mapping associated with cache.
Since we are using a 2-way set associative cache, each cache set has 2 cache lines. The cache uses some bits of the memory address sent by the processor (called **index bits**) to find the correct set. Once the set is located, the cache compares the **tag bits** of the address with the tags stored in the two cache lines of that set. If there is a match, and the **valid bit** is set, it is a **cache hit**. This means the data is already in the cache and is sent to the processor. If there is no match, or the valid bit is not set, it is a **cache miss**. The data must then be fetched from the main memory and placed into one of the cache lines in that set.
In the case of a **cache miss**, the data or instruction requested by the processor must be fetched from the main memory and placed into the cache. Since the size of cache memory is limited, an existing memory block currently in the cache must be evicted to make room for the newly fetched block. The strategy used to determine which cache line gets replaced is called the **cache replacement policy**. In an 2-way **set-associative cache**, the memory block is mapped to a cache set based on the index bits, and it can replace any of the 2 cache lines within that set. Again, policies like FIFO or LRU are used to decide which of the 2 lines in the set will be evicted and replaced by the new block. This design offers a trade-off between the high flexibility of a fully associative cache and the simple implementation of a direct-mapped cache.

Example

The table below gives the parameters for a number of different caches. For each cache, determine S, t, s, and b.

Cache	m	C	B	E	S	t	s	b
1	32	1024	4	256	1	30	0	2
2	32	1024	8	128	1	29	0	3
3	32	1024	32	32	1	27	0	5
4	32	1024	32	2	16	23	4	5

Explanation for Cache 1:
Given:

- $m = 32$, $C = 1024$ bytes, $B = 4$ bytes, $E = 256$

Then:

- Total blocks = $\frac{C}{B} = \frac{1024}{4} = 256$
- Number of sets $S = \frac{C}{B \cdot E} = \frac{256}{256} = 1$
- Block offset bits $b = \log_2 B = \log_2 4 = 2$
- Set index bits $s = \log_2 S = \log_2 1 = 0$
- Tag bits $t = m - s - b = 32 - 0 - 2 = 30$

The same formulas can be used for the other entries.

Note that in this section, we have not gone into the nitty-gritty details of how cache works internally or how it is managed by the system. We have skipped several important topics such as write strategies (like write-through and write-back), cache coherence in multi-core systems, and the structure of real-world cache hierarchies (like L1, L2, and L3 caches). Our goal here was to give a high-level understanding of what a cache is and how it helps speed up memory access. We will revisit some of these advanced topics later in the book if needed. For a deeper explanation, refer to the book Bryant and O'Hallaron [2016].

1.4 Cache and Programming

In this section, we will learn how cache-related performance issues can affect important parts of our programs. We will also look at some simple examples to understand this better.

For simplicity, we will focus only on the **data cache**, assuming that instruction and data caches are separate. Let the memory address be of 32 bits. The configuration of the data cache we will be working with is as follows:

- Cache type: Direct-mapped
- Cache size: 16 KB

- Block size: 32 Bytes

Since the cache is direct-mapped with a total size of 16 KB and a block size of 32 Bytes, it contains:

- $\frac{16\times1024}{32} = 512$ cache lines (or sets because of direct-mapped cache)
- Thus, $\log_2 512 = 9$ index bits,
- and $\log_2 32 = 5$ block offset bits
- The remaining 18 bits will be the tag bits.

1.4.1 Example 1: Vector Sum Reduction

We are interested in computing a scalar value that reduces a vector by summing all its elements. This requires a loop that adds each element of the vector to an accumulator.

```
1 double A[2048], sum = 0.0;
2 for (i = 0; i < 2048; i++) sum = sum + A[i];
```

Listing 1.1: Vector Sum Reduction

In this example, the array `A` contains 2048 double-precision values, with each value occupying 8 bytes (64-bit format). The variable `sum` is also of type `double`, and thus occupies 8 bytes.

The loop iterates through all the elements of the array, adding each value of `A[i]` to the `sum` variable. This is a simple example of a vector reduction operation, often used in numerical computations.

To analyze cache behavior, we must examine the program at a level close to machine code to identify memory load and store operations. Specifically, we want to determine which memory accesses occur and in what order, given our data cache configuration—a direct-mapped 16KB cache with 32-byte block size.

For simplicity, we assume that the variables `i` and `sum` reside in registers. This is a reasonable assumption, as we can ensure that these variables are kept in registers by the compiler or through manual optimization. As a result, they do not involve memory accesses and therefore do not impact cache performance.

Hence, we will focus only on memory accesses to the array `A`, since it is the only data structure being read from memory in the loop. Therefore, analyzing the vector sum reduces to understanding what happens when accessing `A[0]`, `A[1]`, ..., up to `A[2047]`. We will now examine how this sequential access translates into memory operations:

- Each element `A[i]` (for $i = 0$ to 2047) must be loaded from memory into a CPU register. Here, a **load** refers to an instruction that copies data from main memory into a register. Once in a register, the value can be added to the `sum` variable.
- To proceed with the analysis, we assume that the base address of the array `A` (i.e., the address of `A[0]`) is `0xA000`, which corresponds to the binary address `1010100000000000`, with all more significant bits set to zero.

- Since our cache is 16KB (i.e., 2^{14} bytes) with a block size of 32 bytes (i.e., 2^5), the cache is divided into $2^9 = 512$ blocks. So, we need 9 index bits and 5 offset bits. We'll analyze cache behavior by tracking how these 14 bits (9 index + 5 offset) change as the array is accessed sequentially.
- The index bits for `A[0]` are `100000000` (which is decimal 256). So `A[0]` will map to cache index 256.
- Since the array elements are of type `double`, each occupies 8 bytes. A 32-byte block can thus hold 4 consecutive elements: `A[0]`, `A[1]`, `A[2]`, and `A[3]`.
- So, when there's a cache miss for `A[0]`, the entire block containing `A[0]` through `A[3]` is brought into the cache. That means subsequent accesses to `A[1]`, `A[2]`, and `A[3]` will be cache hits.
- The next miss happens at `A[4]`, which maps to index 257. But again, a full block containing `A[4]` to `A[7]` is loaded. So `A[5]`, `A[6]`, and `A[7]` will hit in cache.
- This pattern continues — every 4th access is a miss (cold miss), and the next three accesses are hits due to spatial locality.

We now analyze this behavior assuming the cache is initially empty, which is also known as a **cold start**. The first access to each block (like `A[0]`, `A[4]`, `A[8]`, etc.) causes a cache miss, but the next three accesses within that block are cache hits.

So, in total, there are 2048/4 = 512 cache misses (one for every block of 4), and 2048 − 512 = 1536 cache hits. This gives us a **hit ratio** of:

$$\frac{1536}{2048} = 0.75 \quad \text{or} \quad 75\%$$

This performance gain comes entirely from **spatial locality** — because we're accessing data laid out consecutively in memory.

Now suppose we add a loop before the reduction to preload all relevant memory blocks:

```
1 for (i = 0; i < 2048; i++) tmp = A[i]; // preload
2 double A[2048], sum = 0.0;
3 for (i = 0; i < 2048; i++) sum = sum + A[i];
```

Listing 1.2: Preloading the Array for Temporal Locality

With this change, the second loop sees a **100% hit ratio**. This is because:

- 75% of hits come from **spatial locality** (within each block), and
- 25% of hits come from **temporal locality**, since the data was already loaded in the first loop and hasn't been evicted.

Suppose now the array is defined as `double A[4096]`. Will this make any difference? Consider the case where the loop is preceded by another loop that accesses all array elements in sequential order. The total memory required to store the array is $8 \times 4096 = 32$ KB, while the cache memory is only 16 KB. Hence, the entire array cannot fit into the cache. After execution of the previous loop, the second half of the

array will be in the cache and because of the spatial locality of reference it is anyways not going to be used for vector sum. This is just wasting time due to unnecessary memory accesses to no advantage. As a result, our loop will still experience cache misses, just as we analyzed earlier.

Recall that to estimate the data cache hit rate, we made the following assumptions:

- The variables `sum` and `i` are stored in registers; hence, we ignore their memory accesses.
- The base address of `A[0]` is assumed to be `0xA000`.
- We consider that only load/store instructions access memory operands; all other instructions use register operands.

1.4.2 Example 2: Vector Dot Product

In this case, we are interested in computing the dot product (also called the inner product) between two vectors, A and B, represented by the arrays `A` and `B`. The dot product is computed by taking the running sum of the products of corresponding elements from the two arrays.

As in the previous example, we will assume similar conditions and ignore the accesses to the loop index variable `i` and the variable `sum`, assuming they are stored in registers. Also, we will assume that only load/store instructions access memory operands; all other instructions use register operands.

```
1 double A[2048], B[2048], sum = 0.0;
2 for (i = 0; i < 2048; i++)
3     sum = sum + A[i] * B[i];
```

Listing 1.3: Vector Dot Product

We are interested in analyzing the order of reference sequence, which will be: `load A[0]`, `load B[0]`, and so on, till `load A[2047]`, `load B[2047]`.

We now assume the base addresses of arrays A and B as $0xA000$ and $0xE000$ respectively. Since these arrays are declared as `double`, each element occupies 8 bytes, and therefore, 4 consecutive elements of the array will fit into each 32-byte cache block.

Now, $0xA000$ in binary is 1010 0000 0000 0000, which corresponds to cache index 256. Similarly, $0xE000$ in binary is 1110 0000 0000 0000, which also maps to the same index 256.

Because the index bits are the same, both $A[0]$ and $B[0]$ will be loaded into the same cache set (index 256). This leads to a problem: consider the load sequence. When we first load $A[0]$, there is a cache miss (a cold miss, assuming the cache is initially empty) at cache set index 256. Thus, it loads $A[0]A[1], A[2], A[3]$ into the cache set memory block. Next, when we load $B[0]$, it maps to the same cache set index and causes another miss — this time a conflict miss, because it replaces the block holding $A[0]$.

A conflict miss occurs when multiple memory blocks map to the same cache set and compete for space under a direct-mapped cache scheme. In this case, since $A[0]$ and $B[0]$ conflict with each other, loading $B[0]$ causes memory block containing $A[0], A[1], A[2], A[3]$ to be evicted and load $B[0], B[1], B[2], B[3]$ into the cache set memory block. Now, when `load A[1]` is executed, it again maps to the same cache

set (index 256), and because it is not in the cache (recall the memory block containing $A[0], A[1], A[2], A[3]$ was evicted), we get yet another cache miss.

This process continues, resulting in a conflict miss for every access, significantly reducing cache efficiency. Thus, the hit ratio for our program is 0%. The source of the problem is that the elements of arrays A and B are accessed in order and both map to the same cache index. As a result, each new access causes the previous entry to be evicted, leading to no cache reuse. The hit ratio would have been better if the base address of array A had been different from that of array B, such that the memory blocks mapped to different cache set indices. This would have prevented the conflicts and improved the cache performance.

The question now is: was this a contrived example? The root cause of the problem was that we assumed the base addresses of arrays A and B to be $0xA000$ and $0xE000$, respectively, such that both mapped to the same cache set index. Is this an unreasonable assumption that never occurs in the real world? The answer is no—it is not an unreasonable assumption. In fact, this behavior is consistent with the memory allocation model typically followed by compilers.

To understand this better, we must ask: how are variable addresses assigned by the compiler? Typically, the compiler begins allocating memory from a starting address, which is often outside the programmer's control. Suppose, for instance, that the compiler begins by placing array A at address $0xA000$. It then assigns memory to subsequent variables in the order they are declared. Since array A consists of 2048 elements of type `double` (each 8 bytes), the total space it occupies is $2048 \times 8 = 16384$ bytes, or $0x4000$ in hexadecimal. Therefore, array B will be placed at address $0xA000 + 0x4000 = 0xE000$.

$$\text{Address of } B = 0xA000 + 2048 \times 8 = 0xA000 + 0x4000 = 0xE000$$

$$\text{Binary: } 1010\ 0000\ 0000\ 0000 + 0100\ 0000\ 0000\ 0000 = 1110\ 0000\ 0000\ 0000$$

Thus, both arrays may end up mapping to the same cache set index, causing conflict misses, even though this layout results naturally from how compilers assign addresses in practice.

Thus, the problem arises due to the nature of the cache hardware and the specific issue caused by the vector size of 2048. The compiler typically assigns addresses to variables in the order they are declared.

Our objective is therefore to shift the base address of array B just enough so that the cache index of $B[0]$ differs from that of $A[0]$:

```
1 double A[2052], B[2048];
```

Now, the base address of B would be

$$0xA000 + 2052 \times 8 = 0xE020,$$

which corresponds to a cache index of 257. Hence, $B[0]$ and $A[0]$ no longer conflict for the same cache block. The resulting hit ratio will rise to **75%**.

This is a common optimization trick used in languages like **Fortran**. By doing so, we have improved the cache hit ratio—but what impact does this have on the execution time of the program?

Suppose the following piece of code is a dominant part of the program, i.e., its performance heavily influences the overall execution time. Let us now analyze the number of cycles required for execution.

Assume there are 20 machine instructions inside the loop: 2 load instructions and 18 others (such as additions, register loads, and loop overhead), which do not access memory and thus take only 1 cycle per instruction. The total number of instructions executed is therefore 20×2048.

Assume each instruction takes 1 cycle to execute, and a load miss incurs a penalty of 100 cycles to fetch the data from memory.

- **Best Case (100% hit ratio):** All instructions take only 1 cycle. Thus, total cycles =
$$20 \times 2048 = 40960 \text{ cycles.}$$
- **Worst Case (0% hit ratio):** All 18 non-load instructions take 1 cycle each. The 2 load instructions take 100 cycles each due to cache misses. Thus, the total number of cycles is
$$(18 \times 1 + 2 \times 100) \times 2048 = (18 + 200) \times 2048 = 446464 \text{ cycles.}$$
- **Intermediate Case (75% hit ratio):** 75% of the loads are cache hits (taking 1 cycle), and 25% are misses (taking 100 cycles). So the load-related cycles are
$$0.75 \times 2048 \times 2 \times 1 + 0.25 \times 2048 \times 2 \times 100 = 3072 + 102400.$$
The remaining 18 instructions per iteration (which are not memory loads) always take 1 cycle:
$$18 \times 2048 = 36864.$$
Hence, the total number of cycles is
$$3072 + 102400 + 36864 = 142336 \text{ cycles.}$$

Thus, we observe that a better cache hit ratio significantly reduces the execution time—improving performance by almost a factor of 3 compared to the 0% hit case, and closing the gap toward the ideal case.

Another method is called **Array Merging**, where we merge the arrays A and B together, as shown in the following code:

```
1 struct {double A, B;} array[2048];
2 for (i = 0; i < 2048; i++)
3     sum += array[i].A * array[i].B;
```

The nature of the dot product remains the same, but now it picks two elements from the same array element of the newly defined structure. The idea is that spatial locality of reference will help here because the compiler will store `A[i]` and `B[i]` adjacent in memory since they are part of the same `struct`.

Of course, in many programs, such a redefinition might not be possible, because `A` and `B` may have distinct meanings or uses in the calculations being performed, and it may not always be feasible to restructure the data in this way.

In this method, the hit ratio will again be 75% because each cache block will now contain `A[0]`, `B[0]`, `A[1]`, `B[1]`, and so on. Thus, the first access to `A[0]` will be a cache miss due to a cold start, but the accesses to `B[0]`, `A[1]`, and `B[1]` will be cache hits.

1.4.3 Example 3: DAXPY

We now consider another famous example called DAXPY (**D**ouble precision **AX** **P**lus **Y**), where a is a scalar and X and Y are vectors.

```
double X[2048], Y[2048], a;
for (i = 0; i < 2048; i++)
    Y[i] = a * X[i] + Y[i];
```

Listing 1.4: Double precision AX Plus Y (DAXPY)

This example differs slightly from the previous one, as there are three array references per iteration of the loop. Specifically, for each i, we must load `X[i]`, load `Y[i]`, and then store the updated value back to `Y[i]`.

The reference sequence thus becomes:

```
load X[0], load Y[0], store Y[0], load X[1], load Y[1], store Y[1], ...,
                load X[2047], load Y[2047], store Y[2047]
```

Assuming that the base addresses of arrays `X` and `Y` do not conflict in the cache, we can compute the hit ratio. In this case, out of every 12 memory references (two loads and one store per iteration), there will typically be 1 miss for `X[0]` and 1 miss for `Y[0]` due to cold starts. The remaining references will be cache hits assuming spatial locality is exploited effectively. Thus, the overall hit ratio is approximately $\frac{10}{12} \times 100 = 83.3\%$.

1.4.4 Example 4: 2D Matrix Sum

We will now look at the sum of two-dimensional matrices as shown below:

```
double A[1024][1024], B[1024][1024];
for (j = 0; j < 1024; j++)
    for (i = 0; i < 1024; i++)
        B[i][j] = A[i][j] + B[i][j];
```

This is somewhat similar to DAXPY, and the reference memory access sequence would be: `load A[0,0], load B[0,0], store B[0,0], load A[1,0], load B[1,0], store B[1,0],` and so on.

Before analyzing the cache hit rate, it is important to answer a more fundamental question: **In what order are the elements of a multidimensional array stored in memory?**

In the case of one-dimensional arrays, the elements are stored contiguously in memory. However, when dealing with two-dimensional matrices, we must ask whether the matrix is stored row by row (row-major order) or column by column (column-major order). Different programming languages make different assumptions regarding this.

Compilers implement address calculations for multidimensional arrays based on the language's assumed storage order. Therefore, to compile a program correctly in terms of memory load and store operations, the compiler must know how the language stores multidimensional arrays. It is assumed that the compiler already knows the language-specific storage convention. Otherwise, it would have to make its own assumptions, which could severely affect performance portability.

Performance portability refers to the idea that even if a program remains functionally correct across different compilers or languages, its performance may vary significantly depending on how the underlying compiler interprets the array storage order.

This storage order convention is a property of the language, not the compiler. The two common conventions are:

- **Row-major order:**
  - For a two-dimensional array, the elements of the first row are stored first, followed by the elements of the second row, then the third, and so on.
  - This is the convention used in `C`.
- **Column-major order:**
  - For a two-dimensional array, the elements are stored column by column in memory.
  - This is the convention used in `Fortran`.

Now, we are assuming the programming language to be C. In C, multi-dimensional arrays are stored in **row-major order**, which means that elements like $A[0][0]$, $A[0][1]$, $A[0][2]$, and $A[0][3]$ are stored in adjacent memory locations, typically mapping to the same cache block. Therefore, the given program is written inefficiently.

To understand this, consider the reference sequence: if we access $A[1][0]$ immediately after $A[0][0]$, these elements are stored in different rows and thus different cache blocks, resulting in a cache miss. This indicates poor **spatial locality**.

We observe a mismatch between the **reference order**—the order in which memory is accessed—and the **storage order**—the way memory is actually laid out. Our program steps through the matrix column by column, resembling **column-major order**, whereas C stores matrices in **row-major order**: $A[0][0]$, $A[0][1]$, and so on. As a result, our loop does not exploit spatial locality, although it does demonstrate some **temporal locality**, since the store operation to $B[0][0]$ immediately follows a load from $B[0][0]$, which remains in the cache.

Thus, the loop will not show any spatial locality. Here, we assume that packing has been done to eliminate conflict misses due to base addresses. However, in this case, it does not really matter, as there is no spatial locality anyway, and temporal locality will not be affected by conflict misses. So we get:

- A miss on `load A[0][0]` (due to cold start),
- A miss on `load B[0][0]` (due to cold start),
- A hit on `store B[0][0]` (due to temporal locality),

Now, when we go to the next instruction `load A[1][0]`, we again get a miss (due to cold start), and so on. Thus, the hit ratio is approximately 33%, entirely because of the temporal locality—on the store instructions. There are no hits for the load instructions at all.

Will `A[0][1]` be in the cache when required later in the loop?

Now let us analyze the impact of **loop interchange**. Recall that in the initial version of the loop, the second subscript was in the outer loop and the first subscript was in the inner loop. We now consider the interchanged version, where the first subscript is in the outer loop and the second subscript is in the inner loop, as shown below:

```
double A[1024][1024], B[1024][1024];
for (i = 0; i < 1024; i++)
    for (j = 0; j < 1024; j++)
        B[i][j] = A[i][j] + B[i][j];
```

In this case, the memory reference sequence would be:

- `load A[0][0]`, `load B[0][0]`, `store B[0][0]`
- `load A[0][1]`, `load B[0][1]`, `store B[0][1]`
- ...

We observe that the first time `A[0][0]` is loaded, it results in a cache miss due to a cold start. Similarly, `load B[0][0]` will also result in a cache miss. However, the subsequent `store B[0][0]` will be a cache hit, since the value was just loaded and remains in the cache. Similarly, `load A[0][1]` will likely be a cache hit as it resides in the same cache block as `A[0][0]`, assuming a typical cache line size of 4 words. The same applies to the accesses for `B[0][1]`, `store B[0][1]`, and so on.

Hence, out of the first 12 memory instructions (8 loads and 4 stores), 10 will be cache hits and only 2 will be cache misses. This leads to a cache hit rate of:

$$\frac{10}{12} \times 100 = 83.3\%$$

This improved hit rate demonstrates better **spatial locality** due to the access pattern now matching the row-major storage layout of C arrays.

This leads to an important question: is it always safe to perform loop interchange? Consider the following example, where the secondary subscript is controlled by the outer loop:

```
for (j = 1; j < 2048; j++)
    for (i = 1; i < 2048; i++)
        A[i][j] = A[i+1][j-1] + A[i][j-1];
```

This ordering is clearly incorrect since the second subscript, which denotes the column (i.e., 'j'), is controlled by the outer loop. We have already discussed such cases previously. This example is trivial, but many numerical codes still fall into this trap. We are interested in the version where the 'for' loops are interchanged, since we know that doing so can improve spatial locality.

Let us examine the actual behavior of this loop by tracing its first few iterations. For instance:

$$A[1][1] = A[2][0] + A[1][0], \quad A[2][1] = A[3][0] + A[2][0], \quad \text{and so on.}$$

Then, for the next column:

$$A[1][2] = A[2][1] + A[1][1].$$

Now let us blindly interchange the two 'for' loops and observe the effect:

```
for (int i = 1; i < 2048; i++)  // interchanged
    for (int j = 1; j < 2048; j++)
        A[i][j] = A[i+1][j-1] + A[i][j-1];
```

With this ordering, the first few iterations will now be:

$$A[1][1] = A[2][0] + A[1][0], \quad A[1][2] = A[2][1] + A[1][1], \quad \text{and so on,}$$

until we reach:

$$A[2][1] = A[3][0] + A[2][0].$$

Notice the subtle but crucial difference. In the original loop nest, the computation of $A[1][2]$ used updated values of both $A[2][1]$ and $A[1][1]$. However, in the interchanged version, the value of $A[1][2]$ is computed using the old value of $A[2][1]$ (since 'i = 2' has not yet been processed) and the updated value of $A[1][1]$. This changes the computation semantics and may lead to incorrect results.

Hence, **loop interchange is not always safe**, especially when data dependencies exist across loop iterations. One must carefully analyze the dependencies before performing such optimizations.

Is there any safer way to modify the loops so that the code remains correct while also achieving better spatial locality? The answer is yes. This can be accomplished by rewriting the loop to iterate over the second subscript from higher to lower values, as shown below:

```
for (i = 2047; i > 1; i--)
    for (j = 1; j < 2048; j++)
        A[i][j] = A[i+1][j-1] + A[i][j-1];
```

This transformation ensures that data accessed in close temporal proximity is also stored in adjacent memory locations, thus taking better advantage of spatial locality.

However, this example also illustrates an important principle: **loop interchange should not be applied blindly**. Instead, the loop structure should be carefully analyzed or modified to exploit the memory hierarchy effectively. Such strategies can be learned through experience, systematic experimentation, or by exploring various loop arrangements and identifying those that provide the best spatial locality of reference.

1.4.5 Example 5: Matrix Multiplication

Here is an example for matrix multiplication.

```
double X[N][N], Y[N][N], Z[N][N];
for(i=0;i<N;i++)
    for (j=0lj<N;j++)
        for(k=0;k<N;k++)
            X[i][j] += Y[i][k]*Z[k][j];
```

Assume that N is a power of 2. This is the basic form of matrix multiplication we usually learn in high school.

A common way to optimize it is to reduce the number of memory accesses. In the naive version, we access `X[i][j]` repeatedly in the innermost loop to update the running sum. This can be improved.

We can store the running sum in a temporary variable called `tmp`, and after the innermost loop finishes, we copy it into `X[i][j]`. This way, we access `X[i][j]` only once per iteration, which reduces the load/store pressure.

Here's how the updated code looks:

```
double X[N][N], Y[N][N], Z[N][N];
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        double tmp = 0.0; // Running sum
        for (int k = 0; k < N; k++) {
            tmp += Y[i][k] * Z[k][j]; // Dot product
        }
        X[i][j] = tmp;
    }
}
```

Listing 1.5: Optimized Matrix Multiplication using temporary variable

Note: We use `tmp` to hold the dot product of the i^{th} row of Y and j^{th} column of Z.

This saves several memory references by avoiding frequent reads/writes to `X[i][j]`.

Let's now observe how data is accessed during execution:

- To compute `X[0][0]`, we load:
  - `Y[0][0]`, `Z[0][0]`
  - `Y[0][1]`, `Z[1][0]`
  - ...
  - `Y[0][N-1]`, `Z[N-1][0]`

  Then we store `X[0][0]`.

- To compute `X[0][1]`, we load:
  - `Y[0][0]`, `Z[0][1]`
  - `Y[0][1]`, `Z[1][1]`
  - ...
  - `Y[0][N-1]`, `Z[N-1][1]`

  Then we store `X[0][1]`.

And so on, row by row.

In this version of matrix multiplication (loop order `ijk`), the references to Y happen row-wise: `Y[0][0]`, `Y[0][1]`, ..., and so on. This means Y shows excellent *spatial locality of reference*, since successive elements in memory are accessed in sequence.

However, the references to Z occur column-wise: `Z[0][0]`, `Z[1][0]`, ... This results in poor spatial locality for Z, since non-contiguous memory elements are accessed.

Therefore, this version of the loop has good cache behavior for Y but poor cache performance for Z.

Loop intercahnge

Let us now explore the idea of **loop interchange** further. We can rearrange the three nested loops in any order without affecting the correctness of the program. That is, all permutations of the loop order (`ijk`, `jik`, `ikj`, etc.) compute the same result: the matrix product $X = Y \cdot Z$.

For example, we can interchange the i and k loops, resulting in the order `kji`, as shown below:

```
for (int k = 0; k < N; k++)
    for (int j = 0; j < N; j++){
        double tmp = 0.0;
        tmp = Z[k][j];
        for (int i = 0; i < N; i++)
            X[i][j] += Y[i][k] * tmp;
    }
```

Listing 1.6: Matrix multiplication with loop order kji

In this version, observe that for each fixed pair (k, j), the value `Z[k][j]` is reused for all i. So it can be loaded into a register once and reused across the innermost loop. This reduces the number of memory references to Z significantly.

However, now the access pattern for X and Y becomes column-wise:

- Load `X[0][0]`, load `Y[0][0]`, store `X[0][0]`
- Load `X[1][0]`, load `Y[1][0]`, store `X[1][0]`
- ...

This means both X and Y are accessed in a non-contiguous (column-wise) fashion, which leads to poor spatial locality and a high number of cache misses.

So, although the number of loads for Z is reduced, the cache behavior for X and Y becomes worse.

Thus, we see that there are 6 different possibilities for loop variants: ijk, ikj, jik, jki, kij, and kji. Each of them would be correct in computing the matrix multiplication but would have different cache efficiencies. We already saw the cache efficiencies of the ijk and kij loops. Now let us see the cache efficiencies for the other 4. For example, for ikj, the code would be as follows:

```
for (int i = 0; i < N; i++)
    for (int k = 0; k < N; k++) {
        double tmp = Y[i][k];
        for (int j = 0; j < N; j++)
            X[i][j] += tmp * Z[k][j];
    }
```

Listing 1.7: Matrix multiplication with loop order ikj

Here, we see that the reference sequence would be: load `X[0][0]`, load `Z[0][0]`, store `X[0][0]`, load `X[0][1]`, load `Z[0][1]`, store `X[0][1]` and so on. Thus, we see that X and Z are accessed in a row-wise manner. Therefore, this would have excellent spatial locality for both X and Z.

Now let us consider the jik loop order. The corresponding code is:

```
for (int j = 0; j < N; j++)
    for (int i = 0; i < N; i++) {
        double tmp = 0.0;
        for (int k = 0; k < N; k++)
            tmp += Y[i][k] * Z[k][j];
        X[i][j] += tmp;
    }
```

Listing 1.8: Matrix multiplication with loop order jik

Here, we see that the reference sequence would be: load `Y[0][0]`, load `Z[0][0]`, load `Y[0][1]`, load `Z[1][0]`, and so on. The matrix Y is accessed row-wise, and Z is accessed column-wise. After computing the inner loop, we write to `X[0][0]`.

So, in this case, Y has good spatial locality since it is accessed row-wise. But Z is accessed column-wise and hence shows poor spatial locality. The write to $X[i][j]$ happens once per (i, j), so it is fine.

Thus, this loop order has good cache performance for Y, but poor cache behavior for Z.

Now let us consider the jki loop order. The corresponding code is:

```
for (int j = 0; j < N; j++)
    for (int k = 0; k < N; k++) {
        double tmp = Z[k][j];
        for (int i = 0; i < N; i++)
            X[i][j] += Y[i][k] * tmp;
    }
```

Listing 1.9: Matrix multiplication with loop order jki

Here, we see that the reference sequence would be: load `X[0][0]`, load `Y[0][0]`, store `X[0][0]`, load `X[1][0]`, load `Y[1][0]`, store `X[1][0]`, and so on. So, we observe that both X and Y are accessed column-wise, while $Z[k][j]$ is loaded once per inner loop and reused.

Thus, this version shows good reuse of Z due to register storage, but has poor spatial locality for both X and Y because they are accessed column-wise. Therefore, this loop order results in many cache misses for X and Y.

Now let us consider the kij loop order. The corresponding code is:

```
for (int k = 0; k < N; k++)
    for (int i = 0; i < N; i++) {
        double tmp = Y[i][k];
        for (int j = 0; j < N; j++)
            X[i][j] += tmp * Z[k][j];
    }
```

Listing 1.10: Matrix multiplication with loop order kij

Here, we see that the reference sequence would be: load `X[0][0]`, load `Z[0][0]`, store `X[0][0]`, load `X[0][1]`, load `Z[0][1]`, store `X[0][1]`, and so on. This means that both X and Z are accessed row-wise. The element $Y[i][k]$ is loaded once per (i, k) and reused inside the inner loop.

Thus, this version shows excellent spatial locality for both X and Z, and good reuse of Y. Therefore, this loop order gives very good cache performance.

Thus, we can conclude that out of all the possible variants, the loop orders ikj and kij have the best spatial locality of reference. We can clearly see that both X and Z are accessed row-wise, and thus will have minimal cache misses.

Loop Unrolling

We will now talk about loop unrolling. Consider the following code:

```
double X[10];
for (i = 0; i < 10; i++)
    X[i] = X[i] - 1;
```

Listing 1.11: Loop Unrolling

Now consider the following code, which we call unrolled once:

```
double X[10];
for (i = 0; i < 10; i += 2)
    X[i] = X[i] - 1;
    X[i + 1] = X[i + 1] - 1;
```

Listing 1.12: Unrolled Once

In the unrolled-once version, we have included a statement that corresponds to an additional iteration of the original for loop. Thus, in this version, we have halved the number of iterations of the for loop, but each iteration now performs double the work. For instance, in the first iteration, we execute $X[0] = X[0] - 1$ followed by $X[1] = X[1] - 1$, which corresponds to the first two iterations of the original loop but now combined into a single iteration.

Similarly, we could also have a fully unrolled for loop, where we completely remove the loop and simply write all the statements explicitly as shown:

```
X[0] = X[0] - 1;
X[1] = X[1] - 1;
X[2] = X[2] - 1;
...
X[9] = X[9] - 1;
```

So we can have unrolling once, unrolling twice, and so on. In the case of fully unrolled loops, we get a sequence of statements, each of which explicitly identifies array elements. This operation is called loop unrolling.

Loop unrolling helps because it decreases the overhead involved in managing the loop, such as loading the loop variable `i`, incrementing `i`, storing it back, and checking the loop condition $i < 10$, all of which are eliminated in a fully unrolled version. Thus, unrolled loops are guaranteed to perform better from this perspective.

Let us now see how we can unroll matrix multiplication. Consider the original matrix multiplication code that we had:

```
double X[N][N], Y[N][N], Z[N][N];
for(i = 0; i < N; i++)
    for(j = 0; j < N; j++)
        for(k = 0; k < N; k++)
            X[i][j] += Y[i][k] * Z[k][j];
```

Let us unroll the k loop once as shown:

```
double X[N][N], Y[N][N], Z[N][N];
for(i = 0; i < N; i++)
    for(j = 0; j < N; j++)
        for(k = 0; k < N; k += 2)
            X[i][j] += Y[i][k] * Z[k][j] + Y[i][k+1] * Z[k+1][j];
```

We now have a for loop in which there are 5 array references in the innermost loop (anyways, we are not concerned with $X[i][j]$ as it can be replaced with a temporary variable). Note that we are accessing the elements of Y row-wise as $Y[i][k]$ and $Y[i][k+1]$, which provides good spatial locality of reference. However, the Z matrix is accessed column-wise as $Z[k][j]$ and $Z[k+1][j]$, which is not favorable for spatial locality.

Now let us unroll the j loop once as shown:

```
double X[N][N], Y[N][N], Z[N][N];
for(i = 0; i < N; i++)
    for(j = 0; j < N; j += 2)
        for(k = 0; k < N; k += 2){
            X[i][j] += Y[i][k] * Z[k][j] + Y[i][k+1] * Z[k+1][j];
            X[i][j+1] += Y[i][k] * Z[k][j+1] + Y[i][k+1] *
                Z[k+1][j+1];
        }
```

Now we see that upon unrolling the j loop once, there is a reference to $Z[k][j]$ and in the next statement to $Z[k][j+1]$, and similarly we have references to $Z[k+1][j]$ and $Z[k+1][j+1]$.

Thus, we observe that $Z[k][j]$ and $Z[k][j+1]$ will likely reside in the same cache block, and similarly, $Z[k+1][j]$ and $Z[k+1][j+1]$ will also be in the same cache block. Therefore, we are now exploiting spatial locality for both the arrays Y and Z.

Moreover, we are exploiting the temporal locality of Y since $Y[i][k]$ is accessed in the first statement and then again in the subsequent statement, and similarly $Y[i][k+1]$ is accessed multiple times. Hence, we benefit from both spatial and temporal locality for array accesses.

Note that the four elements of the matrix Z accessed in the two statements are neighboring elements of two consecutive rows and two consecutive columns—forming a neighborhood of size 2×2. This forms the basis for a more general approach to improving locality in programs that use multi-dimensional arrays, called **blocking or tiling**.

The core idea behind blocking is to go beyond the 2×2 locality created by unrolling, and instead to explicitly construct a locality of arbitrary size by reorganizing the iteration order. This is done by introducing two additional outer loops that iterate over blocks of the matrix. The following code demonstrates this blocking strategy:

```
double X[N][N], Y[N][N], Z[N][N];
for(jj = 0; jj < N; jj += B)
    for(kk = 0; kk < N; kk += B)
        for(i = 0; i < N; i++) {
            for(j = jj; j < min(jj + B, N); j++) {
                sum = 0.0;
                for(k = kk; k < min(kk + B, N); k++)
                    sum += Y[i][k] * Z[k][j];
                X[i][j] += sum;
            }
```

```
11          }
```

Listing 1.13: Blocking or Tiling

In this code, the loops over `jj` and `kk` step in increments of B, which represents the blocking size. This effectively divides the matrices Y and Z into smaller submatrices (tiles) of size $B \times B$.

By doing this, we can load a block of data into the cache and reuse it multiple times within the inner loops, thereby significantly improving cache performance. The three innermost loops perform multiplication over the selected $B \times B$ blocks as defined by the current values of `jj` and `kk`.

This reordering of operations in a cache-aware fashion allows us to control and optimize spatial locality. Importantly, the value of B—the block size—can be tuned depending on the cache size and architecture, enabling different levels of locality for different hardware settings. Note that one could experiment with loop interchange as well as blocking in order to optimize even further.

Example

Consider the following matrix transpose routine:

```
1 typedef int array[4][4];
2 void transpose(array dst, array src){
3     int i,j;
4     for(i=0; i<4;i++){
5         for (j=0;j<4;j++){
6             dst[j][i]=src[i][j];
7         }
8     }
9 }
```

Assume the code runs on a machine with the following properties:

- Size of integer = 4 bytes.
- The `src` array starts at address 0 and the `dst` array starts at address 64 (decimal).
- There is a single L1 data cache that is direct-mapped with a block size of 16 bytes.
- The cache has a total data size of 128 bytes and the cache is initially empty.

For each `dst[row][col]`, indicate whether the access is a hit (H) or a miss (M):

	Col 0	Col 1	Col 2	Col 3
Row 0	M	H	H	H
Row 1	M	H	H	H
Row 2	M	H	H	H
Row 3	M	H	H	H

Explanation:
The `src` array starts at address 0 and occupies 64 bytes. The `dst` array starts at

address 64 and occupies another 64 bytes, totaling 128 bytes — the same as the cache size. Hence, both arrays can entirely reside in the cache.
The block size is 16 bytes, which means each block can store 16/4 = 4 integers. Thus:

- Number of blocks in the cache: 128/16 = 8
- Block offset bits: $\log_2(16) = 4$

Execution order:

```
load src[0][0], store dst[0][0],
load src[0][1], store dst[1][0], ...
```

The first access to each `dst[row][0]` is a cold miss (first use), but subsequent accesses to `dst[row][1]`, `dst[row][2]`, and `dst[row][3]` hit the same cache block already loaded, resulting in cache hits.
Since no evictions occur (due to sufficient cache size), all misses are cold-start misses. The pattern shows four cold misses — one for each row — followed by hits.

1.5 Vector Operations

1.5.1 Strip mining

Consider the following example on computing vector sum:

```
double A[2048], B[2048], C[2048];
for (i = 0; i < 2048; i++)
    C[i] = A[i] + B[i];
```

Now, what if a CPU has 4 floating point adders? Then, the CPU is capable of executing 4 iterations of the loop in a single cycle. There is an instruction that helps in using the 4 adders as one. Thus, the architecture can be designed to support an instruction that performs 4 iterations of the vector sum loop at a time. This can be done in the following way:

```
VADD v1_A[0:3], v2_B[0:3], v3_C[0:3]
```

Here, the operands to the vector instruction are small vector operands. Since the processor has 4 adders, each operand is of size 4. This is called a *vector instruction* and is made available in commonly used processors through multimedia extensions. Hardware support for operations on short vectors is provided in existing microprocessors.

For example, consider 256-bit registers, each split into 4×64 bits or 8×32 bits. This allows the register to hold 4 double-precision elements of a vector. Thus, one can load 4 consecutive memory locations—corresponding to neighboring elements of a vector—into a single vector register. A few such registers can then be used as operands to vector add or vector multiplication instructions, thereby exploiting the 4 adders present in the CPU.

This mechanism is built around the idea of registers that can be viewed as containing multiple elements of short vectors. These registers can be loaded by the program

using vector load instructions to contain segments of a larger vector, which is the target of the overall computation. There is typically a concept of *maximum vector length* that can be operated on by a single vector instruction. In the case of 4 adders, this length is 4, since we cannot process more than 4 elements of the vector per instruction.

Modern processors implement this through instruction sets like Intel's SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions). For instance, AVX supports 512-bit registers, which can be used in this vector mode.

From now on, we will use a generic notation as shown:

```
C[0:3] = A[0:3] + B[0:3]
```

Note that this is not C programming language syntax, but is simply used here as a notation to understand the behavior of vectorization. This statement implies that the first four elements of vector `A` and vector `B` will be loaded into vector registers and then simultaneously added by the four CPU adders. The result will be stored in the corresponding locations of the array `C`.

Let us look at an example of vectorization, given that the maximum vector length is VL. Consider the following `for` loop:

```
for (i = 0; i < N; i++)
    A[i] = A[i] + B[i];
```

The vectorized version of this loop would look like:

```
for (i = 0; i < N; i += VL)
    A[i:i+VL-1] = A[i:i+VL-1] + B[i:i+VL-1];
```

This loop does not iterate N times, but only N/VL times.

What if N is not divisible by VL? In that case, we require an additional small loop for the remaining elements, as shown in the following code:

```
for (i = 0; i < (N - N % VL); i += VL)
    A[i:i+VL-1] = A[i:i+VL-1] + B[i:i+VL-1];

for (; i < N; i++)
    A[i] = A[i] + B[i];
```

This technique is called **strip mining**, because we are going through the entire vector VL elements at a time.

In practice, this can be achieved not by explicitly including vector instructions in our program, but by asking the compiler to automatically vectorize the loops. This technique is called *autovectorization*. Autovectorization is a compiler feature wherein the compiler analyzes the loops in your program and attempts to use vector instructions where possible.

For example, in `gcc`, the following command-line options are useful:

- `-ftree-vectorize` - Enables autovectorization.
- `-fopt-info-vec` - Provides feedback on autovectorization (e.g., which loops were vectorized).

Note: These options require an optimization level of at least `-O2`.

Note that there are a few possible complications one must be aware of before relying on vectorization.

1.5.2 Node Splitting

For example, if there are dependencies between statements within a loop, then autovectorization will not be able to vectorize that loop.

For example, consider the following code:

```
for (i = 0; i < N; i++) {
    A[i] = B[i] + C[i];
    D[i] = (A[i] + A[i+1]) / 2;
}
```

In the second statement, observe that both $A[i]$ and $A[i+1]$ are used. This introduces a **data dependency** between iterations, because $A[i]$ and $A[i+1]$ may be updated within the same loop execution due to vectorization.

The issue is not with the first statement, which is trivially vectorizable. However, when executing the second statement, we ideally want the new value of $A[i]$ (just updated) and the *old* value of $A[i+1]$. Unfortunately, vectorization updates all values of A in parallel, so both $A[i]$ and $A[i+1]$ may already be updated, leading to incorrect values in $D[i]$.

This problem can be avoided by a technique called **node splitting**, where we store the old value of $A[i+1]$ in a temporary array before updating $A[i]$, as shown below:

```
for (i = 0; i < N; i++) {
    temp[i] = A[i+1];
    A[i] = B[i] + C[i];
    D[i] = (A[i] + temp[i]) / 2;
}
```

This kind of loop transformation—where we copy intermediate data to avoid dependencies—is known as **node splitting**. It helps expose the loop body to vectorization by eliminating cross-iteration dependencies.

1.5.3 Scalar Expansion

Let's look at the following code:

```
for (i = 0; i < N; i++) {
    X = A[i] + 1;
    B[i] = X + C[i];
}
```

In this code, the first statement cannot be vectorized. That's because `A` is an array, but `X` is just a scalar variable. Even if `A` is a vector, the result of `A[i] + 1` is stored in a scalar, so it breaks the vectorization.

Similarly, the second statement cannot be vectorized unless all the variables involved are vectors or constants. To enable vectorization, we can rewrite the code as:

```
for (i = 0; i < N; i++) {
    temp[i] = A[i] + 1;
    B[i] = temp[i] + C[i];
}
```

Here, we've expanded the scalar variable `X` into a temporary array `temp`. This is called **scalar expansion**. Now both operations involve vectors, so the compiler can apply vectorization more easily.

1.5.4 Loop fission

Now, let's consider another example:

```
for (i = 0; i < N; i++) {
    A[i] = B[i];
    C[i] = C[i - 1] + 1;
}
```

In this code, the two statements appear independent, but they behave differently in terms of dependencies.

- The first statement `A[i] = B[i]` has **no data dependency** and can be vectorized.
- The second statement `C[i] = C[i - 1] + 1` depends on the **previous** value of `C`. So, it **cannot** be vectorized. This is a **loop-carried dependency**.

To improve performance, we can split the loop (also called **loop fission**) as follows:

```
for (i = 0; i < N; i++) A[i] = B[i];
for (i = 0; i < N; i++) C[i] = C[i - 1] + 1;
```

Now, the first loop can be vectorized since it is fully independent. The second loop still cannot be vectorized due to the dependency, but at least we gained some performance from the first loop.

1.5.5 Loop interchange

Next, consider this nested loop:

```
for (j = 1; j < N; j++)
    for (i = 2; i < N; i++)
        A[i][j] = A[i - 1][j] + B[i];
```

This loop iterates over columns first and then rows. Since C/C++ store 2D arrays in **row-major order**, this access pattern has **poor spatial locality**. It jumps across memory rows instead of accessing continuous memory locations. This makes it hard for the compiler to vectorize the loop.

To fix this, we can use **loop interchange**, like this:

```
for (i = 2; i < N; i++)
    for (j = 1; j < N; j++)
        A[i][j] = A[i - 1][j] + B[i];
```

Now the loop accesses elements in a **row-wise** fashion, which improves memory locality. This helps in two ways:

- Better cache usage (due to spatial locality),
- Better chances of vectorization.

1.5.6 Summary of Techniques

Technique	Description
Strip Mining	Process multiple elements in a single loop iteration to expose parallelism and improve cache usage.
Node Splitting	Break up computations by copying intermediate results to avoid dependencies and enable parallel execution.
Scalar Expansion	Replace scalar variables with arrays to allow independent computation on each element.
Loop Fission	Split a loop into multiple loops to isolate independent and vectorizable operations.
Loop Interchange	Swap the order of nested loops to access data in memory-friendly patterns (e.g., row-wise instead of column-wise).

Chapter 2

Parallel Architectures

2.1 Motivation

Moore's Law states:

The number of transistors in a dense integrated circuit doubles approximately every two years. (2.1)

This law has held true for the past 50 years. As a result, the number of cores in a processor has steadily increased.

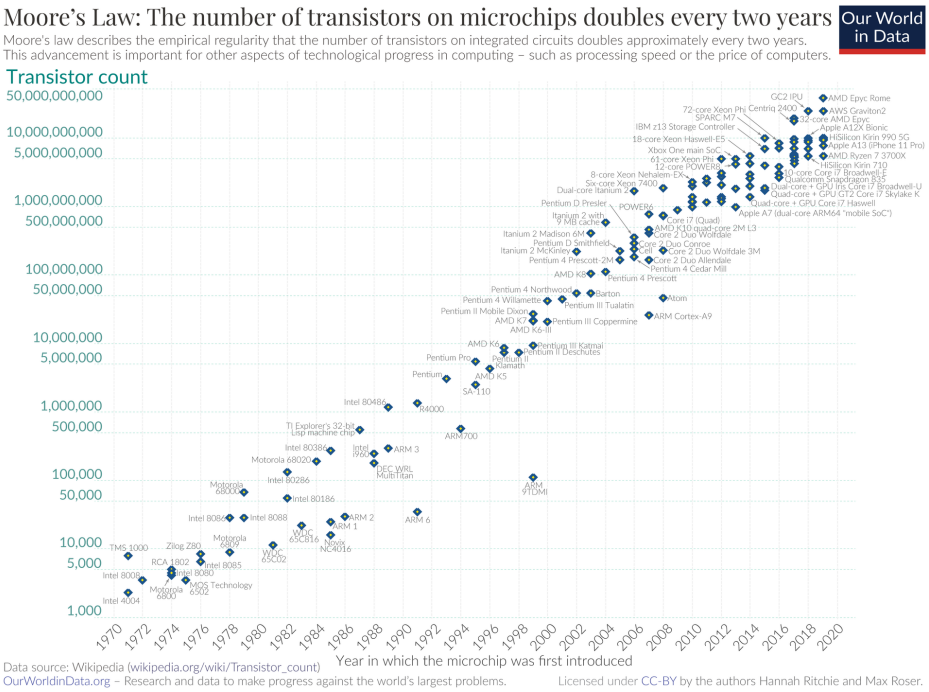


Figure 2.1: Moore's Law Graph

However, we are now reaching a point where computing power is starting to saturate. Power consumption is rising, and heat dissipation is becoming a major challenge. This makes Moore's Law unsustainable in the long run.

One solution to this problem is **parallel computing**.

Parallel computing is the use of multiple processors or cores to perform computations simultaneously. The idea is simple: large problems can often be broken down into smaller parts, and these parts can be solved in parallel.

This leads to a huge reduction in execution time—from days or months to hours or even seconds.

Some common examples where parallel computing is used are:

- Climate modelling
- Bioinformatics
- Computational fluid dynamics (CFD)

Parallel computing is also essential when dealing with large-scale data. For example, companies like Google and Facebook process billions of requests per day using parallel systems.

But not all problems can be parallelized. It's not always possible to divide a problem into smaller, independent parts. Still, for many real-world problems like weather simulations or molecular dynamics, parallelism comes naturally.

Another important point is related to **computer architecture trends**. CPU speeds have mostly saturated in recent years. The performance bottleneck is now caused by slow memory bandwidth and latency.

Parallelism helps here too. It allows us to overlap computation with data transfer, which improves overall performance.

2.2 Classification of Architectures — Flynn's Taxonomy

Flynn's Taxonomy is a way to classify computer architectures based on the number of instruction streams and data streams they can handle. It divides systems into four categories:

- **SISD** – Single Instruction, Single Data (used in traditional serial computers)
- **SIMD** – Single Instruction, Multiple Data
- **MISD** – Multiple Instruction, Single Data
- **MIMD** – Multiple Instruction, Multiple Data

Each category represents a different model of parallelism.

2.2.1 SIMD — Single Instruction, Multiple Data

SIMD stands for **Single Instruction, Multiple Data**. In this model, the same instruction is executed on multiple data elements at the same time. It is highly efficient for problems that apply the same operation across large datasets.

SIMD is implemented using:

- **Vector Processors** – These execute a single instruction on a vector of data elements in parallel.
- **Processor Arrays** – These consist of multiple processing elements working together on different data elements, all controlled by the same instruction.

Some common examples of SIMD-based systems and technologies include:

- Connection Machine CM-2
- Cray-90
- Intel Xeon Phi
- Modern CPUs with vector instruction sets (e.g., Intel AVX-512, which operates on 512-bit vectors)

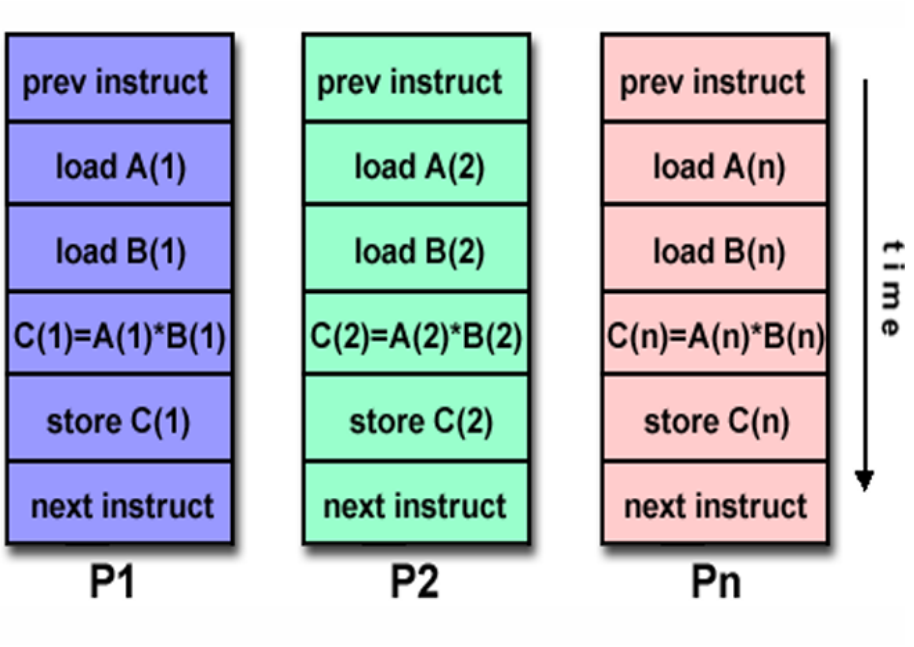


Figure 2.2: SIMD Architecture

You can read more Introduction, Classification of Architectures - Flynn's Taxonomy, and Classification based on Memory, SIMD and other parallel computing concepts in this excellent tutorial: Parallel Computing Tutorial by LLNL Lawrence Livermore National Laboratory [2023].

Consider Figure 2.2. Suppose we are given two vectors, `A` and `B`, each of size n. The goal is to perform element-wise multiplication of these vectors and store the result in a third vector `C`.

Assume we have n processors, labeled $P_1, P_2, \ldots, P_n$. In an SIMD architecture, each processor is assigned one element from each vector. That is:

- Processor P_1 will work on the first elements: $A(1)$ and $B(1)$.
- Processor P_2 will work on the second elements: $A(2)$ and $B(2)$.

- And so on, until processor P_n works on $A(n)$ and $B(n)$.

All processors execute the same instruction in lock-step, but on different pieces of data. The execution proceeds as follows:

1. All processors simultaneously load their assigned element from vector `A`. So, P_1 loads $A(1)$, P_2 loads $A(2)$, ..., P_n loads $A(n)$.
2. Next, all processors load their respective elements from vector `B` in parallel. So, P_1 loads $B(1)$, P_2 loads $B(2)$, ..., P_n loads $B(n)$.
3. Now, each processor performs multiplication of the two elements it holds. So, P_1 computes $A(1) \times B(1)$, P_2 computes $A(2) \times B(2)$, ..., P_n computes $A(n) \times B(n)$.
4. Finally, all processors store the result into the corresponding location in the result vector `C`. That is, P_1 stores the result in $C(1)$, P_2 in $C(2)$, ..., P_n in $C(n)$.

Thus, the entire element-wise multiplication is performed in parallel. This is a typical example of how SIMD architecture operates—using the same instruction applied across multiple data elements simultaneously.

2.2.2 MISD – Multiple Instruction Single Data

The MISD (Multiple Instruction, Single Data) model is quite rare in practice and is primarily used in highly specialized systems. In this architecture, **multiple processors execute different instructions on the same data stream**.

Such systems are mostly found in:

- **Fault-tolerant systems**: where redundancy is crucial for reliability.
- **Redundant systems**: multiple processing units compute the same result in different ways to detect errors.
- **N-modular redundancy**: a technique where N identical modules process the same data and their outputs are compared using a majority voting system.
- **Cryptographic systems**: where the same data is subjected to different algorithms to test for vulnerabilities or perform parallel brute-force attempts.

While theoretically important, MISD is not widely used in general-purpose computing due to its limited applicability.

2.2.3 MIMD – Multiple Instruction Multiple Data

MIMD (Multiple Instruction, Multiple Data) is the most general and widely used parallel architecture today. In this model, **each processor executes a different instruction on a different data element at the same time**.

Some real-world examples include:

- **IBM SP systems**

- **High-performance supercomputers**
- **Clusters and distributed computing environments**
- **Computational grids**

At a given time step, different processors in an MIMD system can:

- Fetch different instructions from memory,
- Operate on different data,
- And execute independently from one another.

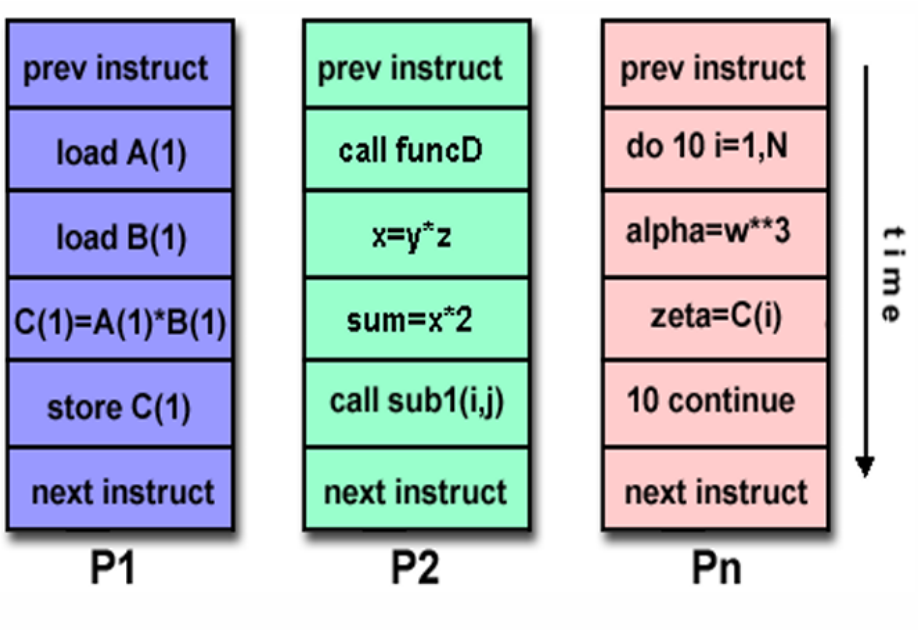


Figure 2.3: MIMD Architecture

As shown in Figure 2.3, each processor $P_1, P_2, \ldots, P_n$ operates independently. For example:

- P_1 might be performing element-wise multiplication,
- P_2 could be executing a sorting routine on a different dataset,
- P_3 might be running a simulation or numerical computation,
- and so on.

This flexibility makes MIMD architectures highly suitable for a wide range of problems, such as:

- Complex scientific simulations
- Multitasking operating systems
- Distributed databases

- Task-parallel and data-parallel programs

Key Idea: MIMD allows maximum flexibility — each processor can follow its own execution thread and process different datasets independently. This makes it the dominant model in modern parallel and distributed computing systems.

2.3 Classification Based on Memory

Parallel computing systems can also be classified based on the type of memory architecture they use. The two broad types are:

- Shared Memory
- Distributed Memory

2.3.1 Shared Memory

In a **shared memory** architecture, all processors have access to a common global memory. This means that each processor can directly read from and write to the same physical memory. Such systems require special hardware support for memory management, which we will discuss later.

One of the main advantages of shared memory systems is that they are **easier to program**. Since all processors work in a single address space, communication between them can be done through regular reads and writes to shared variables.

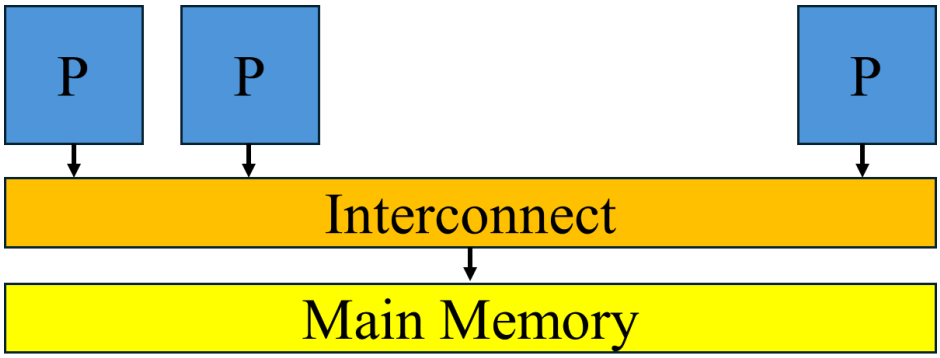


Figure 2.4: Shared Memory Architecture

As shown in Figure 2.4, all n processors (P1, P2, ..., Pn) share the same physical memory. For example, consider the memory address `0x7FFF5FBFFD98`. If processor P1 writes to this address, then processor P2 reading from the same address will see the same data. That is, `0x7FFF5FBFFD98` refers to the *same physical location* for all processors.

This concept is known as a **shared physical address space**. This idea will become clearer when we contrast it with distributed memory systems later.

While shared memory simplifies communication, it introduces several challenges:

- **Race conditions:** When multiple processors access and modify the same memory location without proper synchronization.

- **Cache coherence:** Each processor might keep a local cache of memory. Keeping all caches in sync is non-trivial.
- **Memory contention:** Multiple processors trying to access memory at the same time can cause performance bottlenecks.

We will explain in detail how these issues are managed using cache coherence protocols (like MESI), memory consistency models, and other architectural techniques.

In shared memory systems, communication is **implicit** — processors communicate by reading from and writing to shared memory locations. There is no need for explicit message passing.

In short: All processors access the same memory. This makes programming easy but introduces problems like synchronization and cache consistency.

There are two types of shared memory architectures:

- **Uniform Memory Access (UMA)** – All processors have equal access time to all memory locations.
- **Non-Uniform Memory Access (NUMA)** – Access time depends on how far the memory is from the processor.

In general, the time taken by a processor to access a memory location depends on the distance between that processor and the memory module.

Uniform Memory Access (UMA)

In UMA systems, all processors take the same amount of time to access any memory location. These systems are easy to program because memory behavior is predictable.

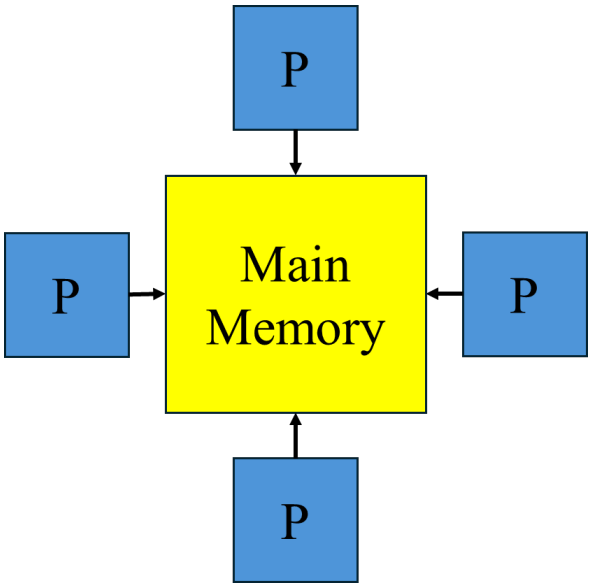


Figure 2.5: Uniform Memory Access Architecture

As shown in Figure 2.5, all processors are equally distant from the memory. So, whether P1 accesses memory location M_1 or P2 accesses memory location M_2, the time taken is the same. All memory locations appear equally accessible to all processors.

This uniform access time simplifies data placement and synchronization.

Non-Uniform Memory Access (NUMA)

In NUMA systems, memory is physically divided and parts of it are closer to some processors than others. So, access time varies depending on which processor is accessing which memory block.

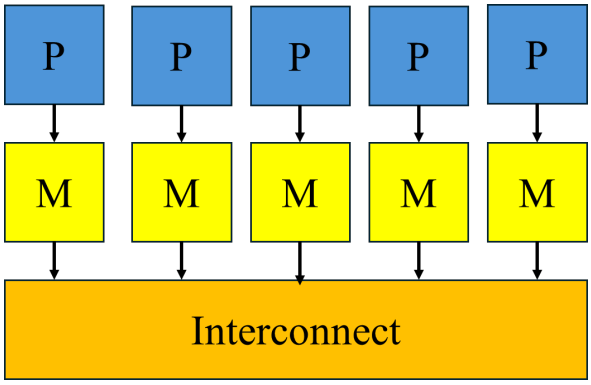


Figure 2.6: Non-Uniform Memory Access Architecture

As shown in Figure 2.6, each processor has a portion of memory that is physically closer to it. Accessing this "local" memory is faster than accessing "remote" memory, which belongs to another processor.

This variation in access time adds complexity. Programmers must be aware of data locality to get good performance. Data structures should ideally be placed in memory close to the processor that uses them the most.

2.3.2 Distributed Memory Architecture

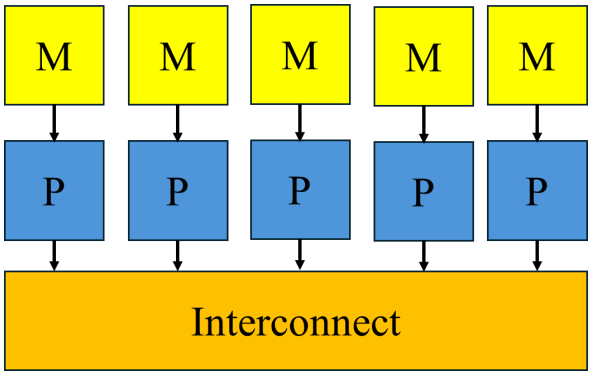


Figure 2.7: Distributed Memory Architecture

As shown in Figure 2.7, each processor has its own private memory. Unlike shared memory systems, there is no common physical address space. If two processors access the same virtual address, say `0x7FFF5FBFFD98`, they may get completely different data — because this address refers to distinct physical memory regions in each processor's local memory.

In distributed memory systems, the only way processors can share data is through **explicit communication**, typically implemented via *message passing*. This means

that a processor must send data to another processor using communication primitives like `send` and `receive`.

Such architectures are generally used in systems with a very large number of cores — such as in high-performance computing clusters or supercomputers — where scaling shared memory becomes impractical.

While distributed memory systems are powerful and scalable, they are also harder to program. Programmers must explicitly manage communication and data distribution among processors.

In short: Distributed memory systems do not share a global address space. Communication between processors happens via message passing, which provides scalability but adds programming complexity.

2.3.3 Shared Memory Architectures: Cache Coherence

Refer to Figure 2.4. In a shared memory system, all processors are connected to the main memory through an interconnect. However, even though the memory is shared, each processor typically has its own private cache for faster access to frequently used data.

Now consider a variable `X` stored in the main memory with an initial value of 5. Suppose:

- Processor `P1` accesses `X`. Since it is not in `P1`'s cache, a **cache miss** occurs, and the value 5 is fetched from main memory and stored in `P1`'s cache.
- Processor `P2` also accesses `X`. It too experiences a **cache miss**, and the value 5 is fetched into `P2`'s cache.
- Now, suppose `P1` updates `X` to 10. Since `X` is already in `P1`'s cache, this results in a **cache hit**, and the value is updated only in `P1`'s local cache.
- If `P2` now accesses `X`, it experiences a **cache hit** as well — but retrieves the stale value 5 from its local cache.

This leads to a serious issue: the value of `X` is no longer consistent across different processor caches. This inconsistency is known as the **Cache Coherence Problem**. It violates the fundamental assumption of a shared memory system — that all processors see a consistent view of memory.

Maintaining cache coherence is critical in such systems and is typically handled via hardware-based coherence protocols like:

- **MESI (Modified, Exclusive, Shared, Invalid)**
- **MOESI (Modified, Owned, Exclusive, Shared, Invalid)**
- **MSI (Modified, Shared, Invalid)**

These protocols ensure that any update to a memory location by one processor is eventually visible to all other processors, preserving consistency.

Cache Coherence Protocols

There are two primary strategies to solve the cache coherence problem in shared memory architectures:

- **Write Update Protocol:** In this approach, whenever a processor writes to a variable, it propagates the updated value to all other processors' caches and also updates the value in the global memory.
  - The **write operation** is slower because it requires broadcasting the updated value to all caches.
  - The **read operation** is faster since the updated data is already available in the local cache of each processor.
- **Write Invalidate Protocol:** Here, when a processor updates a variable, it sends an *invalidate signal* to all other processors to mark their cached copy of that variable as invalid.
  - The **write operation** is faster since it only involves sending an invalidation signal rather than the full data.
  - The **read operation** becomes slower if the processor attempts to access the invalidated data, resulting in a cache miss and fetching from main memory.

Note

In the write update protocol, even if another processor is not going to use the updated data, its cache is still updated — leading to unnecessary data transfers and increased bus traffic.
In contrast, the write invalidate protocol only sends an invalidation signal, and the data is fetched from memory *only if needed*. This reduces unnecessary communication.
Hence, most modern systems prefer the write invalidate protocol due to its efficiency in reducing coherence traffic. However, note that the exact implementation details of these protocols may vary depending on the architecture.

State Transition Diagram

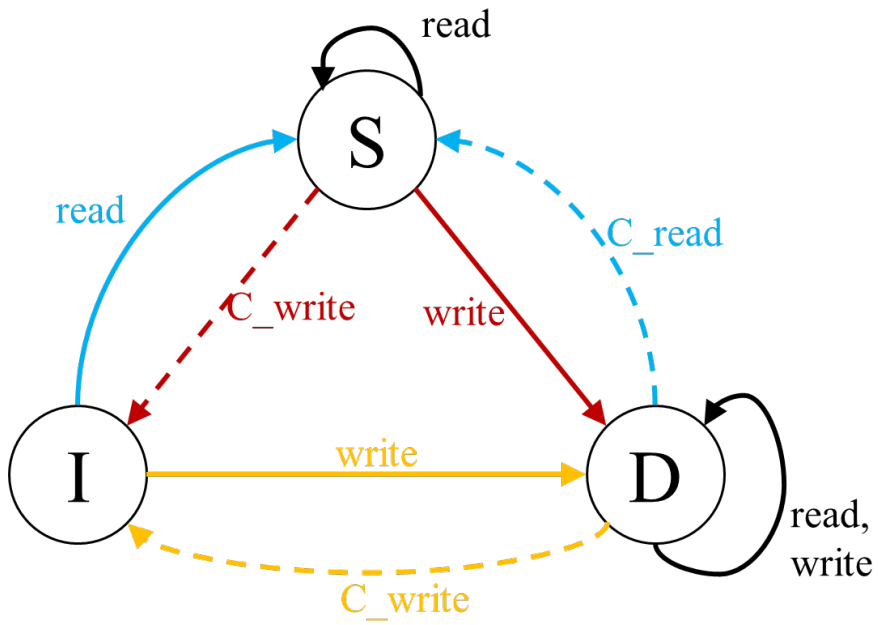


Figure 2.8: State Transition Diagram for Invalidate Protocol

A cache line in a multiprocessor system using an *invalidate-based* coherence protocol can exist in one of the following three states:

- **Shared (S):** The cache line is valid and may be present in multiple processors' caches. No processor has modified it.
- **Dirty (D):** The cache line has been modified by the processor and is not present in any other processor's cache. It is the only valid copy.
- **Invalid (I):** The cache line is no longer valid — usually because another processor has modified the data and sent an invalidation signal.

In Figure 2.8, the state transition diagram for the invalidate protocol is illustrated. The transitions are triggered by processor-initiated memory operations and by actions from the coherence protocol.

- **Solid arrows** denote state transitions caused by *local processor operations* (e.g., read, write).
- **Dashed arrows** denote state transitions triggered by *coherence protocol events* (e.g., receiving an invalidation or update message).

This diagram helps visualize how a cache line changes state depending on processor actions and coherence enforcement.

Now consider Figure 2.8. Suppose the cache line is in the **Shared (S)** state, i.e., the data is shared by at least two caches (in different processors). If it is *read* by either of the processors, it will remain in the Shared state (S), as indicated by the solid black self-loop.

If it is *written* by one of the processors, it will transition from the Shared (S) state to the **Dirty (D)** state for the writing processor, as shown by the solid red arrow from S to D. Due to the cache coherence protocol, the corresponding cache line in the other processor transitions from Shared (S) to **Invalid (I)**, as indicated by the dashed red arrow from S to I.

If the cache line is in the **Dirty (D)** state, i.e., the data has already been modified by a processor, and it is either read or written again by the *same* processor, the state remains Dirty (D). This is depicted by the solid black self-loop from D to D.

If the cache line is in the **Invalid (I)** state, i.e., the data has been modified by another processor, and it is *read*, a cache miss occurs. The data is then fetched from main memory, and the cache line transitions to the Shared (S) state. This transition is shown by the solid blue arrow from I to S in Figure 2.8. As a result of this read, the cache coherence protocol updates the state in the other processor from Dirty (D) to Shared (S), represented by the dashed blue arrow from D to S.

If the cache line is in the **Invalid (I)** state and a *write* occurs, the cache line transitions to the Dirty (D) state, as shown by the solid yellow arrow from I to D. To maintain coherence, the writing processor sends an *invalidate* signal to all other processors. Consequently, any cache line in the Dirty (D) state in another processor transitions to Invalid (I), as indicated by the dashed yellow arrow from D to I.

2.3.4 Implementation of Cache Coherence Protocols

Snooping Protocol

The snooping protocol is commonly used in bus-based architectures, where only one memory operation can occur at any given time. In this setup, all memory operations are broadcast over a shared bus and are *snooped* by other processors.

Each processor is equipped with a specialized cache controller that continuously monitors (or snoops on) the bus for memory transactions initiated by other processors. When a write operation occurs in the cache of any processor, it is broadcast publicly on the bus. This broadcast allows all other cache controllers—listening on the bus—to detect the update and invalidate the corresponding cache lines if they hold stale copies of that data.

While the snooping protocol ensures correctness and consistency, it suffers from several limitations:

- Only one memory operation can be executed at a time, leading to serialization of memory accesses and limited parallelism.
- All processors must constantly snoop on the bus, which increases bus traffic and energy consumption.
- Specialized hardware (cache controllers) is needed for each processor, increasing architectural complexity and cost.
- The protocol is not scalable—performance degrades significantly as the number of processors increases due to bus contention.

Due to these drawbacks, particularly the limited scalability, more advanced systems often employ **directory-based protocols**, which eliminate the need for broadcast communication and offer better performance in large-scale multiprocessor systems.

Directory-Based Protocol

The directory-based protocol is typically used in network-based architectures and is well-suited for large-scale systems due to its scalability and efficiency.

Unlike the snooping protocol, where all processors—even those not involved with the relevant data—continuously listen on the bus, the directory-based approach reduces unnecessary overhead by targeting only the relevant processors. This is achieved by maintaining a *directory* for each cache line, which records which processors currently possess copies of that line.

Instead of broadcasting every memory operation to all processors, coherence messages (such as invalidations) are sent only to those processors that actually hold the data. A centralized directory is used to track the cache coherence states and ownership information.

Directory Representation Assume a system with M cache lines and P processors. The directory can be represented as a matrix of size $M \times P$, where each row corresponds to a cache line and each column corresponds to a processor. Each entry in this matrix is a presence bit:

- A presence bit of `1` at position (i, j) indicates that processor j has a copy of cache line i.
- A presence bit of `0` indicates that the processor does not hold that cache line.

In addition to presence bits, the directory also maintains information about the **current owner** of each cache line—i.e., the processor that has the most updated copy and is responsible for responding to memory requests.

Full Bit Vector Scheme This is known as the *Full Bit Vector Scheme*, where the number of presence bits is $O(M \times P)$. While simple and direct, this approach can become infeasible for large values of M and P because:

- A large number of presence bits consume significant memory just for directory management.
- Less memory is left for storing actual data.
- Directory accesses may become a bottleneck, causing frequent memory swaps and reducing the benefits of parallelism.

Sparse Bit Vector Scheme To address these limitations, a more memory-efficient approach—called the *Sparse Bit Vector Scheme*—is used. Here, the number of presence bits is reduced to $O(M+P)$ by exploiting the sparsity of access patterns. At any given time, only a small subset of cache lines is relevant to a small subset of processors:

- Let $m \ll M$ be the number of cache lines of current interest.
- Let $p \ll P$ be the number of processors actively interacting with those cache lines.

Since the $M \times P$ matrix is sparse, it can be stored using compact data structures such as lists or maps, dramatically reducing memory usage while still preserving coherence and correctness.

2.3.5 False Sharing

A cache is made up of multiple cache lines. All cache coherence protocols—such as write-invalidate or write-update—operate at the granularity of cache lines and not at the level of individual variables.

Thus, if a variable lies in one of the cache lines shared by two processors, and one of the processors updates some other variable that belongs to the same cache line, then—even though the first processor did not update the variable accessed by the second processor—the entire cache line will be invalidated. This situation is known as **false sharing**.

Let us understand this with an example.

Consider a modern-day cache system where cache lines are typically 64 bytes in size. Now suppose we are storing data of type `double`, which takes 16 bytes per variable. This means we can store four `double` variables per cache line.

Now consider two processors, P1 and P2. Assume that a cache line is shared between them. Suppose processor P1 updates the first variable of this cache line. Meanwhile, processor P2 only requires access to the second variable and never touches the first one.

However, since cache coherence protocols work at the level of cache lines and not at the level of individual variables, any update to the cache line by P1 causes the entire cache line to be invalidated in the cache of P2—even though P2 only needs access to a different variable within the same cache line.

This results in:

- Unnecessary cache invalidations,
- Increased cache misses, and
- Degraded system performance.

Thus, although P1 and P2 are not truly sharing the same variable, the coherence mechanism treats the situation as if they are, leading to inefficiencies. This phenomenon is what we refer to as *false sharing*.

For example, consider a Fortran program in which each process or thread accesses a row of a matrix. In Fortran, matrices are stored in a column-major order, meaning elements in a column are stored contiguously in memory. This storage layout can introduce **false sharing** when multiple threads access different rows of the same column.

Suppose processor P1 accesses the first element of row 1 (i.e., the first column), and processor P2 accesses the first element of row 2 (which also lies in the first column). Due to column-major storage, both elements reside in adjacent memory locations and potentially within the same cache line. Thus, even though P1 and P2 are accessing distinct rows, they access memory locations that fall within the same cache line. When processor P1 updates its value, the cache coherence protocol will invalidate the entire cache line in P2's cache, even though P2 does not use P1's data. This causes false sharing, resulting in unnecessary cache invalidations and degraded performance.

Solution: Padding To solve this issue, one can reorganize the code such that each processor accesses an entire set of rows rather than individual elements, and ensure that no two processors write to the same cache line. However, even this approach may

result in overlapping cache lines when the matrix dimensions are not divisible by the number of processors.

In such cases, we employ a technique called **padding**, which involves adding dummy elements (i.e., extra columns or rows) to the matrix to align data to cache line boundaries. Padding ensures that each processor's working data fits neatly into distinct cache lines, thereby avoiding false sharing.

Example: If a cache line is 64 bytes wide and each `REAL*8` (double precision) value occupies 8 bytes, then each cache line holds 8 elements. By adding dummy columns or spacing rows apart with padding, we can ensure each thread works on separate cache lines.

To summarize: false sharing occurs when multiple processors access different variables that happen to reside in the same cache line. Any write to a variable by one processor causes the entire line to be invalidated in others' caches, even if those processors were using different parts of the data. This leads to unnecessary triggering of cache coherence protocols and performance penalties.

Chapter 3

Interconnection networks for a Parallel Computer

Interconnection networks for a parallel computer provide mechanisms for data transfer between processing nodes or between processor and memory modules. A black-box view of interconnection network consists of n inputs and m outputs. Interconnections networks are build using links and switches.**Links** are set of wires or fibres for carrying information. They limit the speed of propagation because of capacitive coupling, attenuation of signal strength which are a function of length of link. **Switch:** A switch maps input ports to output ports. Degree of a switch is the total number of ports on a switch. It supports internal buffering when output ports are busy and allows Routing (to alleviate congestion on Network) and multicast (same output on multiple ports). There are two type of networks:

- Static/Direct Networks - It is point to point communication links among the processing nodes Example: Hypercube, Mesh, Torus, etc.

- Dynamic/Indirect Networks - They are connected by switches linking processing nodes and memory banks. Example: Crossbar, Omega, etc.

The classification is as shown in figure 3.1. The diagram on the left shows a Static Network and the diagram on the right shows Indirect Network.

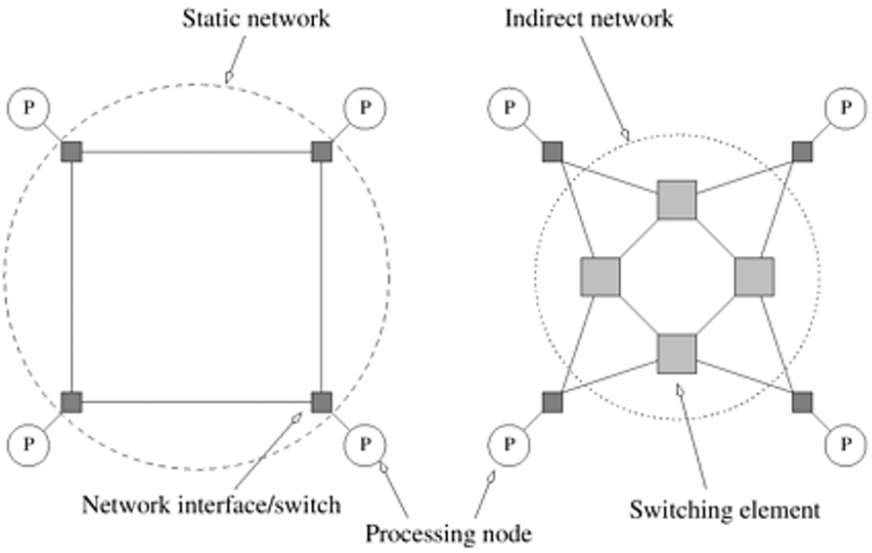


Figure 3.1: Classification of Interconnection Network

3.1 Network Topology

Network topology is evaluated broadly in terms of cost and scalability with performance.

3.1.1 Bus-based networks

It is a shared medium that is common to all node. They are scalable in cost but unscalable in terms of performance. The cost of the network is thus proportional to the number of nodes p, thus scales as $O(p)$. The distance between any two nodes in the network is constant $O(1)$. Since they broadcast information among nodes, there is a little overhead associated with broadcast compared to point to point message transfer. The bounded bandwidth places limitation on the overall performance of the network. Example: Intel Pentium and Sun Enterprise servers. The demands for bandwidth can be reduced by cache for each node i.e. Private data, thus, only remote data is to be accessed through the bus. This is as shown in figure 3.2.

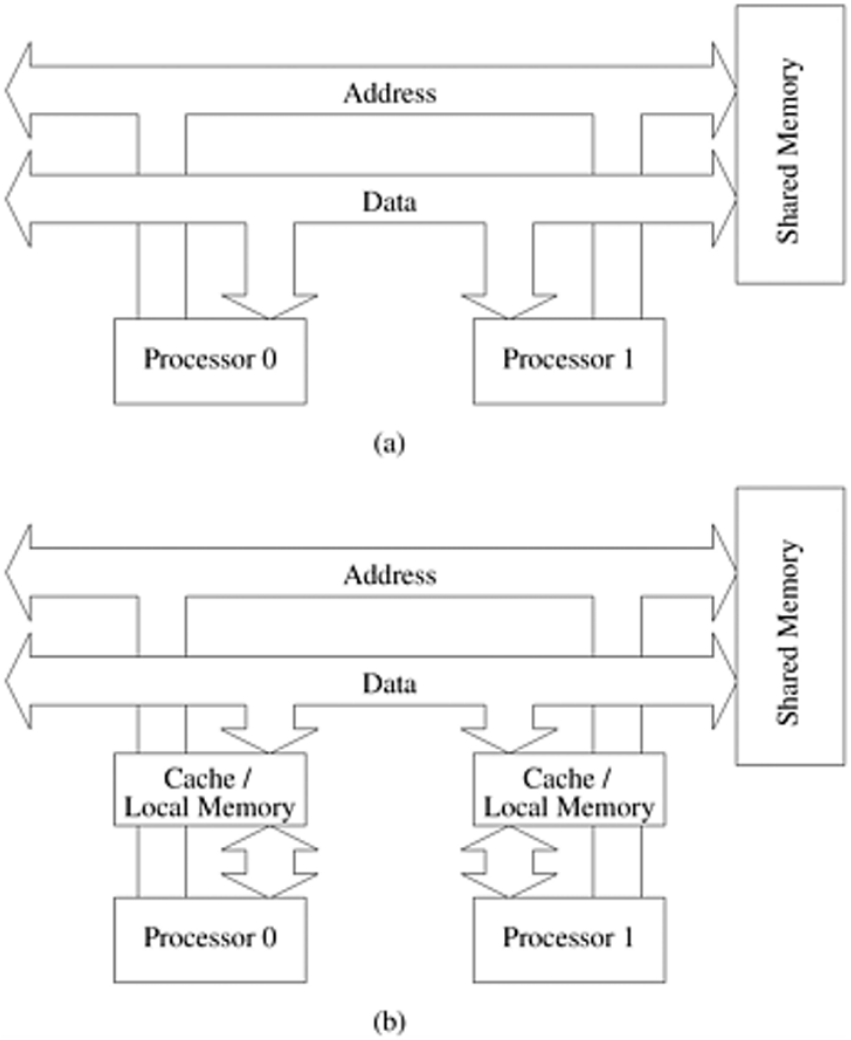


Figure 3.2: Bus-based Network

The figure at the top shows a bus-based interconnects with no local caches. The figure at the bottom shows a bus-based interconnects with local memory/caches.

3.1.2 Cross-bar Networks

Consider the figure 3.3. It connects p processors to b memory banks. It is a non-blocking network i.e. the connection of a processing node to a memory bank does not block the connection of any other processing nodes to other memory banks.

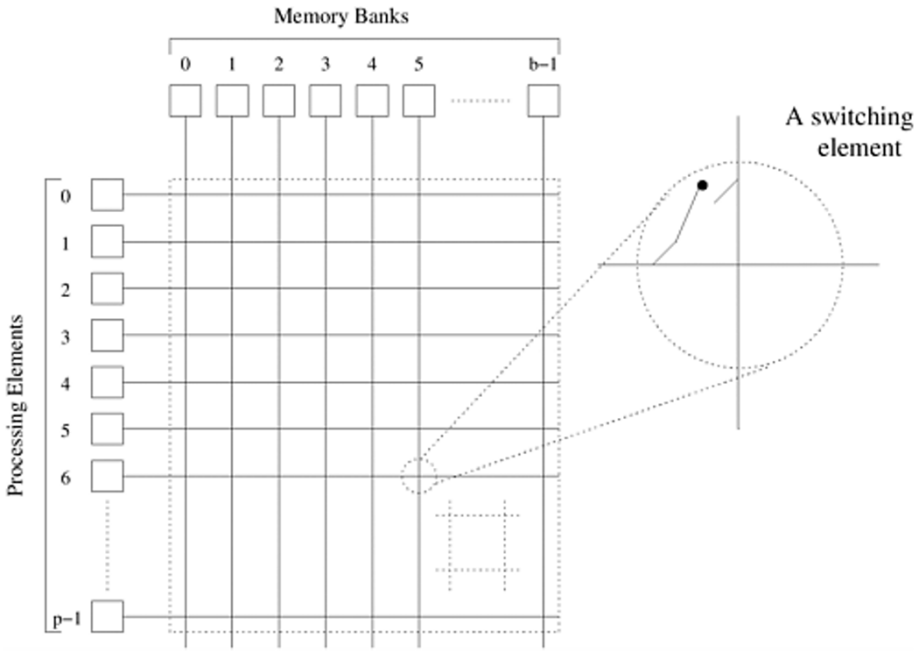


Figure 3.3: Cross-bar Network

The total number of switching nodes is $\Theta(pb)$. Generally, $b > p$ so that all the processors can access some memory bank. Thus as p is increased, the complexity of switching network grows as $\Omega(p^2)$. Thus, as number of processing nodes becomes large, switch complexity is difficult to realize at high data rates. Thus, Cross bar networks are not scalable in terms of cost but are scalable in terms of performance.

3.1.3 Multistage Network

Consider the figure 3.4. It lies between crossbar and bus network topology. It is more scalable than bus in terms of performance and more scalable than cross bar in terms of cost.

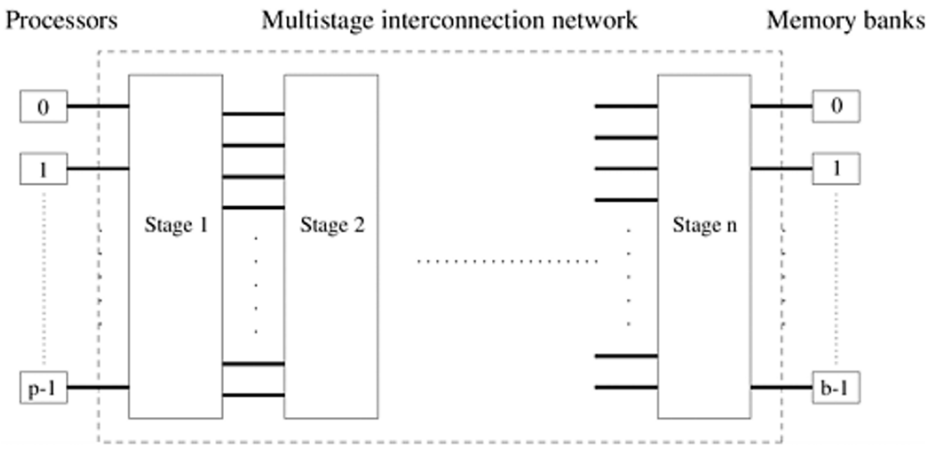


Figure 3.4: Multistage Network

Omega Network is a type of multistage network with p number of inputs (p processing node), p number of outputs (p memory banks) and log(p) number of stages

each consisting of p/2 switches. At any consecutive intermediate stage, a link exists between input i and output j according to the following interconnection pattern:

$$\text{j} = \begin{cases} 2i & 0 \le i \le p/2 - 1 \\ 2i + 1 - p & p/2 \le i \le p - 1 \end{cases} \tag{3.1}$$

This is called a perfect shuffle i.e. left rotation on binary representation of i to obtain j. In order to understand, consider the example shown in the figure 3.5. The figure shows a perfect shuffle interconnection for eight inputs and outputs.

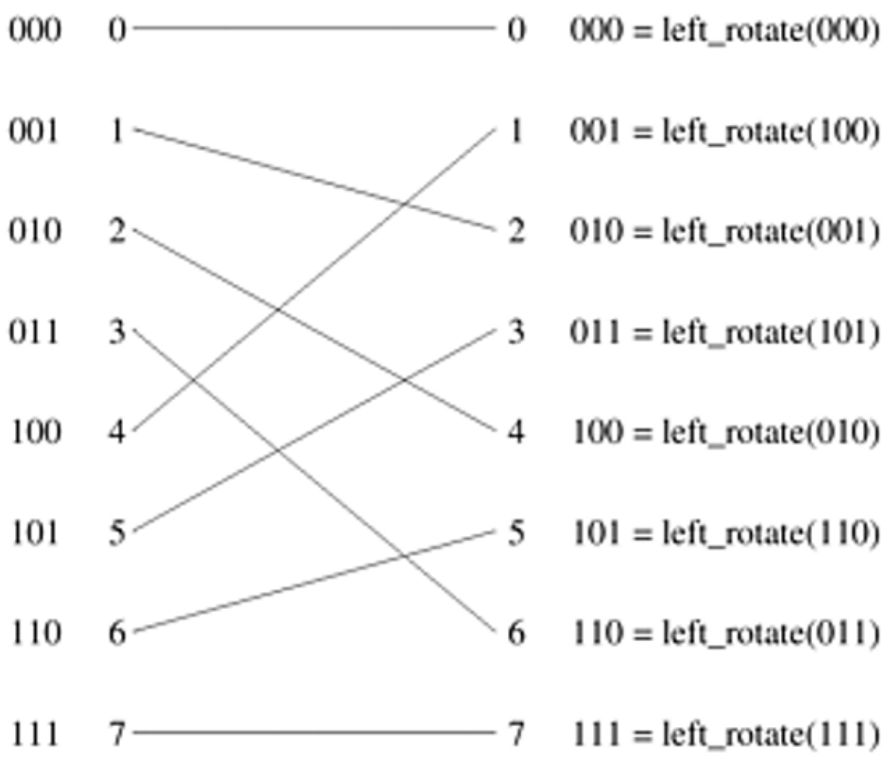


Figure 3.5: A perfect shuffle interconnection for eight inputs and outputs

Starting from the first input 000. It should be connected to the output obtained on left rotating the input i.e. 000. Thus, as can be clearly seen in the figure it is connected to 000. Similarly, the second input 001 should be connected to the output obtained on left rotating the input i.e. 010. Thus, as can be clearly seen in the figure it is connected to 010. Similarly, the third input 010 should be connected to the output obtained on left rotating the input i.e. 100. It can been seen that there is a connection for 010 to 100 and so on for all the inputs till 111 whose left rotation will be 111 and hence a connection between 111 to 111. Switching nodes can be in either of the two configurations pass thorugh or cross-over as shown in the figure

Figure 3.6: Pass-through and Cross-over configuration of Switching Nodes

In figure 3.6 the figure on the left shows pass-through configuration of a switching node in which the inputs are sent straight to outputs and the figure on the right shows cross-over configuration of a switching node in which the inputs are crossed over and sent out.

Now, the number of switches required in each stage is p/2 as a switching element takes two inputs and return two outputs. Thus, since there are p inputs, p/2 switches are required in each stage. Thus, the total number of switches required is log(p) * p/2 = $\Theta(plog(p))$. Recall, that the number of switches required in complete cross bar network scaled as $\Theta(p^2)$ and the number of switching nodes in case of a bus-based network scaled as $\Theta(p)$. Thus, $\Theta(p) < \Theta(p \log p) < \Theta(p^2)$. This, proves that the multistage network is more scalable than the cross bar network in terms of cost and more scalable than the bus network in terms of performance.

Omega Network

Consider the complete omega network as shown in the figure 3.7. We now understand the routing scheme of the omega network.

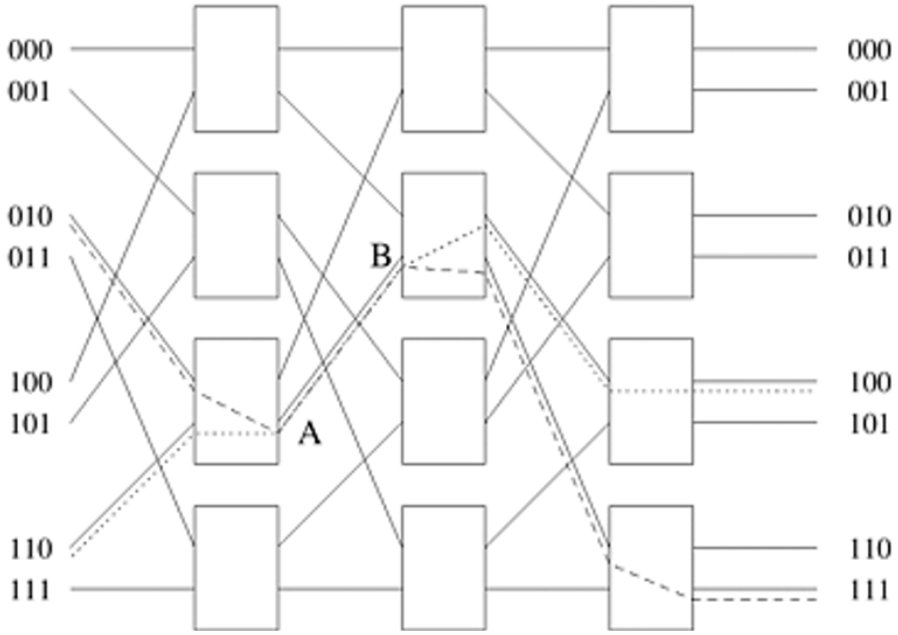


Figure 3.7: Omega Network

Let s be the binary representation of processors that need to write some data to bank t. Now the bits are written in the binary representation. The most significant bit is the left most bit and the least significant bit is the right most bit. Now the routing

scheme works as follows, at the first switching node, if the most significant bits of s and t are the same, then the data is routed in pass-through mode. If bits are different than the data is routed through cross-over mode. At the second switching node, if the second most significant bits of s and t are the same, then the data is routed in pass-through mode. If bits are different than the data is routed through cross-over mode. This is done for all the bits of s and t, by repeating over next switching stage using the next most significant bit. In this way, it uses all log p bits in the binary representation of s and t.

For example, consider a message to be passed from 010 to 111. As shown in the figure 3.7 by dashed lines. As the message reaches the first switching elements because the most significant bits of 010 and 111 are different the message is routed through cross-over mode. As the message reaches the second switching elements because the second most significant bits of 010 and 111 are the same it is routed through pass-through mode. As the message reaches the third switching elements because the third most significant bits of 010 and 111 are different the message is routed through cross-over mode. Thus, the message is passed from 010 to 111.
Consider another example, consider a message to be passed from 110 to 100. As shown in the figure 3.7 by solid lines. As the message reaches the first switching elements because the most significant bits of 110 and 100 are the same the message is routed through pass-through mode. As the message reaches the second switching elements because the second most significant bits of 110 and 100 are different it is routed through cross-over mode. As the message reaches the third switching elements because the third most significant bits of 110 and 100 are the same the message is routed through pass-through mode. Thus, the message is passed from 110 to 100.
Now consider the case where both of the messages in the above examples were to be passed from 010 to 100 and 110 to 111 simultaneously. As shown in the figure 3.7 by solid and dashed lines. In the case when the processor two and six are communicating simultaneously then one may disallow access to another memory bank the another processor. This is because they have a common link in the routing scheme shown by AB in the figure 3.7. This property is called **blocking networks**.

3.1.4 Completely Connected Networks

In this, each node has a direct communication link to every other node in the network. A node can send a message to another node in a single step, since a communication link exists between them. It is a static counter part of crossbar switching networks as the connection between any input/output pair does not block communication between any other pair. For example, a completely connected network of eight nodes is as shown in the figure 3.8.

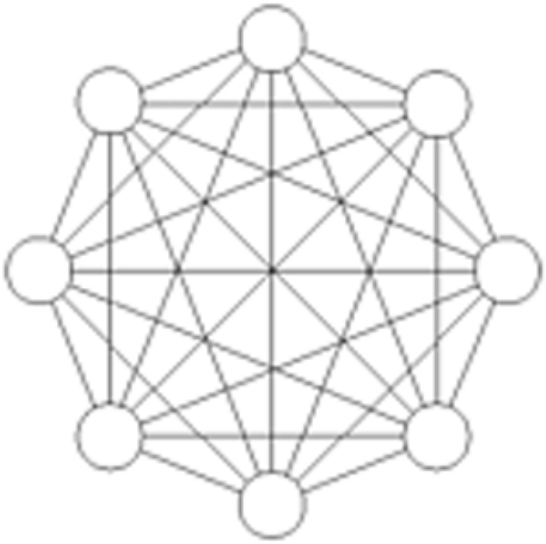


Figure 3.8: Completely Connected Network

Star Connected Network/Star Topology

In this, all the nodes are connected to a central node. The central node acts as a switch to connect the nodes.

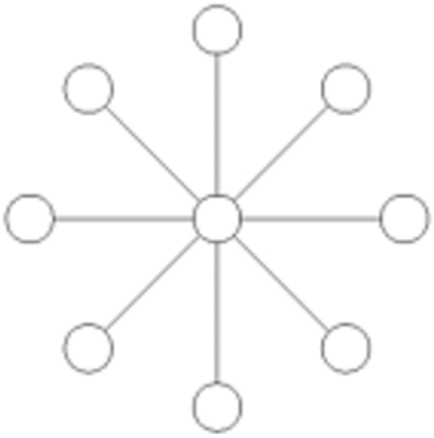


Figure 3.9: Star Connected Network

As shown in the figure 3.9. Every other processor has a communication link connecting it to this processor. It is thus similar to a shared bus. Here, the central processor thus acts as a bottleneck.

3.1.5 Linear Arrays, Meshes and k-d Meshes

Linear arrays are as shown in figure 3.10. It is a one-dimensional array of processing nodes. Each node is connected to its immediate neighbours.

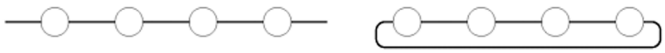


Figure 3.10: Linear Array

As shown in the figure 3.10 there are two possible cases, the left figure shows a linear array with no wraparound links. It is a static network in which each node (except the two nodes at the ends) has two neighbors, one each to its left and right. The right figure shows a linear array with wraparound links also called a 1D torus. The wraparound links connect the last node to the first node and the first node to the last node. The ring has a wraparound connection between the extremities of linear array. Each node has two neighbors in this case.

2D Mesh

It is a linear array extended to 2D as shown in the figure 3.11

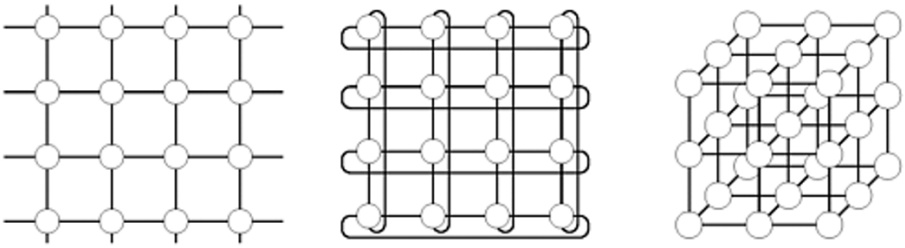


Figure 3.11: Mesh

It has $\sqrt{p}$ nodes in each direction. As shown in the figure there are three possible cases for a 2D mesh. The leftmost figure shows a 2D mesh with no wraparound links. In this each (i,j) node is connected to (i+1,j),(i,j+1),(i-1,j),(i,j-1). It can be laird out in 2D space. A variety of regulatory structure computations map naturally to 2D. 2D mesh were often used as interconnects in parallel machines.

The middle figure shows a 2D mesh with wraparound links. It is also called a 2D Tori. The rightmost figure is a 3D Cube with no wraparound which is a generalization of 2D mesh to 3D. In this, each node is connected to six other nodes, two along each of the three dimensions. 3D simulations can be mapped naturally to 3D.

General Class of k-d meshes

It is a class of topologies with d dimensions and k nodes along each dimension. Thus, in total kd nodes. A 1D linear arrays is a special case of a k-d mesh with d=1. A 2D mesh is a special case of a k-d mesh with d=2. A hypercube has two nodes along each dimensions and has $\log_2 p$ dimensions. It is thus, written as 2-log mesh. A d-dimensional hypercube is constructed by connection two (d-1) dimensional hypercube as shown in the figure 3.12.

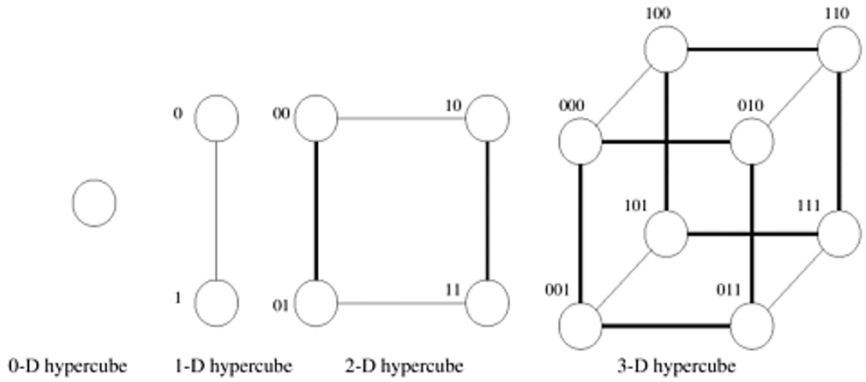


Figure 3.12: Hypercube

The hypercubes follow a numbering scheme which is useful later on for writing many parallel algorithms. Number the nodes then place the two d-1 dimensional (p/2 nodes) hypercubes side by side. Prefix one with 0 and the other with 1. **Property: The minimum distance between two nodes is given by the number of bits that are different in two labels**. For example, 0110 and 0101 are different in two bits. the third and fourth place bits. Thus, the minimum distance between them is two link apart. This property can be used for deriving parallel algorithms for hypercube architecture. A 4D hypercube with 16nodes is as shown in the figure 3.13

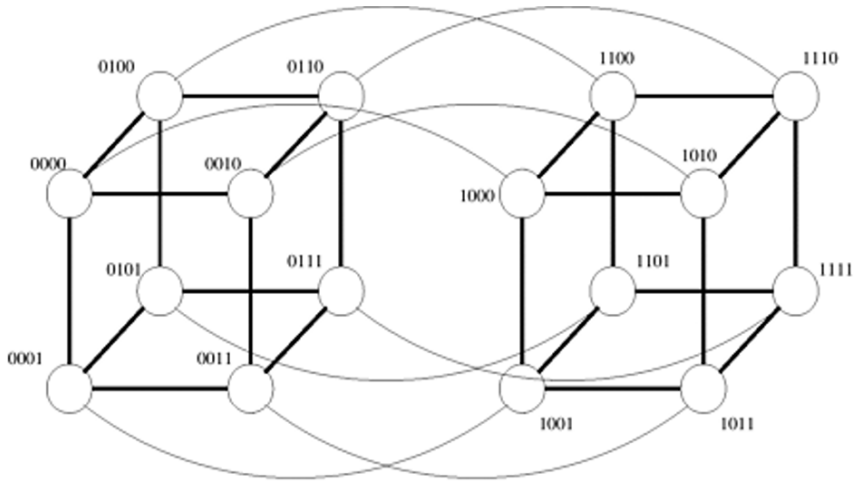


Figure 3.13: 4D Hypercube

3.1.6 Tree Based Networks

It is a hierarchical network. It is a tree with a root node and each node has a parent and children as shown in the figure 3.14.

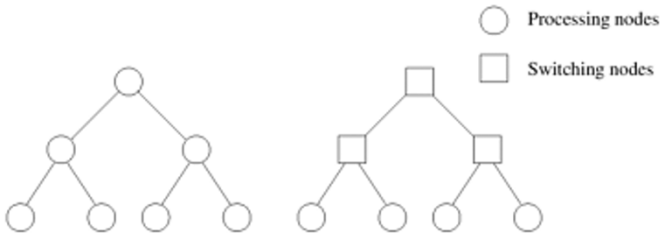


Figure 3.14: Tree Based Network

Note that there is only one path between any pair of nodes. IN the figure 3.14, the left figure shows a Static Tree Network where each processing element is at each node of the tree. The right figure shows a Dynamic Tree Network where the nodes at intermediate level are switching nodes, and the leaf nodes are processing elements.
Routing: Source sends the message up the tree until it reaches the node at the root of the smallest sub tree containing both the source and destination nodes. Then the message is routed down the tree towards the destination node. At higher levels of tree, example, nodes in left sub tree of a node communicate with nodes in right tree, root node must handle all the messages. This leads to communication bottleneck. It is solved by dynamic tree called Fat Tree as shown in figure 3.15 by increasing the number of connection links and switching nodes close to the root.

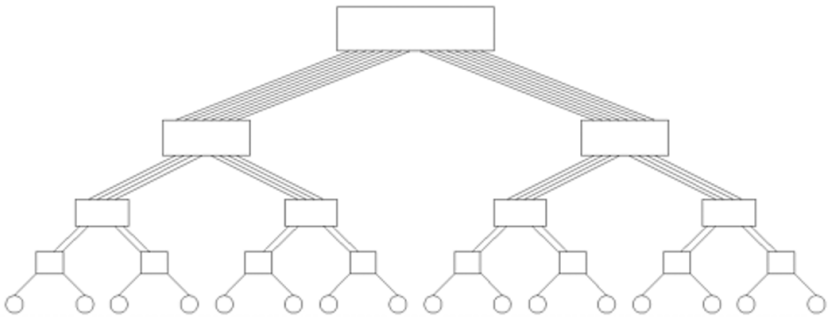


Figure 3.15: Fat Tree with 16 Processing Nodes

The processors are arranged in leaves and all the other nodes correspond to switches. Note that here the property that the number of links from a node to a children is equal to the number of links from the node to its parent. Thus, the edges become fatter as we traverse up. Now any pair of processors can communicate without contention : non-blocking network. It has a constant bisection bandwidth networks as the number of links crossing the bisection is constant. As shown in the example figure 3.15 a two level fat tree has a diameter of four.

3.2 Evaluating Static Networks

The performance of a static network is evaluated in terms of cost, performance and scalability.

3.2.1 Diameter

The diameter of a network is the maximum distance between any two processing nodes in the network. The distance between any two processing nodes is defined as the shortest path (in terms of number of links) between them. For example, the diameter of a completely connected network is 1 and of the star-connected network is two. The diameter of a ring network is $\lfloor p/2 \rfloor$. The diameter of a 2D mesh without wraparound connection is $2(\sqrt{p}-1)$. The diameter of a 2D mesh with wraparound connection is $2\lfloor \sqrt{p}/2 \rfloor$. The diameter of a complete binary tree is $2\log(\frac{p+1}{2})$ because two communicating nodes may be in separate subtree of the root node and a message might have to travel all the way to the root and then down the other subtree.

3.2.2 Connectivity

The connectivity of a network is a measure of the multiplicity of paths between any two processing nodes. A network with high connectivity is desirable since it lowers latency of communication resources.
Arc Connectivity of the network: The minimum no. of arcs that must be removed from the network to break it into two disconnected network. For example, Arc connectivity of Linear array, tree star connected networks is 1. For rings, 2D meshes without wraparound is 2. It is 4 for 2D meshes with wraparound. It is d for d-dimensional hyper cubes.

3.2.3 Bisection Width and Bisection Bandwidth

Bisection Width: The bisection width of a network is defined as the minimum no. of communication links that must be remove to partition into two equal halves. For example, bisection width of ring=2. Bisection width of 2D p node mesh without wraparound = $\sqrt{p}$. For a 2D mesh with wraparound bisection width = $2\sqrt{p}$. Bisection width of a hypercube, but deconstructing the connection we did to get from $2(d-1)$ dimensional hypercube to 1d dimensions hypercube. Hence, p/2 nodes to be cut to separate into two subcubes.
Channel Width: No. of bits that can be communicated simultaneously over a link connecting two nodes. Channel width = no. of physical wires in each communication link. **Channel rate:** The peak rate at which a single physical wire can deliver bits is called the channel rate.
Channel bandwidth: Peak rate at which data can be communicated between the ends of a communication link. Thus,

$$\text{Channel bandwidth} = \text{Channel Width} \times \text{Channel rate} \tag{3.2}$$

Bisection bandwidth: The minimum volume of communication allowed between any two halves of the network.

$$\text{Bisection bandwidth/Cross section bandwidth} = \text{Bisection Width} \times \text{Channel width} \tag{3.3}$$

It is also a measure of cost as it provides a lower bound on areas in 2D packing and volume in 3D. Say bisection width =w, lower bound on area in 2D packaging gives $\Theta(w^2)$ and for volume in 3D packaging is $\Theta(w^{3/2})$. According to it, the hypercubes and completely connected networks are more expensive.

3.2.4 Cost

Number of communication links or number of wires required by the network. For example, linear array =p-1 links to connect p nodes. For, d-dimensional wraparound mesh =dp links. For a hypercube, $\frac{plog}{2}$ links.

3.3 Summary

All of this has been summarise in the following table 3.1.

Network	Diameter	Bisection Width	Arc connectivity	Cost
Completely connected	1	$p^2/4$	p-1	$\frac{p(p-1)}{2}$
Star	2	1	1	p-1
Ring	$\lfloor p/2 \rfloor$	2	2	p-1
Complete Binary Tree	$2\log(\frac{p+1}{2})$	1	1	$p-1$
Linear Array	p-1	1	1	p-1
2D mesh (no wraparound)	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2\sqrt{p}(\sqrt{p}-1)$
2D mesh (wraparound)	$2\lfloor\sqrt{p}/2\rfloor$	$2\sqrt{p}$	4	2p
Hypercube	$\log p$	$\frac{p}{2}$	$\log p$	$\frac{p \log p}{2}$
Wraprwound k-array d cube	$d\lfloor k/2 \rfloor$	$2k^{d-1}$	$2d$	dp

Table 3.1: Summary of Static Networks

For more details on Interconnection networks for Parallel computers refer section 2.4 in Kumar et al. [1994].

Chapter 4

Parallelization Principles

Parallel programs incur overheads that are not present in sequential programs. These overheads include:

- **Communication Overhead** – Time taken to communicate between processors.
- **Synchronization Overhead** – Time taken to synchronize among processors.
- **Idling Overhead** – Time a processor spends waiting for other processors to reach the same execution point.

A well-designed parallel program aims to minimize these overheads.

4.1 Evaluation of Parallel Programs

The performance of a parallel program is evaluated using the following metrics:

- **Execution Time** (T_p) – The total time taken by the parallel program to complete execution.
- **Speedup** (S) – The ratio of the time taken by the *best* sequential program to the time taken by the parallel program:

$$S(p, n) = \frac{T(1, n)}{T(p, n)}$$

  where $T(1, n)$ is the execution time of the best sequential algorithm for a problem of size n, and $T(p, n)$ is the time taken by the parallel algorithm using p processors.

  Ideally, we expect a speedup of p, meaning the parallel execution time should be reduced by a factor of p. However, in practice, $S(p, n) < p$ due to the overheads introduced by communication, synchronization, and processor idling.

In some cases, we may even get $S(p,n) > p$. This is called **superlinear speedup**. A large dataset in a sequential program may not fit in the cache, resulting in many cache misses. However, in a parallel program, the data is distributed among the processors, and each processor deals with a smaller subset of the data. This smaller subset can often fit into the individual processor's cache, leading to fewer cache misses.
Therefore, by decomposing the problem across p processors, each processor only requires a fraction of the total data. If each processor's working data fits into its cache, cache efficiency improves significantly. This leads to better-than-expected performance, and the speedup observed can exceed p. When this happens, we say that the program exhibits **superlinear speedup**.

- **Efficiency** – While speedup tells us how much faster the program runs in parallel, it does not indicate how effectively the resources are being utilized. For example, a speedup of 3 can be achieved using either 3 processors or 100 processors.

  To quantify this, we define **efficiency** as the speedup normalized by the number of processors:

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

  Efficiency tells us how well the program utilizes the available processors. A low efficiency implies that parallelization overheads (communication, synchronization, or idling) are dominating the execution time. Generally, $E(p,n) < 1$, but in rare cases where superlinear speedup occurs, efficiency can exceed 1.

- **Scalability** – This refers to the ability of a parallel program to maintain efficiency as the number of processors increases. Scalability is crucial because we often wish to solve larger problems using more computational resources.

  Scalability analysis involves studying how the speedup and efficiency behave with:

  - increasing number of processors p, or
  - increasing problem size n.

  A well-scaled program maintains high efficiency as we increase p and/or n. Poor scalability indicates that the program cannot effectively utilize more processors due to increasing overheads or limited parallelism.

  Ideally:

  - Speedup $S(p,n)$ should increase linearly with p.
  - Efficiency $E(p,n)$ should remain constant with increasing p.

  However, in practice:

  - Speedup is typically sub-linear due to overheads.
  - Efficiency usually decreases as the number of processors increases.

This behavior is captured by **Amdahl's Law**.

Amdahl's Law: The performance improvement achievable through parallel execution is limited by the fraction of the program that must remain sequential. In the context of parallel programming, Amdahl's Law states that the speedup of a program is fundamentally constrained by the portion that cannot be parallelized.

Let t_s denote the execution time of a sequential program. Suppose a fraction f_s of the program is inherently sequential (non-parallelizable), and the remaining fraction $f_p = 1 - f_s$ is perfectly parallelizable. Then, the execution time of the program on p processors can be expressed as:

$$t_p = f_s t_s + \frac{f_p t_s}{p}$$

The speedup $S(p)$ achieved by using p processors is defined as the ratio of the sequential execution time to the parallel execution time:

$$S(p) = \frac{t_s}{t_p} = \frac{t_s}{f_s t_s + \frac{f_p t_s}{p}} = \frac{1}{f_s + \frac{f_p}{p}}$$

Thus, the speedup is fundamentally limited by the sequential portion f_s of the code. No matter how many processors are used, the non-parallelizable part becomes the bottleneck, preventing infinite speedup.

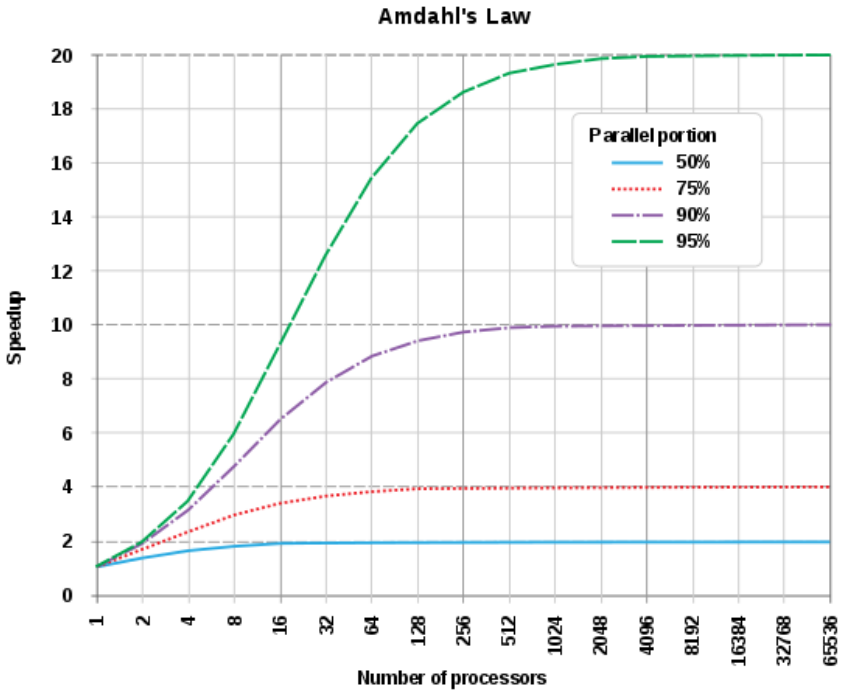


Figure 4.1: Amdahl's Law: Speedup vs. Number of Processors for Different Parallel Fractions

As shown in Figure 4.1, even for highly parallelizable programs (e.g., $f_p = 0.95$), the speedup curve begins to flatten as the number of processors increases. This illustrates that beyond a certain point, adding more processors yields diminishing

returns due to the fixed cost of the sequential part. This effect becomes more prominent as p grows large, clearly demonstrating the limitations imposed by Amdahl's Law.

The Amdahl's Law assumes that the problem size is fixed even on increasing the number of processors, thus the fractions f_s and f_p remain constant. But, in reality, the problem size often increases with the number of processors. Thus, Gustafson's Law was proposed to address this issue.

Gustafson's Law: Increase the problem size proportionally so as to keep the overall time constant. Here, the proportional increase means that the order of computations should increase proportionally to the number of processors, in order to keep the overall execution time approximately constant. It does not mean simply increasing the input size linearly.

Strong Scaling: The scaling behavior observed when the problem size is kept constant while the number of processors increases is known as *strong scaling*. This is the perspective assumed by Amdahl's Law.

Weak Scaling: The scaling behavior observed when the problem size increases proportionally with the number of processors, in order to maintain constant workload per processor, is called *weak scaling*. This is the perspective taken in Gustafson's Law.

- **Isoefficiency:** Generally, the efficiency of a parallel program decreases with an increasing number of processors P and increases with a growing problem size N. Isoefficiency measures the ability of a program to maintain constant efficiency as both the number of processors and the problem size increase.

  For example, consider the formula for the parallel Gaussian Elimination routine used in the LAPACK library for solving linear systems:

$$T_{\text{par}}(N,P) = \frac{2}{3}\frac{N^3}{P}t_f + \frac{(3+\frac{1}{4}\log_2 P)N^2}{\sqrt{P}}t_v + (6+\log_2 P)Nt_m$$

$$T_{\text{seq}}(N) = \frac{2}{3}N^3 t_f$$

$$E = \frac{T_{\text{seq}}(N)}{PT_{\text{par}}(N,P)} = \left(1 + \frac{3}{2}\frac{\sqrt{P}(3+\frac{1}{4}\log_2 P)}{N}\frac{t_v}{t_f} + \frac{3}{2}\frac{P(6+\log_2 P)}{N^2}\frac{t_m}{t_f}\right)^{-1}$$

  where:

  - t_f is the time per floating-point operation,
  - t_v is the time per vector operation,
  - t_m is the time per memory operation.

  From the dominant terms in the efficiency expression, we observe that as P increases, the problem size N must also increase to maintain constant efficiency. Specifically, we find that $N = O(\sqrt{P})$ suffices. Since the computational complexity of Gaussian Elimination is $O(N^3)$, the corresponding isoefficiency function becomes:

$$\text{Isoefficiency} = O(P\sqrt{P})$$

The isoefficiency function thus captures the growth rate of the problem size required to maintain efficiency with increasing processors. A program with a smaller isoefficiency function scales better. That is, it requires fewer additional processors for a given increase in problem size to preserve performance.

For instance, consider two parallel programs with isoefficiency functions $W_1 = O(P)$ and $W_2 = O(\sqrt{P})$. The second program is more scalable since it needs fewer processors for the same increase in workload.

In summary:

- A smaller isoefficiency function indicates better scalability.
- Algorithms with linear or sub-linear isoefficiency (e.g., $O(P)$ or $O(\sqrt{P})$) are considered highly scalable.
- Algorithms with quadratic or exponential isoefficiency functions are poorly scalable.

For further details refer to sections 5.1 to 5.6 in Kumar et al. [1994].

Chapter 5

Parallel Programming Classification and Steps

5.1 Parallel Program Models

Classification based on the way the program and data are written and executed:

- **Single Program Multiple Data (SPMD):** A single program is written and run on multiple processors. Each processor runs the same program but operates on different data using processor IDs to determine which part of the data to handle. It is the most common model used in parallel programming.
- **Multiple Program Multiple Data (MPMD):** Different processors run different programs on different data. It is mainly used in distributed memory systems. For example, it is used in climate modeling. However, it is not a very popular model in general-purpose parallel computing.

5.1.1 Programming Paradigms

Classification based on how the parallel program handles memory:

- **Shared Memory Model:** A common global memory is used to access data among multiple processes or threads. Examples include Threads, OpenMP, and CUDA.
- **Message Passing Model:** No global memory is shared. Processes explicitly send and receive data when required. The most common example is MPI (Message Passing Interface).

5.2 Parallelizing a Program

There is no universal rule for parallelizing a program, as it heavily depends on the specific application. Given a sequential program or algorithm, there are four general steps to produce a parallel version:

- **Decomposition:** Identify parallel tasks with a high degree of potential concurrent activity. This involves splitting the problem into tasks that can be executed in parallel.

- **Assignment:** Assign the tasks to processors by grouping them into processes with balanced workloads. That is, each process should ideally perform an equal amount of work. Assignment is not necessarily part of the decomposition step. Instead, it focuses on grouping tasks for each process to achieve load balance while minimizing communication and management overhead.

  This step is often performed using structured methods such as code inspection (e.g., parallel loops) or domain knowledge of the application. Grouping can be done statically (at compile time) or dynamically (at runtime) depending on the load characteristics.

  Both decomposition and assignment are typically independent of the underlying architecture or programming model. However, the cost and complexity of using different primitives can influence these decisions.

- **Orchestration:** Manage the execution of the parallel program. This involves handling synchronization and communication between processes or threads to ensure correctness and performance.

- **Mapping:** Map the processes to physical processors. The goal is to minimize communication overhead by efficiently placing processes on processors, especially in distributed-memory systems.

5.2.1 Data Parallelism and Data Decomposition

Data Parallelism: In this approach, the given data is divided across the processing entities. Each process *owns* and *computes* on a portion of the data, and determines which data to communicate with other processes. This is often referred to as the *owner computes* rule. The data is partitioned into chunks, and each chunk is assigned to a process. Thus, parallelism is driven by the data distribution rather than the task structure.

Domain Decomposition: In simulation-based applications, the multi-dimensional domain is divided into subdomains corresponding to the number of processing entities. This technique is known as domain decomposition. The P processes are arranged into a multidimensional *process grid*.

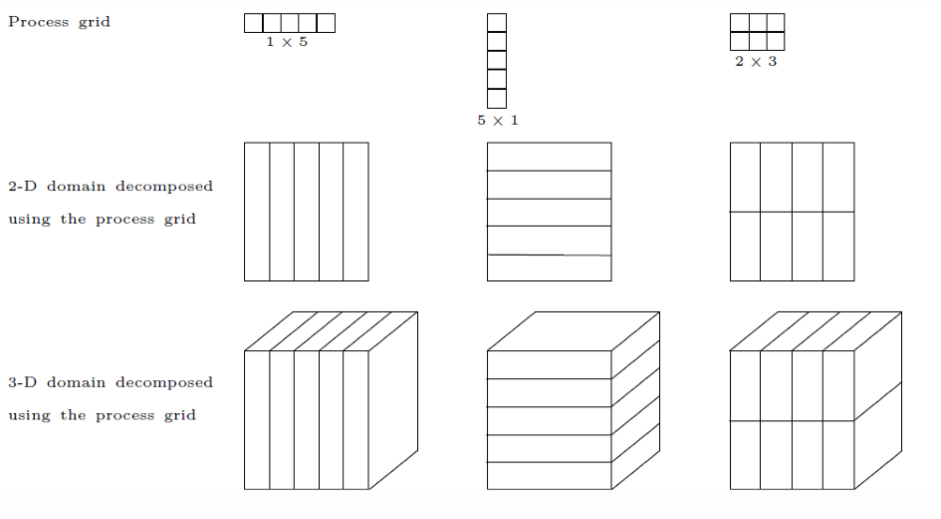


Figure 5.1: Domain Decomposition

As shown in Figure 5.1, if the process grid is arranged as a 1×5 grid, then a two-dimensional domain can be decomposed into five equal parts. Each process is assigned one subdomain. The decomposition is performed in such a way that inter-process communication is minimized.

The same principle applies to higher-dimensional domains. Figure 5.1 illustrates several ways to decompose the domain based on different process grid arrangements. The key objective is to map the computational domain onto the process grid such that communication costs are reduced, while ensuring balanced computational workload across processes.

Data Distributions

To divide data along a particular dimension using processes arranged in that dimension, various data distribution schemes are employed. The most common schemes include:

- **Block Distribution:** The data is divided into contiguous blocks, and each block is assigned to a process. This is the simplest form of distribution and works well when the data is uniformly distributed.

- **Cyclic Distribution:** Data elements are distributed to processes in a round-robin (cyclic) fashion. This is particularly useful when the data is irregular or when load balancing is important.

- **Block-Cyclic Distribution:** A hybrid scheme combining block and cyclic distributions. Data is divided into small blocks, and these blocks are then distributed cyclically among the processes. This offers both improved load balancing and spatial locality.

An example is illustrated in Figure 5.2.

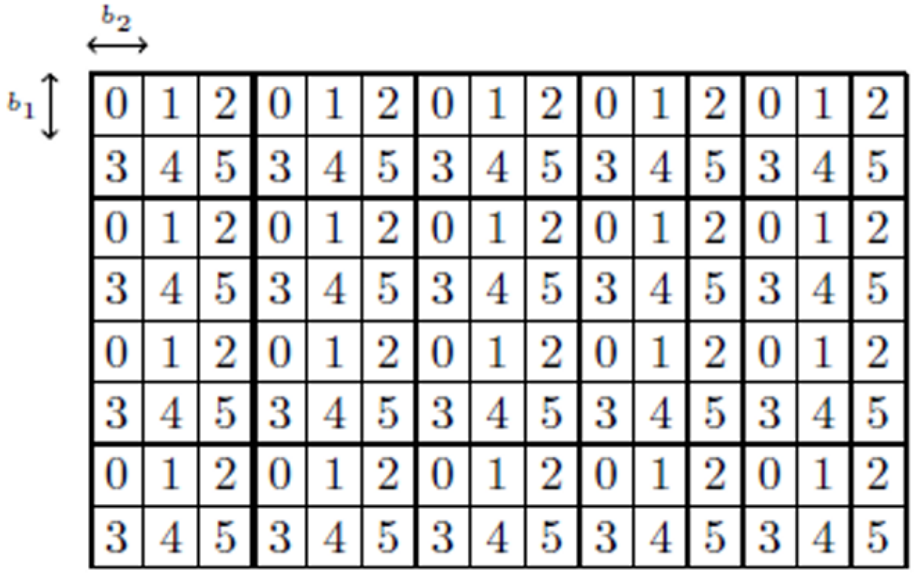


Figure 5.2: Illustration of Data Distribution Schemes

In this figure, the process grid is arranged as a 2×3 grid, and the data blocks are distributed in a cyclic manner. Each process receives a portion of the data based on the chosen distribution scheme, and the pattern of assignment directly influences the communication and computation balance in parallel applications.

Task Parallelism

In task parallelism, a problem is decomposed into a set of tasks that can be executed concurrently. This approach is especially useful when the data is not uniformly distributed, or when the computation cannot be easily partitioned based on data.

Independent tasks are identified and assigned to different processors through a process called *mapping*. The objectives of task mapping are twofold:

1. To balance the workload across all processors.
2. To minimize inter-task communication by reducing inter-group dependencies.

This decomposition is typically represented using a *task graph*, also known as a *task dependency graph*, as shown in Figure 5.3.

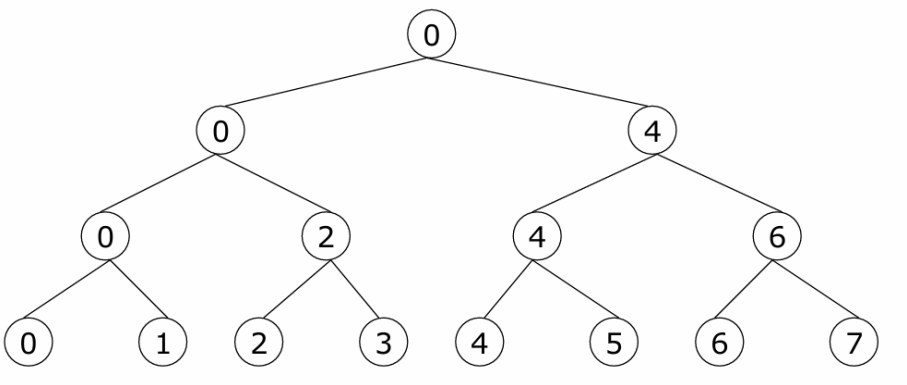


Figure 5.3: Task Dependency Graph

Each node in the graph represents a task, which may correspond to a statement or a block of statements in a program. The edges indicate dependencies between tasks—i.e., one task must be completed before another can begin.

Consider the task graph in Figure 5.3, which follows a tree-like structure. The eight leaf nodes at the bottom are independent and can be executed in parallel. These are then merged in successive levels: 8 tasks combine to form 4, then 2, and finally a single task at the root.

Assume we have 8 processors available. We can assign the 8 independent tasks at the leaf level to the 8 processors. When two tasks need to be merged, the computation can be assigned to one of the processors already involved with the previous tasks. This leverages data locality—one processor already holds partial results, and only the additional data from the other processor needs to be communicated, thereby reducing communication overhead.

In general, the problem of optimally assigning tasks from a task graph to processors—such that both load is balanced and communication is minimized—is NP-complete. Therefore, heuristic approaches are typically employed to generate efficient mappings.

5.2.2 Orchestration

Orchestration is the process of managing the execution of a parallel program. It involves coordinating synchronization and communication between processes to ensure correctness and efficiency.

Maximizing Data Locality

This refers to executing computations such that the data required is already available locally in the processor, thus reducing communication overhead. Orchestration aims to:

- **Minimize the volume of data exchange** — e.g., avoid sending intermediate results unnecessarily.
- **Minimize the frequency of interactions** — e.g., use techniques such as packing to reduce communication calls.

Consider computing the dot product of two vectors $\boldsymbol{A}$ and $\boldsymbol{B}$. The operation involves multiplying corresponding elements and summing the results. The multiplication step is inherently parallelizable — each processor can independently compute partial products.

For the summation step, we have two options:

- Each processor sends its partial product to a master process, which performs the summation.
- Each processor computes its partial sum locally, and only the final results are sent to the master process for aggregation.

The second option is more efficient, as it reduces the volume of data communicated.

Point-to-point communication between processors involves two important costs:

- **Latency** — the time to establish the communication connection.
- **Bandwidth** — the rate (e.g., MB/s) at which data is transferred.

To illustrate how orchestration can minimize these costs, suppose a processor needs to send 10 arrays to another processor. There are two options:

1. Send each array individually. This incurs the latency cost 10 times.
2. Pack all 10 arrays into a single message and send them together. This incurs the latency cost only once.

While the total data volume remains the same, the second method reduces the number of communications, thereby reducing total latency. This strategy is known as **packing**. It significantly improves performance by minimizing the frequency of interactions.

Minimizing contention and hotspots

(Avoiding communication bottlenecks): Avoid using the same communication pattern across all processes. For a given communication pattern, following one order may result in severe contention and degraded performance, while a different ordering might alleviate these issues.

Consider a situation where every process needs to communicate with every other process. This is illustrated in Table 5.1.

Process	A	B	C	D	E
A	-	2	3	4	5
B	1	-	3	4	5
C	1	2	-	4	5
D	1	2	3	-	5
E	1	2	3	4	-

Table 5.1: Communication Pattern

In this example, assume that communication proceeds over discrete time steps t. At each time step t, each processor with ID i attempts to communicate with processor t if $t \neq i$.

For instance, at $t = 0$:

- Processor A ($id = 0$) does not communicate, since $t = id$.
- Processor B ($id = 1$) communicates with processor A.
- Processor C communicates with processor A.
- Similarly, processors D and E also communicate with processor A.

At $t = 1$:

- Processor A communicates with B.
- Processor B remains idle.

- Processor C communicates with B.
- Processors D and E also communicate with B.

Observe the pattern: at time t, all processes except the one with $id = t$ attempt to communicate with processor t. This results in a serious communication bottleneck—one processor becomes the hotspot and is overwhelmed with incoming messages. Even at the hardware level, switches and network links typically support only one transmission at a time, further compounding the issue.

Now, consider scaling this to 1000 processes. At some time step t, 999 processes will attempt to send messages to a single process, creating a massive bottleneck.

Solution: Design an algorithm where each process follows a distinct communication schedule, such that no single process becomes the target of all communications at any given time. This strategy is called **minimizing contention and hotspots**. It avoids synchronized communication patterns that lead to bottlenecks and ensures more balanced use of the network.

Overlapping computations with interactions

One of the key techniques to improve performance in parallel programs is **overlapping computations with interactions**. The idea is to split the computations into two categories:

1. Computations that depend on communicated data (Type 1).
2. Computations that do *not* depend on communicated data (Type 2).

While waiting for communication required for Type 1 computations to complete, a process can proceed with Type 2 computations that are independent of the incoming data. This approach can significantly reduce idle time by ensuring useful work is being done during communication delays.

This strategy is especially effective on the *receiver side*, where computations that do not depend on the received data can be scheduled during the communication phase. In other words, communication is being *hidden* behind computation, thereby increasing efficiency.

Replicating data or computations

Another technique to reduce communication overhead is **replicating data or computations**. This method balances the additional cost of computation or storage against the gain from avoiding expensive communication.

Consider two processes, Process 0 and Process 1. Suppose Process 0 computes a matrix A using the outer product of two vectors a and b, and then sends part of the matrix A to Process 1. After receiving it, Process 1 performs its computations.

Instead, an alternative strategy is to replicate vectors a and b to Process 1. Now, both processes can independently compute matrix A, after which they operate on their respective portions. This significantly reduces communication cost, since transmitting vectors a and b (of size n) is much cheaper than sending portions of matrix A (of size $n \times n$), assuming n is large.

Although this increases the computational burden (both processes now compute A independently), the gain in communication efficiency often outweighs the extra

computation. This technique is particularly useful in high-performance settings where communication costs dominate.

Ultimately, the decision to replicate data or computations depends on the trade-off between communication and computation costs, and it should be made based on profiling and performance analysis.

5.2.3 Mapping

Mapping refers to assigning processes to specific processors — essentially deciding which process runs on which processor.

- **Network topology and communication pattern:** Mapping can depend heavily on the network topology and the communication pattern among processes. An intelligent mapping of the communication pattern to the network topology can significantly impact the performance of the parallel program.

  For example, consider a 2D mesh network topology where processes are arranged in a 2D grid and each process communicates with its neighboring processes. In such a case, we prefer a *natural mapping* — that is, mapping processes that communicate with each other to processors that are physically close to one another. This minimizes communication overhead by reducing the distance over which data must travel, leading to faster communications.

- **Heterogeneous systems:** Mapping also depends on processor speeds, especially in heterogeneous systems, where processors may have different computational capabilities. In homogeneous systems, all processors have the same speed and uniform communication cost.

  In contrast, in heterogeneous systems, both computation speeds and communication costs vary across processors. Thus, mapping strategies must consider these variations.

  For example, consider a system with two processors, A and B, where A is faster than B. If the communication cost between A and B is lower than the computation time required by B, it is better to assign the task originally intended for B to A. This is because communication in this case is cheaper than letting the slower processor handle the computation. As a rule, heavy compute tasks should be mapped to faster processors, while lighter tasks can be assigned to slower processors.

All data and task parallel strategies generally follow *static mapping*, i.e., the mapping of tasks to processors is done at the beginning of the program and remains constant throughout the execution.

However, in some cases, *dynamic mapping* is employed, where the mapping is performed at runtime. This is useful in scenarios where the load on processors is non-uniform and may vary over time. In such situations, dynamic mapping is handled by a master processor, which allocates tasks based on the current load on each processor.

The idea is that a global memory or a managing process holds a pool of tasks. The master processor distributes some tasks initially to all the worker processors. Once a processor completes its assigned tasks, it requests more work from the coordinator

or master. This technique is referred to as **self-scheduling** or **work stealing**, where tasks are scheduled dynamically based on processor load.

Here, the tasks are not statically bound to specific processors; instead, the processors dynamically determine which tasks to execute next. As a result, in heterogeneous systems, faster processors can finish their workload sooner and "steal" additional work from slower processors. This helps balance the load more effectively and improves overall performance.

To summarize, the various steps in parallel program execution are shown in Table 5.2.

Step	Architecture-Dependent?	Description
Decomposition	Mostly no	Identifying parallel tasks with a high potential for concurrency without excessive overhead.
Assignment	Mostly no	Balancing the workload and reducing communication volume by assigning tasks to processors efficiently.
Orchestration	Yes	Minimizing non-inherent communication through data locality; reducing communication and synchronization costs; managing execution while satisfying task dependencies; avoiding serialization at shared resources.
Mapping	Yes	Assigning related processes to the same processor if necessary; leveraging network topology to reduce communication overhead by efficiently mapping processes to processors.

Table 5.2: Summary of Parallel Program Execution Steps

As seen in Table 5.2, the *Decomposition* and *Assignment* steps focus on identifying independent tasks and distributing them evenly. Once the tasks and communication patterns are established, the *Orchestration* step aims to minimize communication overhead, while the *Mapping* step maps tasks to processors in a way that reduces runtime communication cost.

The *Orchestration* and *Mapping* steps are architecture-dependent, as they rely on the system's network topology and communication characteristics. On the other hand, *Decomposition* and *Assignment* are mostly architecture-independent and are typically handled at the programming level.

5.2.4 Example

Given a 2D array of float values, we wish to repeatedly average each element with its immediate neighbours until the difference between two iterations is less than a specified tolerance value. Consider a 2D domain discretized into a grid of points. We initialize each grid point with some value, which is then updated iteratively over time.

This is commonly encountered in problems like solving the heat equation, where heat diffuses over a 2D plate. The grid points represent the temperature at different

positions on the plate, and averaging simulates how the heat spreads. We are interested in studying how these temperatures evolve over time.

The pseudocode for the averaging process is shown in Listing 5.1.

```
do {
    diff = 0.0;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++) {
            temp = A[i][j];
            A[i][j] = average(neighbours);
            diff += abs(A[i][j] - temp);
        }
} while (diff > tolerance);
```

Listing 5.1: Parallel Program for Averaging

Consider Figure 5.4, which illustrates the layout of grid points and the neighbours for a typical interior point.

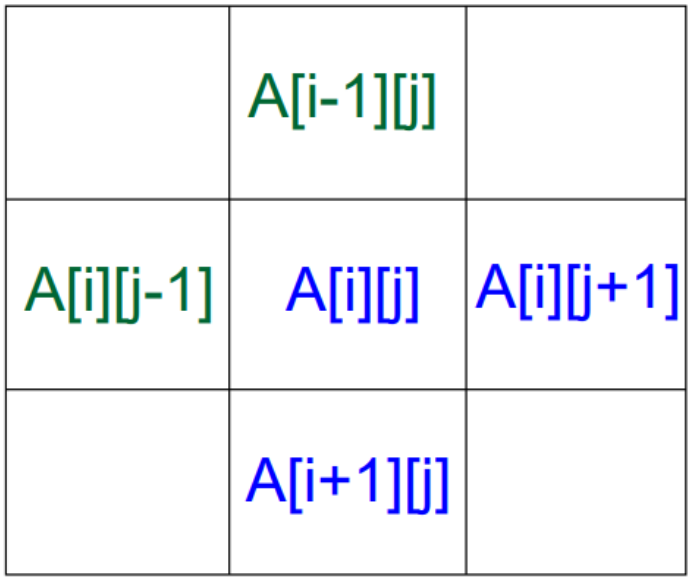


Figure 5.4: Grid points and their neighbours

In each iteration, we sweep through all grid points, update the value at each point by averaging the values of its four (or more) immediate neighbours, and accumulate the absolute difference in the variable `diff`. We continue this process until `diff` falls below a given tolerance.

There are two classical iterative methods for this problem:

- **Jacobi Method:** Uses neighbour values from the previous time step. All grid points are updated simultaneously using an auxiliary array to store the new values.
- **Gauss-Seidel Method:** Uses the most recently available values — i.e., when updating a grid point, the already-updated values from the current iteration (lower index values) are used whenever possible. This usually converges faster but is inherently more sequential.

To parallelize this computation, we need to decompose the grid into sub-regions that can be processed independently by different threads or processes. Each task will be responsible for a subset of grid points. However, care must be taken to handle boundary conditions and ensure proper synchronization or communication between tasks when accessing neighbouring values.

1. One way is to consider each element in parallel. A parallel process or a parallel thread for each of the elements, i.e., a concurrent task for each element update. This would require a maximum concurrency of n^2 tasks (for $n \times n$ elements in the grid). Practically, this is not feasible as the number of tasks is very large, and the overhead of creating and managing the tasks is very high. This many parallel threads is not practically possible, as for larger values of n, the number of threads required will be n^2. Thus, many threads would have to be mapped to the same processors and would require a lot of context switching between threads and processes, which can affect the performance.

2. Another way is to consider tasks for elements in the anti-diagonal. Note that values in a particular anti-diagonal depend on the values from the previous anti-diagonal but not among the elements within the same anti-diagonal. Now we consider a particular anti-diagonal—for some element in the anti-diagonal, using the Gauss-Seidel method, we require the values of the neighbours from the previous anti-diagonal at the latest time step and the value from the next anti-diagonal at the previous time step. This works out, since we are computing anti-diagonal by anti-diagonal, starting from the top-left corner and moving toward the bottom-right corner. Thus, the values of the neighbours are available from the previous anti-diagonal at the latest time step and the next anti-diagonal at the previous time step. Also, note that the values of the elements in the anti-diagonal are independent of each other and thus can be computed in parallel. This is as shown in the Figure 5.5. Therefore, we can consider the tasks for the elements in the anti-diagonal and compute their values in parallel. This way, we can reduce the number of tasks to $2n-1$ (for $n \times n$ elements in the grid). This is a better approach than the previous one, as the number of tasks is reduced and the tasks are independent of each other.

   Note that for the diagonals at the ends, the number of elements will be smaller, and hence the parallelism will also be limited. This approach also incurs synchronization costs because the processes assigned to a particular diagonal cannot start executing until the earlier diagonals have finished executing. Thus, the parallelism is limited by the number of elements in the diagonal and the synchronization overhead.

3. What we follow is block distribution of the data. Consider Figure 5.6, which shows the block distribution of the data.

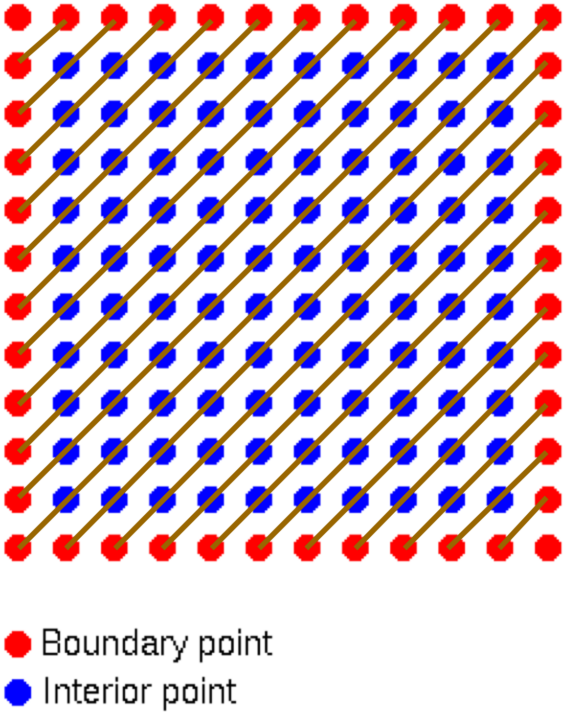


Figure 5.5: Anti-diagonals

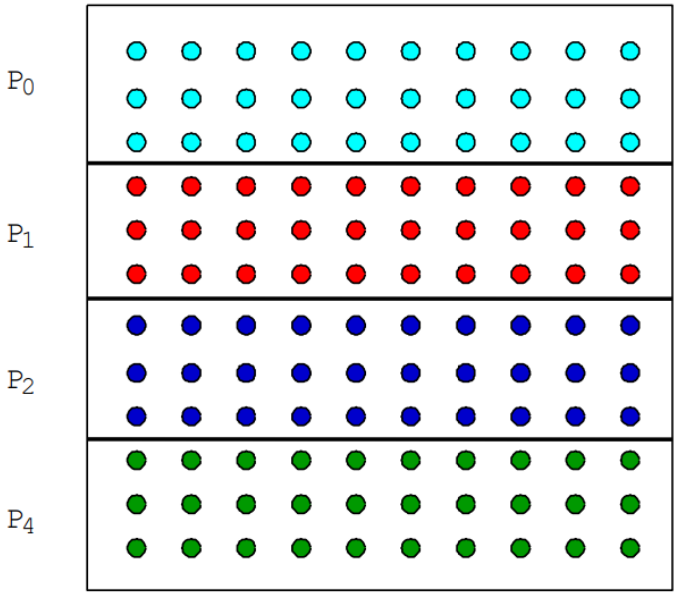


Figure 5.6: Block Distribution

In this approach, we divide the 2D grid across the rows. The first set of rows is assigned to P0, the second set to P1, and so on. Each processor will be responsible for updating the values of the elements in the rows assigned to it.

For the orchestration step, we need to identify the synchronization and communication requirements. The goal is to ensure correctness while minimizing communication and synchronization calls. The implementation details depend on the underlying programming/model architecture—whether it is based on a

shared memory model or a message-passing model.

Shared Address Space / Shared Memory Model

Now we consider the Shared Address Space (SAS) version of the program, as shown in Listing 5.2.

```
int n, nprocs; /* Matrix: (n+2) x (n+2) elements */
float **A, diff = 0;
LockDec(lock_diff);
BarrierDec(barrier1);

main()
begin
    read(n);            /* Read input parameter: matrix size */
    read(nprocs);       /* Read number of processors */

    A = malloc(a 2-D array of (n+2) x (n+2) floats);

    Create(nprocs - 1, Solve, A);  /* Spawn nprocs - 1 processes */

    initialize(A);      /* Initialize the matrix A */

    Solve(A);           /* Main process also solves */

    Wait_for_End(nprocs - 1);  /* Wait for other processes to finish
       */
end main
```

Listing 5.2: Shared Address Space Version

Note that we are using $(n+2)$ grid points to handle boundary conditions, but only `nprocs` processors. The array `A` is shared among all processes. The main process creates the other processes and then continues with solving.

Let us now look at the pseudocode for the `Solve` function.

```
procedure Solve(A)  /* Solve the equation system */
float **A;
begin
    int i, j, pid, done = 0;
    float temp;

    mybegin = 1 + (n / nprocs) * pid;
    myend   = mybegin + (n / nprocs);

    while (!done) do  /* Outer loop over sweeps */
        diff = 0;  /* Initialize difference to 0 */

        Barrier(barrier1, nprocs);  /* Synchronize all processes */

        for (i = mybegin; i < myend; i++) do  /* Sweep over grid */
            for (j = 1; j <= n; j++) do
                temp = A[i][j];  /* Save old value */
                A[i][j] = 0.2 * (A[i][j] + A[i][j-1] + A[i-1][j] +
                   A[i][j+1] + A[i+1][j]);  /* Compute average */
                diff += abs(A[i][j] - temp);
            end for
        end for
```

```
22
23          if (diff / (n * n) < TOL) then
24              done = 1;
25      end while
26  end procedure
```

Listing 5.3: Solve Function

Now, this particular function is executed by all the processes. But depending on the process ID, different processes will handle different parts of the grid. This approach is called the **Single Program Multiple Data (SPMD)** model. The same program is executed by all processes, but each works on a different portion of the data.

The expressions

$$\texttt{mybegin} = 1 + \left(\frac{n}{\texttt{nprocs}}\right) \cdot \texttt{pid}, \quad \texttt{myend} = \texttt{mybegin} + \left(\frac{n}{\texttt{nprocs}}\right)$$

determine the starting and ending rows of the grid that each process is responsible for. These are calculated using the process ID.

All the processes now enter the `while` loop. The matrix `A` and the variable `diff` are shared by all processes, making them global. At the beginning of each iteration, all processes synchronize using a `barrier`, as shown in the code.

Each process then executes the loop that updates a subset of the grid. Although every process runs the same code, they work on different rows depending on their `pid`. The elements in the grid are updated by computing an average, and the difference between the new and old values is added to `diff`.

Since `diff` is a shared variable, it gets updated by all the processes. At the end of the iteration, it is compared with a predefined tolerance to determine whether the loop should terminate.

This was just an overview of the program. To understand the benefits of parallel computing, it's important to appreciate synchronization primitives like the `barrier`.

A `barrier` is a synchronization point. When a process reaches a barrier, it waits there until all other processes have reached the same point. Only after that can any of the processes proceed. In the program above, this ensures that all processes finish their computation for one iteration before moving to the next. Without this, some processes might start the next iteration using stale values from neighboring processes that haven't finished updating their part of the grid.

The matrix `A` is a global variable and is shared among all processes. During execution, each process updates its portion of the matrix. However, issues can arise at the boundaries between two adjacent processors.
For example, if a process updates a boundary element before its neighboring process has finished updating its own value, inconsistent or stale values might be used. This can lead to incorrect results.
One way to avoid this problem, especially in the Jacobi method, is to use a temporary array local to each process. This temporary array holds the updated values for the current iteration based on the global matrix values from the previous step. Once all processes have completed their local updates (ensured via a `barrier` call), the temporary arrays are copied back into the shared matrix `A`, and the next iteration begins.
This ensures that all updates, especially at the boundaries, are synchronized. In our current discussion, we are ignoring this issue for simplicity and assuming that such synchronization is handled internally.

After solving the matrix `A`, the code in Listing 5.2 uses a `Wait_for_End` call. This makes the main process wait until all the other processes have finished. It can be implemented using an *all-to-one* communication pattern, where each worker process sends a message to the main process to signal completion.

Note that the variable `diff` is a global (shared) variable. Since all processes read and write to it, it can lead to incorrect or garbage values due to race conditions. A race condition happens when multiple processes access and update a shared variable at the same time, leading to unpredictable results. This often arises because of context switching at the assembly level.

To solve this, we use **locks**. A lock is a mechanism that ensures only one process can access a shared resource at a time. When a process wants to update the shared variable, it first locks it, then accesses and modifies it, and finally unlocks it. This ensures exclusive access.

In computer science, a **lock** or **mutex** (short for mutual exclusion) is a synchronization primitive. It ensures that only one thread or process can acquire the lock at any given time. All other threads trying to acquire the lock are blocked until the lock is released. This ensures that shared resources are protected and accessed safely.

This concept is called **mutual exclusion** — processes exclude each other, allowing only one to enter the critical section (the part of the code accessing the shared variable). In our case, both `diff` and the matrix `A` are shared variables. So, care must also be taken when processes update boundary elements of the matrix. Those elements may be read and written by neighboring processes at the same time.

To avoid this, locks can also be used while updating matrix elements at the boundaries. However, another common and more efficient approach is to use a temporary matrix that is local to each process. Each process computes new values into its local matrix, based on the previous values in the global matrix. After all processes finish (synchronized using a `barrier`), they copy their local results into the global matrix. This approach is called **double buffering**.

Locks, while helpful, are expensive. If each grid point update needs to acquire a lock (especially for `diff`), it introduces a lot of waiting. Every process will have

to wait for others to release the lock before updating `diff`. This adds overhead and reduces performance.

But here's an observation: we only need `diff` at the end of each iteration — to check convergence. So, instead of updating the global `diff` every time, each process can maintain a `mydiff` variable locally. At the end of the iteration, all local `mydiff` values are added together to get the global `diff`. This is called a **reduction operation**. It significantly reduces locking overhead.

Previously, locks were required for each of the $O(n^2)$ grid updates. Now, with local diffs and a single reduction step, we reduce the locking overhead to $O(p)$, where p is the number of processes — a huge improvement!

The updated `Solve` function with this idea is shown in Listing 5.4.

```
procedure Solve(A)  /* Solve the equation system */
float **A;
begin
    int i, j, pid, done = 0;
    float mydiff, temp;

    mybegin = 1 + (n / nprocs) * pid;
    myend   = mybegin + (n / nprocs);

    while (!done) do  /* Outer loop over sweeps */
        mydiff = diff = 0;  /* Initialize differences */

        Barrier(barrier1, nprocs);  /* Synchronize before sweep */

        for (i = mybegin; i < myend; i++) do  /* Grid sweep */
            for (j = 1; j <= n; j++) do
                temp = A[i][j];  /* Save old value */
                A[i][j] = 0.2 * (A[i][j] + A[i][j-1] + A[i-1][j] +
                    A[i][j+1] + A[i+1][j]);
                mydiff += abs(A[i][j] - temp);
            end for
        end for

        lock(diff-lock);
            diff += mydiff;  /* Accumulate local diff to global diff
                */
        unlock(diff-lock);

        Barrier(barrier1, nprocs);  /* Synchronize before checking */

        if (diff / (n * n) < TOL) then
            done = 1;  /* Check convergence */

        Barrier(barrier1, nprocs);  /* Final barrier before next
            sweep */
    end while
end procedure
```

Listing 5.4: Updated Solve Function

Note that in the updated pseudocode, a new local variable `mydiff` is introduced for each process. This variable is updated independently by each process during the iteration. At the end of the iteration, all local `mydiff` values are added together to get the global `diff` value, using the locking mechanism as shown in Listing 5.4.

The statements `lock(diff-lock)` and `unlock(diff-lock)` are used to acquire and release the lock on the shared `diff` variable. This reduces the contention for the lock, since it is needed only once per process per iteration, not for every grid point update.

A `barrier` call is placed after the accumulation step. This ensures that all processes have completed their computations and updated the global `diff` before any process checks the termination condition. Without this synchronization, some processes might check the condition before `diff` is fully updated, leading to incorrect results.

Another `barrier` is placed right after the termination condition check. This is important because if the condition is not met, one process might enter the next iteration and set `diff = 0` too early. Meanwhile, another process still evaluating the condition might see this incorrect zero value, which would be wrong. The second barrier ensures that all processes finish checking the correct value of `diff` before any of them proceed to the next iteration.

The assignment `done = 1` is done redundantly by all processes once convergence is reached. This is acceptable because the statement is identical to what appears in a sequential program.

This brings us to an important point. One of the advantages of shared memory programming is that the code structure remains close to the sequential version. Parallelism is introduced simply by adding constructs like `locks` and `barriers`. This makes shared memory programs easier to write and debug compared to message passing programs, where the entire structure is different and processes need to explicitly communicate using send and receive operations.

Message Passing Version

In the message passing model, we cannot assume that matrix `A` is a global shared array accessible to all processes. There is no concept of shared variables here. Each process must maintain its own local part of the matrix.

If a process needs values that were updated by another process, it must explicitly receive them through message passing. So, data has to be sent between processes.

We use *domain decomposition* to divide the 2D domain among different processes. Each process follows the *owner computes* rule, meaning the process responsible for a subdomain also computes values in that region.

Since there is no global data structure, each process maintains local data structures. This increases the need for communication. To handle this, we introduce something called **ghost rows**.

Ghost rows are explained in Figure 5.7.

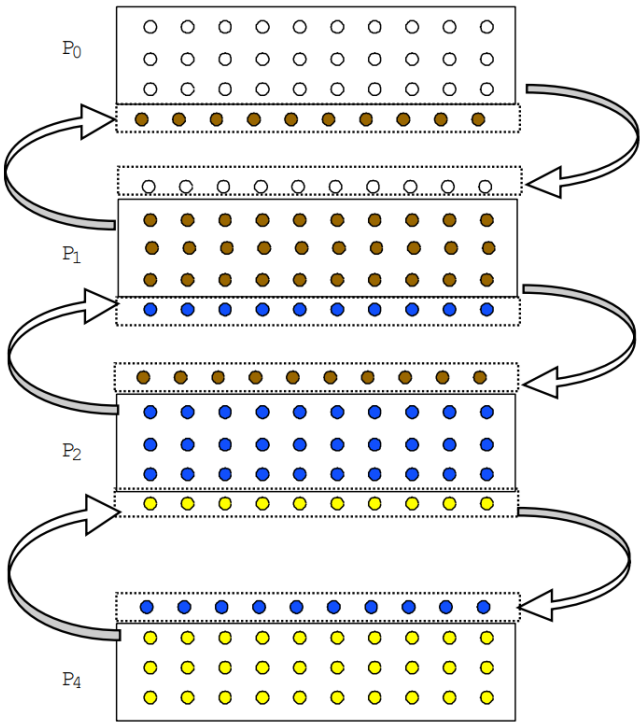


Figure 5.7: Ghost Rows

In this figure 5.7, the 2D grid is divided among 4 processes, just like in the shared memory model. Each process is responsible for updating values in its assigned rows.

But unlike the shared memory model, each process now has its own local array to store these rows. In addition to this, each process has extra rows called **ghost rows**, as shown in the figure.

These ghost rows are used to store boundary values received from neighboring processes. For example:

- Process 0 has a ghost row at the bottom to store the top boundary row from Process 1.
- Process 1 has a ghost row at the top for Process 0 and at the bottom for Process 2.
- And so on for the other processes.

Ghost rows are not updated by the process that holds them. Instead, they are updated by the neighboring processes that own the actual data. The purpose of ghost rows is to help compute local values at the boundaries using up-to-date data from neighbors.

Ghost rows are also called *halo rows*, *halo regions*, or *ghost zone regions*.

Once each process receives the required ghost row data from its neighbors, it can simply perform local (sequential) computation using its local array.

Now consider the message passing version of the program shown in Listing 5.5.

```
1 int n, nprocs; /* matrix: (n+2) x (n+2) elements */
2 float **myA;
3
```

```
main()
begin
    read(n);            /* read input parameter: matrix size */
    read(nprocs);       /* read number of processes */
    A = malloc(a 2-D array of (n+2) x (n+2) doubles);
    Create(nprocs-1, Solve, A);
    initialize(A);      /* initialize the matrix A somehow */
    Solve(A);
    Wait_for_End(nprocs-1);
end main
```

Listing 5.5: Message Passing Version - Main Function

Now consider the `Solve` function for the message passing version, shown in Listing 5.6, which includes array allocation and ghost row communication.

```
procedure Solve(A) /* solve the equation system */
    float **A; /* A is a (n+2)-by-(n+2) array
begin
    int i, j, pid, done = 0;
    float mydiff, temp;
    myend = (n / nprocs);
    myA = malloc(array of (n / nprocs) x (n) floats);
    intialize(myA); /* Intialization myA locally*/

    while (!done) do /* outermost loop over sweeps */
        mydiff = 0; /* initialize local difference to 0 */

        if (pid != 0) then
            SEND(&myA[1,0], n * sizeof(float), (pid - 1), row);
        if (pid != nprocs - 1) then
            SEND(&myA[myend,0], n * sizeof(float), (pid + 1), row);
        if (pid != 0) then
            RECEIVE(&myA[0,0], n * sizeof(float), (pid - 1), row);
        if (pid != nprocs - 1) then
            RECEIVE(&myA[myend + 1,0], n * sizeof(float), (pid + 1),
                row);

        for (i = 1 to myend) do
            for (j = 1 to n) do
                temp = A[i,j]; /* save old value */
                A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
                    A[i,j+1] + A[i+1,j]); /* compute average */
                mydiff += abs(A[i,j] - temp);
            end for
        end for

        if (pid != 0) then
            SEND(mydiff, sizeof(float), 0, DIFF);
            RECEIVE(done, sizeof(int), 0, DONE);
        else
            for k = 1 to nprocs - 1 do
                RECEIVE(tempdiff, sizeof(float), k, DIFF);
                mydiff += tempdiff;
            end for
            if (mydiff / (n * n) < TOL) then done = 1;
            for k = 1 to nprocs - 1 do
                SEND(done, sizeof(int), k, DONE);
            end for
```

```
42      end while
43  end procedure
```

Listing 5.6: MPI - Solve Function

In Listing 5.6, `myA` is the local array for each process. It stores the values of the grid elements corresponding to the domain allocated to the process—some subset of rows and all columns.

There are multiple ways to initialize these arrays: one option is to have each process read from a file; another is to let one process initialize the entire matrix and then distribute chunks to the others. For simplicity, we assume the processes are somehow able to initialize their local matrices.

Inside the `while` loop, each process maintains a local `mydiff` variable (note: no shared memory, so each process computes this separately).

At the start of each iteration, every process exchanges boundary rows with its neighboring processes:

- If `pid` $\neq 0$, the process sends its top row to `pid - 1`.
- If `pid` $\neq$ `nprocs - 1`, it sends its bottom row to `pid + 1`.
- Then, it receives the boundary rows from its neighbors and stores them in its **ghost rows**.

The ghost rows allow a process to compute its inner grid points using up-to-date boundary values received from neighbors. These ghost rows are only used for reading during the computation and are updated only via explicit communication (not by the owning process).

Once the ghost rows are updated, each process updates its portion of the grid just like in the sequential version. It also calculates its local difference `mydiff` during the update.

After computation:

- Non-zero processes send their `mydiff` to process 0.
- Process 0 receives all `mydiff` values, aggregates them into a global `diff`, and checks the convergence condition.
- If convergence is met, it sends a `done` signal to all other processes.
- All processes exit the loop when `done = 1`.

This is how the message passing version of the program works using MPI-style pseudocode.

5.2.5 Notes on Message Passing Version

- In the shared memory model, although there's no explicit communication, when one process writes to a variable and another reads it, this can still be seen as a form of communication. However, it is **receiver-initiated**, meaning the communication completes when the receiving process accesses the variable.

- In contrast, in the message passing model, communication is **sender-initiated**. A process must explicitly send data to another process using message-passing primitives like `SEND` and `RECEIVE`.
- Synchronization in the message passing model is **implicit**. Unlike in the shared memory model where synchronization is often done using explicit barriers, here it happens through communication operations. For instance, a `SEND` and `RECEIVE` pair naturally synchronizes two processes.
- This brings up a critical question: **can deadlocks occur in MPI communication?** Yes, they can. A deadlock may happen if all processes are stuck waiting for others to send data while none are sending anything.

  For example:

  - All processes issue a `RECEIVE` before any `SEND`. Since no data is being sent, all processes block and wait indefinitely.
  - Alternatively, all processes issue a blocking `SEND` that only completes when the data is received. But since no process is yet ready to receive, all `SEND`s block, leading again to a deadlock.
- To avoid such situations, we can use **non-blocking communication** or design the program so that at least one process starts with a `SEND` and another with a `RECEIVE`.
- Such communication is called **synchronous communication**, where the `SEND` completes only after the matching `RECEIVE` starts. This ensures synchronization between processes.
- Communication is performed in entire rows at the beginning of each iteration — not point-by-point. This means an entire row is sent in one go instead of sending one grid point at a time, which is more efficient.
- Observe the communication pattern where each process sends its `mydiff` value to process 0, which aggregates all values to compute the global difference. This is an **All-to-One communication pattern**, known as a **reduction**.
- Reduction is a common pattern in parallel programming. Many parallel libraries provide built-in support for such operations. In this case, we add the local `mydiff` values using a reduction operation.
- After computing the global difference, the master process (process 0) broadcasts the `done` signal to all other processes. This is a **One-to-All communication**, also known as a **broadcast**.
- The original code using `SEND` and `RECEIVE` can be simplified using `REDUCE` and `BROADCAST` operations, as shown below:

```
/* Communicate local diff values and determine if done,
   using reduction and broadcast */
REDUCE(0, mydiff, sizeof(float), ADD);
if (pid == 0) then
    if (mydiff / (n * n) < TOL) then
```

```
6           done = 1;
7       endif
8       BROADCAST(0, done, sizeof(int), DONE);
```

- The `REDUCE` operation collects the `mydiff` values from all processes at process 0 using an `ADD` operator to compute the global difference. Other operators like `MAX`, `MIN`, etc., can also be used depending on the need.

- After checking the termination condition, the master process uses `BROADCAST` to send the `done` signal to all other processes. This makes the communication more concise and efficient.

5.2.6 Send and Receive Alternatives

Recall that in the message-passing version, we used the `SEND` and `RECEIVE` operations to communicate data between processes. However, it may happen that the `SEND` operation causes the sending process to wait until the corresponding receiving process initiates its `RECEIVE` operation, and vice versa. This mutual waiting results in synchronous communication, where both sender and receiver are blocked until the message transfer is complete.

To avoid this blocking behavior, asynchronous communication can be used. Many MPI libraries support asynchronous `SEND` and `RECEIVE` operations, where a process does not need to wait for the corresponding operation on the other process to begin. In an asynchronous `SEND`, the sender proceeds without waiting for the receiver to initiate the `RECEIVE`. Similarly, in an asynchronous `RECEIVE`, the receiver does not block waiting for the sender to begin transmission.

Asynchronous operations can be further categorized into **blocking asynchronous** and **non-blocking asynchronous** variants. These will be discussed in detail in the MPI section.

5.2.7 Summary

- **Shared Address Space (SAS)**: Shared and private data are explicitly separated. Communication occurs implicitly through shared memory access patterns. Synchronization is explicit, typically performed via atomic operations such as locks and barriers.

- **Message Passing**: There is no concept of shared memory; all data structures reside in local address spaces. Communication is explicit via `SEND` and `RECEIVE` operations. In the example discussed, synchronization is implicit—no explicit barrier is used—but explicit synchronization constructs also exist in message-passing models.

- While decomposition and assignment strategies were similar in both SAS and message-passing models, the key differences lie in orchestration and mapping. Table 5.3 summarizes the comparison:

	Shared Address Space	**Message Passing**
Explicit in global structure?	Yes	No
Communication	Implicit	Explicit
Synchronization	Explicit	Implicit
Explicit replication of border rows?	No	Yes

Table 5.3: Comparison between Shared Address Space and Message Passing Models

For more details on parallelization principles refer to sections 3.1 and 3.5 in Kumar et al. [1994] and sections 2.2 and 2.3 in Culler et al. [1998] for example.

Chapter 6

Shared Memory Parallelism - OpenMP

Recall the term **processes** used in the context of shared memory address space models. In this setting, the term typically refers to **threads**. Threads are lightweight processes that share the same address space. Since multiple threads can access shared variables concurrently, synchronization mechanisms like barriers and locks are required to ensure data consistency and avoid race conditions.

The programmer must explicitly specify which variables are shared across threads and which are private to each thread. Because threads operate within the same address space, they can access common data structures and variables directly.

The most widely used programming interface for shared memory parallelism is OpenMP (Open Multi-Processing), a portable and scalable programming model. OpenMP provides a set of compiler directives, library routines, and environment variables to express parallelism at a high level in Fortran, C, and C++ programs.

So, what are compiler directives? These are special annotations or statements within the code that instruct the compiler to perform certain actions. For example, compilers support various optimization levels like `-O1`, `-O2`, etc. Through compiler directives, you can direct the compiler to optimize specific parts of the code selectively. In OpenMP, such directives are used to express shared-memory parallelism and are often referred to as **pragmas**.

One of the key advantages of OpenMP is that it makes parallel programming relatively simple. You can start with a serial program, then gradually insert OpenMP directives to indicate parallel regions, and finally compile and run the program with appropriate OpenMP flags to enable parallel execution.

The first version of OpenMP was released in 1997. As of now, the widely used version is OpenMP 6.0, released in 2024.

6.1 Fork-Join Model

OpenMP follows the **fork-join** execution model. The program begins with a single thread, known as the *master thread*. When the program encounters a parallel construct, the master thread *forks* a team of worker threads to execute the parallel region. Once the parallel region is completed, the worker threads *join* back into the master thread, and execution resumes sequentially.

This behavior is illustrated in Figure 6.1.

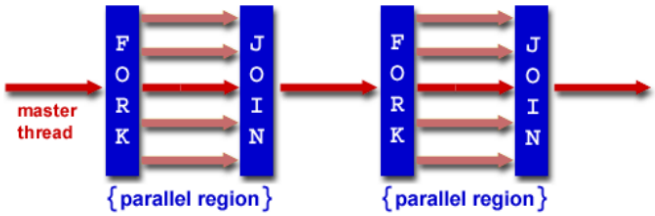


Figure 6.1: Fork-Join Model

As seen in the figure, a single thread starts executing the program. Upon encountering a parallel region, a team of threads is spawned to execute that block in parallel. After completion, the threads synchronize and terminate, returning control to the master thread.

One of the major advantages of OpenMP is its support for **selective parallelism**—only specific regions of the code are parallelized based on need. This aligns well with **Amdahl's Law**, which highlights that only parts of a program are parallelizable while others must remain serial. OpenMP does not attempt to parallelize the entire program but instead allows the programmer to specify which regions benefit most from parallel execution.

OpenMP provides several key constructs to support parallel programming:

- **Work-sharing constructs:** Specify how the work in a parallel region should be divided among the threads.
- **Synchronization constructs:** Ensure proper coordination and data consistency between threads.
- **Data environment constructs:** Define which variables are shared across threads and which are private to each thread.
- **Library routines and environment variables:** Provide utility functions and allow customization of the execution environment (e.g., setting the number of threads using environment variables before program execution).

A *construct* in OpenMP refers to a compiler directive paired with the block of code (region) to which it applies.

OpenMP primarily supports **loop-level parallelism**, especially for `for`/`do` loops. If there are no loop-carried dependencies, iterations can be safely divided among threads. Each thread then executes a subset of the loop iterations. This offers **fine-grained parallelism**, giving the programmer detailed control over which specific regions of code are parallelized.

Additionally, OpenMP allows **dynamic parallelism**—the number of threads used in a parallel region can be varied based on computational load. Regions that benefit from greater parallelism can use more threads, while lighter regions may use fewer. This flexibility is useful for optimizing performance dynamically at runtime.

Alongside fine-grained parallelism, OpenMP also supports **coarse-grained parallelism**, where entirely different sections of the program are executed in parallel.

Modern versions of OpenMP include advanced features such as:

- **Offloading to accelerators (e.g., GPUs):** Through target directives, specific regions of code can be executed on devices such as GPUs.
- **SIMD vectorization:** Enables automatic vectorized execution of operations across multiple data elements.
- **Task-core affinity:** Allows tasks to be pinned to specific cores, which can help reduce data movement and improve cache utilization.

These features make OpenMP not just a tool for simple parallelism, but a comprehensive framework suitable for modern multicore and heterogeneous architectures.

6.2 Parallel Construct

`#pragma omp parallel` is the primary parallel construct in OpenMP. It is used to create a **team of threads**, where each thread executes the specified code block in parallel. The syntax is:

```
#pragma omp parallel
{
    /* parallel region */
}
```

This block of code is referred to as the **parallel region**. When the compiler encounters the `#pragma omp parallel` directive, it forks a team of threads. Each thread in the team executes the code inside the braces. Before this directive is reached, the program executes sequentially on the **master thread**.

Thus, a **parallel construct** consists of the 'parallel' directive and the corresponding code block that is executed concurrently by multiple threads.

How many threads are created? This depends on how OpenMP is configured. There are several ways to control the number of threads:

- Set the environment variable at runtime: `export OMP_NUM_THREADS=number`
- Use the function call: `omp_set_num_threads(n);`
- Use the clause `num_threads(n)` directly in the pragma directive.

OpenMP also provides various **clauses** to modify the behavior of the parallel region:

- `if (condition)`: Executes the parallel region in parallel only if the condition evaluates to true; otherwise, it is executed serially by the master thread.
- `num_threads(n)`: Specifies the exact number of threads to be used in the parallel region.
- `default(shared | none)`: Sets the default data-sharing attributes for variables—either shared or requiring explicit declaration.
- `private(list)`: Declares a list of variables that are private to each thread. Each thread gets its own uninitialized copy.

- `firstprivate(list)`: Like ‘private’, but initializes each thread’s copy with the value of the variable outside the parallel region.
- `shared(list)`: Declares a list of variables to be shared across all threads.
- `copyin(list)`: For variables declared ‘threadprivate‘, initializes the value of each thread’s copy with that of the master thread.
- `reduction(operator: list)`: Each thread gets a private copy of the listed variables, which are combined at the end of the parallel region using the specified operator (like ‘+’, ‘*’, ‘max’, etc.).
- `proc_bind(master | close | spread)`: Controls thread-to-core binding. ‘master’ binds threads to the same core as the master thread; ‘close’ places them on nearby cores; ‘spread’ distributes them across cores.

Consider the following example, which prints the number of threads created inside a parallel region:

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

int main()
{
    int nthreads;
    #pragma omp parallel
    {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }
    return 0;
}
```

If, say, 4 threads were used, you’ll see the line ”Number of threads = 4” printed 4 times—once by each thread.

Now consider a basic Hello World example in OpenMP:

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

int main()
{
    int nthreads, tid;
    #pragma omp parallel private(nthreads, tid)
    {
        tid = omp_get_thread_num();
        printf(``Hello World from thread %d\n", tid);
    }
    return 0;
}
```

Here, each thread prints its own message. The ‘private’ clause ensures that ‘tid’ and ‘nthreads’ are thread-local, preventing race conditions. Try running this code on your machine to see multiple ”Hello World” messages printed in parallel.

OpenMP makes it straightforward to parallelize sequential programs incrementally. You can write the core logic sequentially and then wrap performance-critical sections with appropriate OpenMP constructs.

6.3 Work Sharing Construct

The work sharing construct in OpenMP is used to distribute work among multiple threads. Once multiple threads have been spawned using the `parallel` construct, the work sharing construct specifies how the workload should be divided among these threads.

There are three main types of work sharing constructs in OpenMP:

- **loops** – used to distribute iterations of a loop among threads,
- **sections** – used to assign distinct code blocks to different threads,
- **single** – ensures that a particular block of code is executed by only one thread.

6.3.1 for-loop

The `for` construct is used to parallelize `for`-loops or `do`-loops in OpenMP using the syntax:

```
#pragma omp for [clause[, clause]...]
```

It is typically used inside a `parallel` region to distribute loop iterations among the threads. The following clauses can be used with the `for` construct:

- **private(list)**: Specifies variables that are private to each thread.
- **firstprivate(list)**: Like `private`, but initializes each private copy with the value of the original variable.
- **lastprivate(list)**: The variable is made private, but its value from the last iteration is copied back after the loop.
- **linear(list)**: Specifies linear induction variables with a fixed step size.
- **reduction(operator:list)**: Performs a reduction on the specified variables using the given operator across threads.
- **schedule([modifier:]kind[,chunk])**: Controls how iterations are divided among threads.
- **collapse(n)**: Useful for nested loops—collapses `n` nested loops into a single loop with a larger iteration space for parallelization.
- **ordered(n)**: Enforces ordered execution for sections of code marked with `ordered`.
- **nowait**: Prevents implicit barrier synchronization at the end of the loop. Threads need not wait for others.

The `for` construct is one of the most commonly used and important constructs in OpenMP. It assumes that loop iterations are independent of one another, i.e., there are no dependencies between iterations. If such dependencies exist—e.g., an iteration relies on the result of a previous iteration—then the `for` construct should not be used.

Schedule Clause: The `schedule` clause determines how loop iterations are assigned to threads. OpenMP supports several scheduling policies:

1. **schedule(static, chunk)**: Divides the iteration space into equal-sized chunks (of size `chunk`) and assigns them to threads in round-robin fashion.

   For example, consider a loop with 20 iterations and a chunk size of 2. Ten chunks are created, and with four threads, chunks are assigned as:

   Thread 0: chunk 0, 4, 8; Thread 1: chunk 1, 5, 9; ...

2. **schedule(dynamic, chunk)**: Also splits the loop into chunks, but assigns them to threads dynamically at runtime. Threads that finish early can grab the next available chunk.

3. **schedule(runtime)**: Defers scheduling strategy selection to runtime based on environment variables or OpenMP runtime configuration.

Example: Consider the following program that performs element-wise addition of two arrays:

```
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000

int main(){
    int i, chunk;
    float a[N], b[N], c[N];

    // Initialization
    for (i = 0; i < N; i++)
        a[i] = b[i] = i * 1.0;

    chunk = CHUNKSIZE;

    #pragma omp parallel shared(a, b, c, chunk) private(i)
    {
        #pragma omp for schedule(dynamic, chunk) nowait
        for (i = 0; i < N; i++) {
            c[i] = a[i] + b[i];
        }
    }

    return 0;
}
```

Listing 6.1: Addition of Two Arrays Using `for` Construct

In this example, we are adding arrays `a` and `b` to produce `c`. Since each element-wise addition is independent, this is a good candidate for parallelization.

We use `#pragma omp parallel` to create threads. The variables `a`, `b`, `c`, and `chunk` are marked as `shared` because they need to be accessed by all threads, while `i` is marked as `private` since each thread should have its own loop counter to avoid conflicts.

The inner `for` loop is parallelized using the `for` construct with `schedule(dynamic, chunk)` to allow dynamic load balancing. The `nowait` clause prevents threads from waiting for others once their iterations are complete.

Note that the two pragmas can be combined into a single directive as follows:

```
#pragma omp parallel for shared(a, b, c, chunk) private(i)
    schedule(dynamic, chunk) nowait
```

6.3.2 Coarse-Level Parallelism — Sections and Tasks

In addition to fine-level parallelism (as seen in the `for` loop example), OpenMP supports **coarse-level parallelism**.

While fine-level parallelism typically deals with distributing iterations of a loop among multiple threads, coarse-level parallelism involves executing **independent blocks of code**—like separate functions or computational phases—in parallel.

Sections Construct: If we want to execute two or more independent code blocks (e.g., different functions) concurrently, OpenMP provides the `sections` construct. This allows different threads to execute different code blocks in parallel, rather than all threads executing the same block of code as in `for` loops.

Consider the example shown in listing 6.2:

```
#pragma omp parallel sections [clauses]
{
    #pragma omp section
    {
        // structure block i (e.g., function1());
    }

    #pragma omp section
    {
        // structure block j (e.g., function2());
    }
    ...
}
```

Listing 6.2: Sections Construct

In this example, each `#pragma omp section` defines a code block (or task) that can be executed by a separate thread. The enclosing `parallel sections` directive spawns a team of threads, and each section is assigned to one thread for execution. This is useful when you have multiple, independent tasks that can be run concurrently.

Tasks Construct: OpenMP also supports dynamic task creation using the `task` construct. This is useful in scenarios such as recursive algorithms, adaptive simulations, or when the number and structure of tasks are not known in advance.

Inside a parallel region, individual threads can create tasks using the `task` directive. These tasks are then dynamically assigned to available threads by the OpenMP runtime system. This enables load balancing and fine control over task execution.

Here's a simplified example of the `task` construct, shown in listing 6.3:

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        {
            // Task 1 code
        }

        #pragma omp task depend(in: x)
        {
            // Task 2 depends on variable x from Task 1
        }
    }
}
```

Listing 6.3: Tasks Construct

In the above code:

- `#pragma omp single` ensures only one thread in the team creates tasks (preventing all threads from redundantly spawning the same tasks). - `#pragma omp task` defines a task to be executed by any thread. - The `depend` clause allows specifying task dependencies.

Task Dependencies and Task Graphs: OpenMP's `depend` clause enables you to define data dependencies between tasks. For instance, if Task 2 depends on a variable modified by Task 1, you can ensure that Task 2 will only execute once Task 1 has completed. This forms a **task dependency graph**, where:

- **Nodes** represent tasks.
- **Edges** represent dependencies.

The OpenMP runtime builds and schedules this graph automatically, respecting the constraints provided. This powerful feature enables writing highly dynamic and dependency-aware parallel programs.

Thus, tasks enable **dynamic coarse-level parallelism** where the number and structure of tasks can evolve during runtime, while also maintaining control over execution order through explicit dependencies.

6.4 Synchronization Directives

Once the work is distributed among threads, there is always a possibility that multiple threads may access and update common variables simultaneously. This can lead to race conditions and inconsistent results. Therefore, we need to synchronize threads and protect shared data using appropriate locking or ordering mechanisms.

OpenMP provides several synchronization directives for this purpose. Note that all these constructs must be enclosed within a parallel region (created by a parallel construct):

- `#pragma omp master`: The block of code following this directive is executed only by the master thread. This is useful, for example, when only one thread (typically the master) should perform tasks like printing to the console to avoid cluttered output.
- `#pragma omp critical`: Ensures that the enclosed block is executed by only one thread at a time. This is useful for updating shared variables, where concurrent access by multiple threads could lead to inconsistent or garbage values.
- `#pragma omp barrier`: Forces all threads in the team to wait at this point until every thread has reached it. Useful for synchronization at intermediate stages of the parallel region.
- `#pragma omp atomic`: Ensures that the specific memory operation (such as a simple update to a variable) following this directive is executed atomically—only one thread performs the update at a time, but with lower overhead than a full critical section.
- `#pragma omp ordered`: Specifies that the block of code should be executed in the order of thread IDs. This is helpful when we need ordered output—for example, when printing parts of an array sequentially from multiple threads.
- `#pragma omp flush`: Ensures memory consistency between threads by forcing a thread's private view of memory to be synchronized with shared memory. This is particularly useful when a thread updates a shared variable, and other threads must see the updated value immediately.

  Consider the following example using the `flush` directive, shown in listing 6.4.

```
int sync[NUMBER_OF_THREADS];
float work[NUMBER_OF_THREADS];

#pragma omp parallel private(iam, neighbor) shared(work, sync)
{
    iam = omp_get_thread_num();
    sync[iam] = 0;
    #pragma omp barrier

    // Perform computation and write result to shared array
    work[iam] = compute(iam);

    // Announce completion of work
    #pragma omp flush(work)       // Ensure work is visible to
       other threads
    sync[iam] = 1;
    #pragma omp flush(sync)       // Ensure sync is updated in
       shared memory

    // Wait for neighbor to complete its work
    neighbor = (iam > 0 ? iam : omp_get_num_threads()) - 1;
    while (sync[neighbor] == 0) {
```

```
21          #pragma omp flush(sync)
22      }
23
24      // Read neighbor's result
25      ... = work[neighbor];
26  }
```

Listing 6.4: Flush Directive

In this example, each thread computes its result and writes it to the shared `work` array. Neighboring threads need access to this data, so we use the `flush` directive to ensure memory visibility.

- The first `flush(work)` guarantees that the result written by a thread is visible to other threads before it signals completion. - The update `sync[iam] = 1` is used to notify others that the computation is complete. - The second `flush(sync)` ensures that this flag is visible to all threads. - The neighbor thread checks `sync[neighbor]` in a loop and repeatedly flushes it until the update is visible. - Once visible, it safely accesses `work[neighbor]`.

This pattern provides a consistent memory view and safe inter-thread communication. The `flush` directive is essential when shared variables are updated by one thread and read by others soon after.

6.5 Data Scope Attribute Clauses

OpenMP provides several clauses to control the scope and visibility of variables in parallel regions. These are known as *data scope attribute clauses*.

1. **private(list):** Each thread gets its own uninitialized copy of the variable(s) in the list. These variables are not shared and are re-created per thread.

2. **firstprivate(list):** Similar to `private`, but each thread's copy is initialized with the value the variable had in the master thread before entering the parallel region.

3. **lastprivate(list):** Like `private`, but the value from the last iteration (or the last thread to execute the region) is copied back to the original variable after the region ends.

4. **shared(list):** All threads share the same memory location for the listed variables. Any update by one thread is visible to all others.

5. **default(shared/private/none):** Sets the default sharing behavior for all variables not explicitly listed in other clauses. `default(none)` forces all variables to be explicitly scoped.

6. **reduction(operator:list):** A private copy is created for each thread. After the parallel region, the copies are combined using the specified operator and assigned to the original variable.

7. **copyin(list):** Used with `threadprivate` variables. Copies the master thread's value of the variable(s) to all threads at the beginning of a parallel region.

8. **copyprivate(list):** Used in a `single` construct to copy the value of a private variable from one thread to all others once the construct ends.

Declaring large multidimensional arrays as `private` leads to significant overhead because each thread allocates a copy. Using `firstprivate` adds further overhead due to per-thread initialization.

6.5.1 threadprivate

The `threadprivate` directive makes a global variable behave like a private variable — each thread has its own copy — but with values preserved across multiple parallel regions.

```
#include <omp.h>
int alpha[10], beta[10], i;
#pragma omp threadprivate(alpha)

main() {
    omp_set_dynamic(0); // Disable dynamic threads

    // First parallel region
    #pragma omp parallel private(i, beta)
    for (i = 0; i < 10; i++)
        alpha[i] = beta[i] = i;

    // Second parallel region
    #pragma omp parallel
    printf("alpha[3] = %d and beta[3] = %d\n", alpha[3], beta[3]);
}
```

Listing 6.5: threadprivate Directive

`alpha` retains its value in the second region due to `threadprivate`. `beta`, being merely `private`, does not preserve its value across regions.

To ensure thread consistency, the number of threads must be the same across regions. Hence, `omp_set_dynamic(0)` is used to disable dynamic thread allocation.

6.5.2 default-example

This example demonstrates the use of `default(none)`, along with `private`, `shared`, and `threadprivate` clauses.

```
int x, y, z[1000];
#pragma omp threadprivate(x)

void fun(int a) {
    const int c = 1;
    int i = 0;

    #pragma omp parallel default(none) private(a) shared(z)
    {
        int j = omp_get_num_thread(); // Thread-local variable
        a = z[j];      // OK: 'a' is private, 'z' is shared
        x = c;         // OK: 'x' is threadprivate, 'c' is const
        z[i] = y;      // ERROR: 'i' and 'y' not in any clause
```

```
14      }
15  }
```

Listing 6.6: default-Example

- `x` is `threadprivate`, so assignment `x = c` is legal.
- `a` is declared `private`, and `z` is shared, so `a = z[j]` is valid.
- `z[i] = y` fails because `i` and `y` are not listed in any clause and `default(none)` prevents implicit sharing.

6.6 Library Routines (API)

In addition to compiler directives and data-sharing clauses, OpenMP provides a powerful set of runtime library routines that offer functionality such as:

- Querying the OpenMP execution environment (e.g., number of threads, thread IDs).
- Managing locks and mutual exclusion.
- Configuring execution policies (e.g., dynamic threads, nested parallelism).

These functions are defined in the `omp.h` header file and can be categorized as follows:

6.6.1 1. Query Functions

- `omp_get_num_threads()`: Returns the number of threads in the current team.
- `omp_get_max_threads()`: Returns the maximum number of threads available.
- `omp_get_thread_num()`: Returns the thread ID of the calling thread (from 0 to `n-1`).
- `omp_get_num_procs()`: Returns the number of processors available.
- `omp_in_parallel()`: Returns true (non-zero) if called inside a parallel region.

6.6.2 2. Execution Environment Control

- `omp_set_num_threads(int num)`: Requests that `num` threads be used in subsequent parallel regions.
- `omp_set_dynamic(int flag)`: Enables/disables dynamic adjustment of the number of threads.
- `omp_get_dynamic()`: Returns whether dynamic threads are enabled.
- `omp_set_nested(int flag)`: Enables/disables nested parallel regions.
- `omp_get_nested()`: Returns whether nested parallelism is enabled.

6.6.3 3. Locking Routines

OpenMP provides two types of locks: regular locks and nested locks. Nested locks allow reentrant locking by the same thread.

- `omp_lock_t lock;` `omp_nest_lock_t nlock;` — Lock types.
- `omp_init_lock(&lock)` / `omp_init_nest_lock(&nlock)` — Initialize a lock.
- `omp_destroy_lock(&lock)` / `omp_destroy_nest_lock(&nlock)` — Destroy a lock.
- `omp_set_lock(&lock)` / `omp_set_nest_lock(&nlock)` — Acquire a lock.
- `omp_unset_lock(&lock)` / `omp_unset_nest_lock(&nlock)` — Release a lock.
- `omp_test_lock(&lock)` / `omp_test_nest_lock(&nlock)` — Attempt to acquire lock without blocking.

6.6.4 4. Timing Functions

Used for profiling and benchmarking.

- `omp_get_wtime()`: Returns elapsed wall-clock time (in seconds).
- `omp_get_wtick()`: Returns resolution (tick) of the timer used by `omp_get_wtime()`.

These APIs give fine-grained control over OpenMP's runtime behavior, enabling programmers to build high-performance, scalable, and thread-safe applications.

6.7 Locks

OpenMP supports both **simple locks** and **nestable locks**.

Simple Locks: A simple lock can only be acquired once by a thread. If a thread that already holds a simple lock attempts to acquire it again (i.e., calls `omp_set_lock` twice in a row), a deadlock will occur. This is because only the thread that currently holds the lock can release it, and if it waits on itself, it will wait forever. Therefore, simple locks must never be nested.

Nestable Locks: In contrast, **nestable locks** allow a thread to acquire the same lock multiple times without causing a deadlock. Internally, OpenMP maintains a counter for the number of times the lock has been acquired by the same thread. The lock is only released when the number of `omp_unset_nest_lock()` calls matches the number of `omp_set_nest_lock()` calls.

Nestable locks are especially useful in complex software such as numerical libraries where multiple paths in a program may enter critical functions that need mutual exclusion.

The key distinctions between the two types of locks are:

- **Simple lock:** Cannot be acquired recursively. Available only if currently unlocked.
- **Nestable lock:** Can be acquired recursively by the same thread. Considered available if either unlocked or already owned by the calling thread.

Example: Nested Lock Usage

Consider the following example illustrating the need for nested locks, shown in Listing 6.7:

```
#include <omp.h>

typedef struct {
    int a, b;
    omp_nest_lock_t lck;
} pair;

void incr_a(pair *p, int a)
{
    // Called only from incr_pair, no need to lock
    p->a += a;
}

void incr_b(pair *p, int b)
{
    // Called both from incr_pair and elsewhere,
    // hence must be protected with a nested lock

    omp_set_nest_lock(&p->lck);
    p->b += b;
    omp_unset_nest_lock(&p->lck);
}
```

Listing 6.7: Nested lock — Function design

In this example:

- The structure `pair` holds two integers `a`, `b` and a nested lock `lck`.
- `incr_a()` is called only from `incr_pair()`, hence no locking is required.
- `incr_b()` is invoked from both `incr_pair()` and elsewhere concurrently, so we must protect it with a nested lock.

Example Continued: Parallel Region with Nested Locks

Now we illustrate a complete example using OpenMP sections to call both `incr_pair()` and `incr_b()` in parallel. This is shown in Listing 6.8:

```
void incr_pair(pair *p, int a, int b)
{
    omp_set_nest_lock(&p->lck);
    incr_a(p, a);
    incr_b(p, b);
    omp_unset_nest_lock(&p->lck);
}
```

```

void f(pair *p)
{
    extern int work1(), work2(), work3();

    #pragma omp parallel sections
    {
        #pragma omp section
        incr_pair(p, work1(), work2());

        #pragma omp section
        incr_b(p, work3());
    }
}
```

Listing 6.8: Nested lock — Parallel sections

Explanation:

- `f()` defines a parallel region with two sections.
- One thread executes `incr_pair()` which internally calls both `incr_a()` and `incr_b()`.
- Another thread directly calls `incr_b()`.
- Since `incr_b()` may be accessed concurrently, and can be nested inside `incr_pair()`, a **nested lock** is necessary to avoid race conditions and deadlocks.

This example clearly demonstrates how nested locks are essential when a function may be both independently and dependently called, and shared data must be safely modified under concurrent access patterns.

6.7.1 Example 1: Jacobi Solver

This example implements the Jacobi iterative method for solving discretized grid-based problems (such as Laplace's equation), except here we are parallelizing across *iterations* (i.e., grid points) instead of just rows.

Consider the following code, shown in Listing 6.9:

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

int main(int argc, char** argv){
    ...
    int rows, cols;
    int* grid;
    int chunk_size, threads = 16;
    int i, j;

    // Allocate and initialize the grid
    grid = malloc(sizeof(int) * rows * cols);
    for (i = 0; i < rows; i++) {
        for (j = 0; j < cols; j++) {
            grid[i * cols + j] = ...; // Some initialization
        }
    }
```

```
18
19      chunk_size = rows / threads;
20
21      #pragma omp parallel for num_threads(16)
22          private(i, j) shared(rows, cols, grid)
23          schedule(static, chunk_size) collapse(2)
24      for (i = 1; i < rows - 1; i++) {
25          for (j = 1; j < cols - 1; j++) {
26              grid[i * cols + j] = 0.25 * (
27                  grid[i * cols + (j - 1)] + grid[i * cols + (j + 1)] +
28                  grid[(i - 1) * cols + j] + grid[(i + 1) * cols + j]
29              );
30          }
31      }
32      ...
33      return 0;
34  }
```

Listing 6.9: Jacobi solver using OpenMP

As seen in the code, we initialize a 2D grid stored in a flattened 1D array. The Jacobi update computes each internal point as the average of its four neighbors (left, right, top, and bottom).

To parallelize the computation using OpenMP:

- We define `chunk_size` as `rows / threads`, which controls the number of iterations (in terms of row blocks) assigned per thread.
- The `#pragma omp parallel for` directive initiates parallel execution of the nested loops using 16 threads.
- Variables `i` and `j` are declared as `private` to ensure each thread gets its own copy.
- `rows`, `cols`, and the pointer to `grid` are declared `shared` since all threads work on the same data array.
- `schedule(static, chunk_size)` ensures iterations are distributed statically in contiguous blocks to the threads, which is efficient for load-balanced problems.
- The `collapse(2)` clause combines both loops into a single iteration space of size `(rows - 2) * (cols - 2)`. This allows better load balancing by distributing the total work across threads, rather than just distributing outer loop iterations.

Had we not used `collapse(2)`, only the outer loop (over `i`) would have been parallelized. In such a case, each thread would be responsible for all iterations of the inner loop (`j`), potentially causing load imbalance and underutilization of cores, especially for small `rows`.

6.7.2 Breadth First Search (BFS) Algorithm

The Breadth First Search (BFS) algorithm explores a graph level by level. Starting from a given source vertex at level 0, BFS first visits all its immediate neighbors (level 1), then the neighbors of those neighbors (level 2), and so on, until all reachable

vertices are visited. This traversal pattern makes BFS naturally parallelizable at each level of the graph.

We now present two parallel implementations of BFS: one using **nested parallelism**, and the other using **task-based parallelism**.

Version 1: Nested Parallelism

The following code snippet (Listing 6.10) illustrates a BFS implementation using nested `parallel for` constructs.

```
...
level[0] = {s};
curLevel = 0;
dist[s] = 0;
for (v != s) dist[v] = -1;

while (level[curLevel] != NULL) {
    #pragma omp parallel for ...
    for (i = 0; i < length(level[curLevel]); i++) {
        v = level[curLevel][i];
        neigh = neighbors(v);

        #pragma omp parallel for ...
        for (j = 0; j < length(neigh); j++) {
            w = neigh[j];
            if (dist[w] == -1) {
                level[curLevel + 1] = union(level[curLevel + 1], w);
                dist[w] = dist[v] + 1;
            }
        }
    }
}
```

Listing 6.10: Parallel BFS using nested parallelism

In this approach:

- We initialize the BFS by placing the source vertex s at level 0 and marking all other vertices as unvisited (distance $= -1$).
- For each level, we use the outer `#pragma omp parallel for` to parallelize across the current level's vertices.
- For each vertex v at this level, we obtain its neighbors and use an inner `parallel for` to visit all neighbors in parallel.
- If a neighbor w has not yet been visited (i.e., `dist[w] == -1`), we add it to the next level and update its distance.

This is an example of **nested parallelism**: for each thread processing a vertex in the outer loop, additional threads are spawned to process its neighbors.

Version 2: Task-Based Parallelism

The next version (Listing 6.11) uses OpenMP tasks to dynamically create units of work for visiting each neighbor.

```
...
level[0] = {s};
curLevel = 0;
dist[s] = 0;
for (v != s) dist[v] = -1;

while (level[curLevel] != NULL) {
    #pragma omp parallel
    {
        #pragma omp single
        for (v in level[curLevel]) {
            for (w in neighbors(v)) {
                #pragma omp task
                {
                    if (dist[w] == -1) {
                        level[curLevel + 1] = union(level[curLevel +
                            1], w);
                        dist[w] = dist[v] + 1;
                    }
                }
            }
        }
    }
}
```

Listing 6.11: Parallel BFS using OpenMP tasks

In this version:

- We begin with a `parallel` region and enclose the main loop with a `single` directive to avoid redundant loop execution.
- For each neighbor of a vertex, we dynamically spawn an `omp task` to evaluate whether that neighbor should be added to the next level.
- Tasks are scheduled dynamically by OpenMP's runtime system and assigned to available threads, improving efficiency in irregular graphs.

This method is more **adaptive** and **scalable**, especially in graphs with high variation in vertex degrees, since tasks can be scheduled independently and load-balanced across cores at runtime.

For further details on OpenMP, I would suggest the reader to refer to Lawrence Livermore National Laboratory [2021] for tutorials on OpenMP or YouTube Lectures by Intel Software [2014] and official website of OpenMP OpenMP Architecture Review Board [2025] to get a more indepth and broader understanding of OpenMP concepts. The ppt referred in the lectures can be found in the YouTube description and serves as an excellent guide for understanding the lectures as well as OpenMP overall in general.

Chapter 7

Message Passing Interface (MPI)

MPI is a standardized and portable message-passing system designed to enable distributed-memory parallelism. Unlike shared memory paradigms such as OpenMP—where parallelism is achieved implicitly through directives—MPI requires explicit communication and synchronization between processes. This results in higher programming complexity, as each process must execute a specific program that communicates with others explicitly. For a gentler introduction refer to the last four YouTube lectures by Prof. Mathew Jacob on High performance Computing (HPC) NPTEL [2020].

Despite its complexity, MPI remains the dominant model for large-scale high-performance computing (HPC). This is because, although modern compute nodes may contain many cores and support shared-memory programming within a node, large-scale scientific and engineering problems often exceed the capacity of a single node. In such cases, multiple nodes must collaborate. Since nodes cannot directly access each other's memory, communication between them must occur via message passing. This is where MPI becomes indispensable.

MPI is especially well-suited for:

- Cluster computing, where each node has its own local memory.
- Large-scale simulations in scientific computing.
- Explicit control over data distribution and parallel execution.

In this chapter, we will explore:

- The basic execution model of MPI.
- Communication primitives such as point-to-point and collective communication.
- Process topologies, communicators, and derived data types.
- Practical code examples to reinforce concepts.

7.1 MPI Introduction

The **Message Passing Interface (MPI)** is a standardized and portable communication protocol for explicit message passing in **MIMD** (Multiple Instruction, Multiple Data) architectures. One of the key strengths of MPI is its **portability**—a correctly

written MPI program can run on any hardware that supports an MPI implementation, without requiring modification.

MPI is widely supported by hardware vendors and has become the de facto standard for distributed-memory parallel programming. The latest major version is **MPI-3**, released in 2012.

The MPI standard covers the following core components:

- **Point-to-Point Communication:** Enables direct communication between pairs of processes using `send` and `receive` primitives.
- **Collective Communication:** Provides communication operations involving groups of processes, such as broadcast, scatter, gather, and reduction.
- **Communication Contexts:** Define the scope and isolation of communications. This helps prevent interference between distinct communication domains in large applications.
- **Process Topologies:** MPI allows the specification of logical communication patterns between processes (e.g., Cartesian grids or graphs), analogous to physical network topologies like meshes, buses, or rings. While hardware-level topologies refer to physical connections, process topologies in MPI refer to how processes logically exchange information in a structured manner.
- **Profiling Interface:** MPI provides hooks for performance profiling by allowing users to wrap or intercept MPI function calls. This facilitates custom instrumentation and performance monitoring tools.
- **Parallel I/O (MPI I/O):** Introduced in **MPI-2**, this supports coordinated reading and writing of files by multiple processes. It helps improve I/O performance and scalability for large data sets.
- **Dynamic Process Management:** Also part of **MPI-2**, it allows spawning new processes at runtime and managing groups of communicating processes dynamically, a useful feature in adaptive or interactive applications.
- **One-sided Communication (Remote Memory Access - RMA):** Allows a process to access the memory of another process without that process explicitly participating in the communication. It enables operations like `put`, `get`, and `accumulate` to be performed without a matching receive on the target.
- **Extended Collectives:** Introduced in later versions, these provide more optimized and scalable collective communication routines, with additional features such as non-blocking collectives.

MPI offers a rich set of functions to build parallel programs over distributed memory architectures. These include initialization routines, communication functions, synchronization primitives, topological mapping utilities, and environment management tools. We will explore each of these components in the subsequent sections through examples and explanations.

7.2 Communication Primitives

7.2.1 Point-to-Point Communication

MPI supports **point-to-point communication** through the functions `MPI_Send` and `MPI_Recv`, which allow direct message exchange between pairs of processes.

```
MPI_Send(buf, count, datatype, dest, tag, comm)
MPI_Recv(buf, count, datatype, source, tag, comm, status)
MPI_Get_count(status, datatype, count)
```

The communication is initiated by one process using `MPI_Send`, and completed when the destination process calls `MPI_Recv`. Let's break down the parameters:

MPI_Send

- `buf`: Pointer to the data buffer containing the message to be sent.
- `count`: Number of elements in the buffer.
- `datatype`: MPI predefined datatype of the buffer elements (e.g., `MPI_INT`, `MPI_FLOAT`).
- `dest`: The rank of the receiving process within the given communicator. MPI assigns each process a **rank** from `0` to `p-1`, where `p` is the number of processes.
- `tag`: An integer label to identify the message. This is useful when multiple messages are exchanged between the same sender and receiver.
- `comm`: The **communicator**, which defines the communication context or group of processes involved. The default communicator is `MPI_COMM_WORLD`, which includes all spawned processes.

MPI_Recv

- `buf`: Buffer to store the incoming message.
- `count`: Maximum number of elements that can be received in the buffer.
- `datatype`: Datatype of the expected incoming elements.
- `source`: Rank of the sending process.
- `tag`: Tag used to identify the message. Should match the tag used in `MPI_Send`.
- `comm`: Communicator specifying the communication domain.
- `status`: A pointer to an `MPI_Status` structure. It contains information about the received message such as the source, tag, and actual number of received elements.

To retrieve the actual number of received items, the function `MPI_Get_count` is used:

```
MPI_Get_count(&status, datatype, &recv_count);
```

Communicator and Rank Context

When an MPI program is launched, all processes are part of a global communication group referenced by the communicator `MPI_COMM_WORLD`. Each process is assigned a **unique rank** within this communicator. However, MPI supports creation of subgroups (possibly overlapping), and a process can have different ranks within different subgroups.

For example, a process may have rank 4 in the global communicator but rank 0 within a smaller subgroup. This introduces ambiguity: referring to a rank alone is insufficient unless the associated communicator is also specified. This is why all MPI communication routines require both a rank and a communicator.

Thus, MPI functions identify processes by the *(communicator, rank)* pair. The communicator ensures that message passing occurs in the correct scope and group, avoiding interference between unrelated communication domains.

7.2.2 Point-to-Point Communication Example

Consider the following simple SPMD-style MPI program executed by two processes. It demonstrates basic point-to-point communication using `MPI_Send` and `MPI_Recv`.

```
comm = MPI_COMM_WORLD;
MPI_Comm_rank(comm, &rank);

for (i = 0; i < n; i++) a[i] = 0;

if (rank == 0) {
    MPI_Send(a + n / 2, n / 2, MPI_INT, 1, tag, comm);
} else {
    MPI_Recv(b, n / 2, MPI_INT, 0, tag, comm, &status);
}

/* Process local arrays a or b */

/* Perform reverse communication here if needed */
```

Listing 7.1: Simple MPI Point-to-Point Example

Assume that there are only two processes in the world communicator `MPI_COMM_WORLD`, with no subgroups.

Explanation:

- `MPI_Comm_rank(comm, &rank)`: Retrieves the rank (ID) of the calling process within the communicator `comm`.
- `MPI_INT`, `MPI_DOUBLE`, etc., are MPI datatypes, also referred to as **type specifiers**.
- The array `a[]` is initialized with zeros. Then, process 0 sends the second half of array `a` (i.e., from index `n/2`) to process 1 using `MPI_Send`.
- Process 1 receives this data into its own array `b[]` using `MPI_Recv`.
- Both processes then perform computation on their local data independently.

Tags and Communication Scope:

- `tag` is a user-defined integer used to distinguish between messages (e.g., set to `5` in this case).
- `comm` refers to the communicator scope (here, `MPI_COMM_WORLD`).

Wildcards: MPI allows wildcard values in `MPI_Recv` for flexible message matching:

- `MPI_ANY_SOURCE`: Allows receiving from any source process, regardless of rank.
- `MPI_ANY_TAG`: Allows receiving messages of any tag, useful when message ordering is not important.

Common MPI Utility Functions:

- `MPI_Init`: Initializes the MPI environment. Must be called before any MPI function.
- `MPI_Finalize`: Cleans up and shuts down the MPI environment. Must be called at the end.
- `MPI_Comm_size(comm, &size)`: Determines the number of processes in the communicator `comm`.
- `MPI_Comm_rank(comm, &rank)`: Retrieves the rank of the calling process in the communicator.
- `MPI_Wtime()`: Returns an elapsed wall-clock time in seconds. Useful for performance timing.

7.2.3 Example 1: Finding Maximum Using Two Processes

Consider the following C program that finds the maximum element in an array using two MPI processes.

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    int N;
    int *A, *local_array;
    int max, local_max, rank1_max, i;
    MPI_Comm comm;
    MPI_Status status;
    int rank, size;
    int LARGE_NEGATIVE_NUMBER = -999999;

    MPI_Init(&argc, &argv);

    comm = MPI_COMM_WORLD;
    MPI_Comm_size(comm, &size);
    MPI_Comm_rank(comm, &rank);

```

```
    if (size != 2) {
        printf("This program works with only two processes.\n");
        exit(0);
    }

    if (rank == 0) {
        /* Read N from console */
        scanf("%d", &N);
        MPI_Send(&N, 1, MPI_INT, 1, 5, comm);

        A = (int*)malloc(N * sizeof(int));
        /* Initialize the array A */
        for (i = 0; i < N; i++) A[i] = rand() % 100;

        local_array = (int*)malloc((N / 2) * sizeof(int));
        for (i = 0; i < N / 2; i++) {
            local_array[i] = A[i];
        }

        MPI_Send(A + N / 2, N / 2, MPI_INT, 1, 10, comm);
    } else {
        MPI_Recv(&N, 1, MPI_INT, 0, 5, comm, &status);

        local_array = (int*)malloc((N / 2) * sizeof(int));
        MPI_Recv(local_array, N / 2, MPI_INT, 0, 10, comm, &status);
    }

    local_max = LARGE_NEGATIVE_NUMBER;
    for (i = 0; i < N / 2; i++) {
        if (local_array[i] > local_max) {
            local_max = local_array[i];
        }
    }

    if (rank == 1) {
        MPI_Send(&local_max, 1, MPI_INT, 0, 15, comm);
    } else {
        max = local_max;
        MPI_Recv(&rank1_max, 1, MPI_INT, 1, 15, comm, &status);
        if (rank1_max > max) {
            max = rank1_max;
        }
        printf("Maximum number is %d\n", max);
    }

    MPI_Finalize();
}
```

Listing 7.2: MPI Maximum Element Using Two Processes

Explanation:

- The program is written in the SPMD style: all processes execute the same code but operate on different data.
- `MPI_Init` initializes the MPI environment.

- `MPI_COMM_WORLD` is the default communicator including all spawned processes.
- Each process determines its **rank** and the **size** of the communicator using `MPI_Comm_rank` and `MPI_Comm_size`, respectively.

Program Flow:

- If the number of processes is not 2, the program terminates.
- Rank 0:
  - Reads the array size `N` from the console and sends it to rank 1.
  - Dynamically allocates and initializes the full array `A` of size `N`.
  - Copies the first half into `local_array`, and sends the second half to rank 1.
- Rank 1:
  - Receives `N` and then receives the second half of the array into its own `local_array`.
- Both ranks compute the local maximum of their respective halves.
- Rank 1 sends its local maximum to rank 0.
- Rank 0 compares both values and prints the final global maximum.

MPI Functions Used:

- `MPI_Init`, `MPI_Finalize`: To initialize and finalize MPI.
- `MPI_Comm_rank`, `MPI_Comm_size`: To get the rank and number of processes.
- `MPI_Send`, `MPI_Recv`: For point-to-point communication.

Note: Use of consistent message tags (e.g., 5, 10, 15) helps prevent mismatches in sender–receiver logic. In real-world practice, error handling should also be included.

7.3 Buffering and Safety

7.3.1 Blocking Communications

One of the important aspects of point-to-point communication in MPI is the use of send and receive buffers. To understand this better, consider the following sequence:

Suppose a user program invokes `MPI_Send(user_buf, ...)`. Here, `user_buf` refers to the memory buffer from which data is to be sent. This call is internally handled by the MPI library linked to the user program. The MPI library is responsible for managing the hardware-level communication and typically interacts with the underlying system through functions such as `tcp_send()` using TCP/IP protocols.

At the receiver's end, there is a corresponding call to `MPI_Recv`, which is also handled internally by the MPI library using a lower-level receive mechanism. MPI libraries maintain internal buffers like `send_buf` and `recv_buf`. The data in `user_buf`

is first copied into the MPI library's `send_buf`, and then sent to the destination. At the receiving side, the incoming data is first stored in the `recv_buf`, and later copied to the user-defined `user_buf`.

The `MPI_Send` function spawns internal threads within the MPI library that handle additional responsibilities such as inserting message headers, acknowledgments, and invoking the lower-level network send. These internal threads allow the main thread (that called `MPI_Send`) to return immediately after the user buffer is safely copied to the MPI library buffer, thereby allowing the user program to continue its execution.

Similarly, `MPI_Recv` is called blocking because it returns only after the data from the internal receive buffer has been safely copied into the user-provided buffer.

Case Analysis:

Case 1: Safe Communication Pattern

- **Process 0:** `MPI_Send` followed by `MPI_Recv`
- **Process 1:** `MPI_Recv` followed by `MPI_Send`

This is a well-structured program where each `MPI_Send` has a matching `MPI_Recv`. This pattern works safely and deterministically, avoiding any deadlocks.

Case 2: Deadlock Situation

- **Process 0:** `MPI_Recv` followed by `MPI_Send`
- **Process 1:** `MPI_Recv` followed by `MPI_Send`

Here, both processes start by attempting to receive from the other. Since neither process sends data initially, both are blocked, waiting for data that never arrives. This results in a classic **deadlock**.

Case 3: Unsafe Pattern (May Work or May Deadlock)

- **Process 0:** `MPI_Send` followed by `MPI_Recv`
- **Process 1:** `MPI_Send` followed by `MPI_Recv`

This situation can potentially work. When `MPI_Send` is invoked by Process 0, it immediately copies data from the user buffer to the internal `send_buf` and returns. While the data is being transmitted in the background, Process 0 continues execution and invokes `MPI_Recv`. If Process 1 performs similar steps, the system may make progress.

However, whether this actually works depends on the size of the message and the available space in the MPI library's internal `send_buf`. If the internal buffer is large enough to accommodate the entire user message, the function call completes and the program proceeds.

On the other hand, if the message is too large to fit into the internal buffer, then only part of it is copied. To make space, the MPI library needs to first send the copied part, which can only happen if the receiver is ready. But since the receiver itself is trying to send, neither process proceeds—resulting again in a **deadlock**.

Therefore, this is considered an **unsafe communication pattern** and should be avoided in practice.

Summary:

- **Blocking Communication** implies that the user buffer must be safely copied to the internal buffer (for `MPI_Send`) or vice versa (for `MPI_Recv`) before the function returns.
- Communication safety depends on the order of send and receive calls and the available space in the MPI internal buffers.
- **Deadlocks** occur when processes are mutually waiting on each other to receive messages, especially when both use blocking receives before sends.
- **Best Practice:** Always write send-receive pairs such that at least one side is guaranteed to post a receive before the corresponding send can block due to buffer capacity limits.

7.3.2 Non-Blocking Communications

MPI also supports **non-blocking communication** primitives. In non-blocking functions, the control returns to the user program *immediately*, without waiting for the communication (send or receive) to complete. Specifically, these functions do not even wait for the data to be copied into the internal send/receive buffers of the MPI library. Instead, MPI spawns background threads that handle the communication, and the program continues executing subsequent instructions.

This implies a key restriction: after issuing a non-blocking communication call, the programmer must not reuse or modify the send or receive buffer until the communication has been explicitly completed. Doing so may result in undefined behavior because the communication may still be in progress.

Recall that in **blocking** communication:

- For `MPI_Send`, the function returns only after the user buffer has been safely copied to the internal `send_buf`.
- For `MPI_Recv`, the function returns only after the data has been received from the internal `recv_buf` into the user-provided buffer.

In contrast, in non-blocking communication:

- `MPI_Isend` spawns a background thread immediately and returns, even before the user buffer is copied.
- `MPI_Irecv` returns before any data is received.

This early return allows the programmer to perform other computations that do not depend on the communication—thereby overlapping communication with computation and improving performance.

Non-Blocking Functions:

- `MPI_Isend(buf, count, datatype, dest, tag, comm, request)`
- `MPI_Irecv(buf, count, datatype, source, tag, comm, request)`

- `MPI_Wait(request, status)`
- `MPI_Test(request, flag, status)`
- `MPI_Request_free(request)`

When `MPI_Isend` or `MPI_Irecv` is called, a **request handle** is returned. This handle must be passed to the `MPI_Wait` function to ensure completion of the operation. The user may perform independent computations between the posting (`MPI_Isend`/`MPI_Irecv`) and the completion (`MPI_Wait`) of the communication.

This approach enables **asynchronous communication**, promoting better resource utilization and concurrency. The combination of `Isend`/`Irecv` with `Wait` is often referred to as **posting** and **completion**.

Other Non-Blocking Collective Utilities: When a process posts multiple non-blocking operations, the following utility functions can be used to monitor and complete them:

- `MPI_Waitany(count, array_of_requests, index, status)`
- `MPI_Testany(count, array_of_requests, index, flag, status)`
- `MPI_Waitall(count, array_of_requests, array_of_statuses)`
- `MPI_Testall(count, array_of_requests, flag, array_of_statuses)`
- `MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)`
- `MPI_Testsome(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)`

These functions allow the programmer to determine the completion status of any, all, or some non-blocking requests. For example:

- `MPI_Testany` checks if any one of the non-blocking operations is complete.
- `MPI_Testall` checks if all posted operations are complete.
- `MPI_Waitsome` waits for at least one of them to complete.

This granularity is useful when managing multiple communications simultaneously.

Illustrative Cases:

- **Case 1:** Consider the scenario where
  - Process 0: `MPI_Send`(msg1), `MPI_Send`(msg2)
  - Process 1: `MPI_Irecv`(msg2), `MPI_Irecv`(msg1)

Here, the ordering mismatch is resolved by the use of non-blocking receives. Although Process 0 performs blocking sends, the non-blocking receives allow Process 1 to proceed immediately and post both receives. The messages will eventually match correctly. This program will execute safely regardless of buffer sizes.

- **Case 2:** Now consider:
  - Process 0: `MPI_Isend`, `MPI_Recv`
  - Process 1: `MPI_Isend`, `MPI_Recv`

  This scenario is safe because both sends are non-blocking. The programs will post their sends and then proceed to blocking receives. Since the sends are non-blocking, no deadlock occurs and the `Recv` calls will eventually succeed. This shows that many unsafe blocking communication patterns can be converted to safe versions using non-blocking functions.

Summary:

- Non-blocking communication enables overlapping communication with computation.
- Buffers used in non-blocking operations must not be accessed until completion via `MPI_Wait` or `MPI_Test`.
- It improves performance but requires careful handling to ensure correctness.
- Complex communication patterns involving multiple messages can be effectively managed using `Waitany`, `Waitall`, and their variants.

7.3.3 Example: Finding a Particular Element in an Array

Consider the problem of searching for a particular element in a large array, similar to a database search. Assume that all elements in the array are unique and that only two processes are available. The array is divided equally: the first half remains with process 0, and the second half is sent to process 1. The key element to search for is also sent to process 1. Both processes then search independently in parallel. If any process finds the element, it immediately informs the other and the program terminates.

This is a typical example of an **SPMD** (Single Program, Multiple Data) program. The communication pattern is inherently non-deterministic because only one of the two processes will find the element, and we cannot predict which one. Therefore, non-blocking communication is ideal in this case.

```
#include "mpi.h"
int main(int argc,char** argv){
    ...
    int other_i, other_rank, flag;
    MPI_Request request;
    ...
    other_rank = (rank == 0) ? 1 : 0;

    // Post non-blocking receive
```

```
    MPI_Irecv(&other_i, 1, MPI_INT, other_rank, 10, comm, &request);

    for(i = 0; i < N/2; i++) {
        if(A[i] == elem) {
            // Element found: inform the other process
            MPI_Send(&i, 1, MPI_INT, other_rank, 10, comm);
            if(rank == 0) {
                global_pos = i;
            }
            // Cancel the posted receive
            MPI_Cancel(&request);
            break;
        } else {
            // Check if other process has found it
            MPI_Test(&request, &flag, &status);
            if(flag == 1) {
                MPI_Wait(&request, &status);
                if(rank == 0) {
                    global_pos = other_i + N/2;
                }
                break;
            }
        }
    }

    if(rank == 0) {
        printf("Found the element %d in position %d\n", elem,
            global_pos);
    }

    MPI_Finalize();
}
```

Explanation:

- Each process computes the rank of the other using `(rank == 0) ? 1 : 0` and stores it in `other_rank`.
- A non-blocking receive (`MPI_Irecv`) is posted to receive the index from the other process if it finds the element.
- Each process begins searching through its half of the array. If a match is found:
  - It sends the index to the other process using a blocking `MPI_Send`.
  - Updates the global position (if it is rank 0).
  - Cancels the posted non-blocking receive using `MPI_Cancel`, since no message will be received.
- If the current process does not find the element, it periodically tests using `MPI_Test` whether a message has been received from the other process. If so:
  - The message is awaited with `MPI_Wait` to ensure completion.
  - The global position is updated accordingly.

Why Non-Blocking is Better Here: Using `MPI_Irecv` instead of a blocking `MPI_Recv` allows each process to continue searching independently without waiting. If we had used blocking receives:

- The process would be forced to wait at the `Recv` call, even if no message was ever going to be sent.
- We would have to move the `MPI_Recv` to the end of the loop and communicate at every iteration, even if the element was not found—leading to excessive communication.

Thus, **non-blocking communication is highly useful in non-deterministic scenarios**, where a process may or may not send a message. It enables each process to continue working independently while remaining responsive to updates from others, avoiding deadlock and unnecessary message overhead.

7.4 Communication Modes

MPI supports multiple communication modes, each offering a different trade-off between performance, safety, and synchronization semantics. So far, we have primarily discussed the standard communication mode, where a `MPI_Send` may complete either before or after the corresponding `MPI_Recv` is executed on the receiving side, depending on buffer availability.

Standard Mode

In **standard mode** (`MPI_Send`), the send can be initiated before or after the corresponding receive. If the MPI implementation has sufficient internal buffer space, the send may complete even before the receive is posted. However, if the message size exceeds the available buffer, then the send operation blocks until the matching receive is called and starts consuming the data.

Buffered Send

MPI provides support for **buffered communication** through the `MPI_Bsend` function. Here, the user explicitly allocates a buffer of sufficient size and attaches it to MPI using `MPI_Buffer_attach`. This user-provided buffer is then used by the MPI library for send operations. Because of this, the send operation can always complete immediately after copying data to the attached buffer, regardless of whether the receive has been posted. This ensures communication safety and eliminates dependency on internal MPI buffers.

Synchronous Send

In **synchronous mode** (`MPI_Ssend`), the send can also be initiated before or after the receive is posted. However, the key difference is that the send completes only when the matching receive has started and has begun consuming the message. This provides a stronger synchronization guarantee. If `MPI_Ssend` returns, you can be certain that the corresponding `MPI_Recv` has been initiated, making it particularly useful when strict coordination between processes is required.

Ready Send

The **ready mode** (`MPI_Rsend`) offers a highly efficient communication path but must be used with caution. It assumes that the corresponding receive has already been posted. If this assumption is violated (i.e., the send is executed before the matching receive), the program results in undefined behavior and may even crash. This mode is especially suitable for large messages, where bypassing internal buffering and transferring directly from sender to receiver is more efficient.

Mode	Start Condition	Completion Condition
`Standard (MPI_Send)`	Before or after `Recv`	Completes before `Recv` (if buffer exists), or after `Recv` otherwise
`Buffered (MPI_Bsend)`	Before or after `Recv`	Always completes before `Recv` (as buffer is attached by user)
`Synchronous (MPI_Ssend)`	Before or after `Recv`	Completes only when `Recv` starts consuming the message
`Ready (MPI_Rsend)`	**Only after** `Recv` is posted	Completes after `Recv` has started

Table 7.1: Summary of MPI Communication Modes

Table 7.1 summarizes the differences between the four communication modes supported by MPI. The choice of mode can significantly impact program correctness, safety, and performance. Buffered and ready sends are typically used for performance-critical applications with strict control over program flow, while standard and synchronous sends offer safer defaults.

7.5 Collective Communications

Collective communications refer to communication operations that involve a group of processes, rather than just two. These operations are extremely useful because a significant portion of parallel programs written using MPI rely heavily on collective communication for coordination, data distribution, and aggregation.

7.5.1 Allgather, Allgatherv

Let us now look at an example of matrix-vector multiplication. Suppose we wish to multiply an $M \times N$ matrix A with a $N \times 1$ vector b, resulting in a $M \times 1$ vector x, i.e., $Ab = x$.

Now, using an MPI domain decomposition strategy, suppose we divide the matrix and the vectors **across the rows**, such that the first set of rows belongs to process P_0, the next set belongs to process P_1, and so on as shown in figure 7.1.

Let us consider the rows assigned to some process P_i. To compute its portion of the result vector x, process P_i must multiply its local rows of A with the **entire** vector b. Note the following:

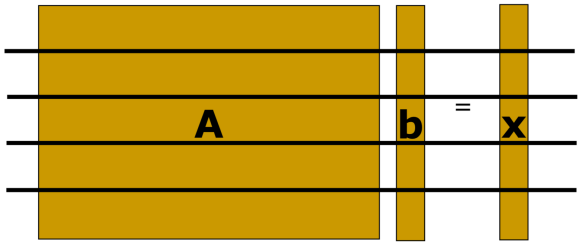


Figure 7.1: Row-wise Matrix-Vector Multiplication

- The matrix rows are already local to P_i, so no communication is needed for accessing the matrix.
- The corresponding output elements of x that result from multiplying the local rows are also stored locally, and thus need not be communicated to other processes.
- However, the input vector b is **distributed across all the processes**, so process P_i only has access to a part of b. Therefore, each process must obtain the **entire** b vector to perform its computation.

One solution is to use point-to-point communication primitives such as `MPI_Send` and `MPI_Recv`. Each process could, in a loop, send its part of the b vector to all the other processes and receive the missing parts from them. While this is logically correct, it results in a verbose and cumbersome implementation.

Instead, observe the communication pattern here: all processes require **the same global data** — the full b vector — which is initially distributed among them. This is a textbook use case for the **all-gather** communication pattern.

In `MPI_Allgather`, each process contributes its piece of data (in this case, a chunk of the b vector), and all processes collect the entire assembled vector. That is, all processes end up with the full b vector. This makes `MPI_Allgather` the ideal communication primitive for this scenario.

Thus, using `MPI_Allgather`, the row-wise matrix-vector multiplication can be implemented efficiently and clearly in MPI.

```
1 MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
    recvtype, comm)
```

The entire communication pattern involving sending and receiving data between all processes can be captured using a single collective communication call — namely, `MPI_Allgather`. Instead of writing a for loop of `MPI_Send` and `MPI_Recv` calls, each process now simply invokes a one-line function call with its local data. This local data can be a single element, an array, or even a matrix. For example, process 0 may send its portion of the data (say, A_0) by filling the appropriate `sendbuf`, `sendcount`, and `sendtype`. Simultaneously, all processes will gather the full dataset into their local `recvbuf`, with corresponding `recvcount` and `recvtype`. Thus, all processes allocate a receive buffer, and after the `MPI_Allgather` call, this buffer will contain the entire combined dataset from all processes. It is as shown in the figure 7.2. This function must be called by all processes that are part of the communicator `comm`, as it is a collective communication operation.

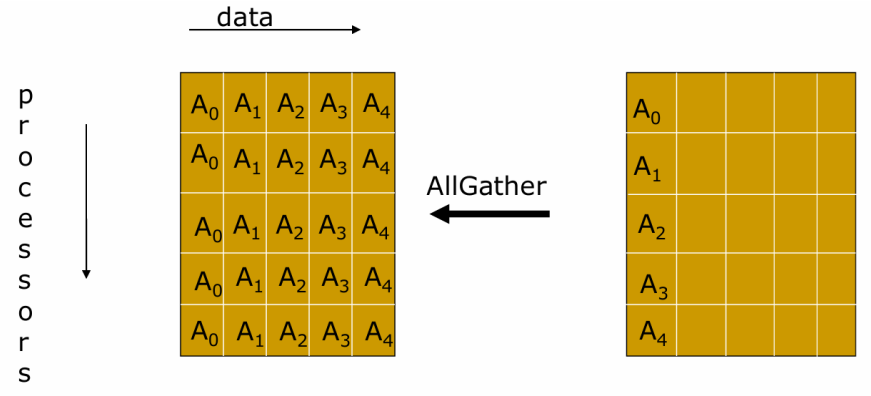


Figure 7.2: `MPI_Allgather` operation

There is also a vector variant called `MPI_Allgatherv`, which allows different processes to contribute different amounts of data:

```
MPI_Allgatherv(sendbuf, sendcount, sendtype,
               recvbuf, recvcounts, displs, recvtype, comm);
```

Here, `recvcounts` is an array that specifies how much data each process contributes, and `displs` specifies where in the receive buffer each contribution should be placed.

Let us now consider an example using `MPI_Allgather` to perform matrix-vector multiplication:

```
MPI_Comm_size(comm, &size);
MPI_Comm_rank(comm, &rank);
nlocal = n / size;

MPI_Allgather(local_b, nlocal, MPI_DOUBLE, b, nlocal, MPI_DOUBLE,
    comm);

for (i = 0; i < nlocal; i++) {
    x[i] = 0.0;
    for (j = 0; j < n; j++) {
        x[i] += a[i*n + j] * b[j];
    }
}
```

In this code:

- Each process first determines the total number of processes (`size`) and its own rank (`rank`) in the communicator.
- The matrix A is assumed to have n total rows and is divided evenly among the processes, so each process works on $n_{\text{local}} = n/\texttt{size}$ rows.
- Each process possesses a local portion of the input vector b (stored in `local_b`) and needs the full vector b to compute its portion of the result vector x.
- To achieve this, all processes perform a single `MPI_Allgather` call to collect the full b vector.

The final loop then performs the dot product between each local row of matrix A and the global vector b, and stores the result in the local portion of x.

Using collective communication like `MPI_Allgather` significantly reduces programming complexity. More importantly, it gives the MPI library knowledge of the global communication pattern, allowing it to choose an efficient communication algorithm optimized for the underlying architecture. In contrast, if only point-to-point communication (i.e., `MPI_Send` and `MPI_Recv`) is used, the MPI library cannot infer any global pattern and thus cannot optimize the communication strategy.

Note: The order in which data is stored in the receive buffer is based on process ranks. That is, data from process 0 appears first, followed by data from process 1, and so on.

7.5.2 Reduce, AllReduce

Suppose we now divide the matrix A column-wise, such that the first set of columns resides in process 0, the second set in process 1, and so on. Since b and x are vectors, they are still divided row-wise as shown in figure 7.3. In this layout, the communication

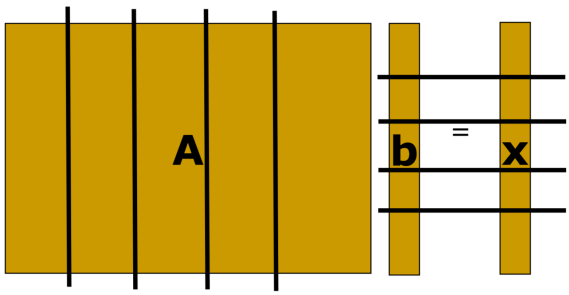


Figure 7.3: Column-wise matrix-Vector Multiplication

requirements change. For computing the inner product (i.e., matrix-vector multiplication), each processor already owns the portion of vector b corresponding to the local columns of matrix A. Therefore, **no communication is required beforehand** — each process can compute its local contribution to the final result independently.

However, once local computations are complete, communication is required to **aggregate** these local results. Each processor computes a partial result (a local vector), and these must be **summed across all processes** to obtain the final result vector x. This is an example of a **reduction** communication pattern, where all processes contribute values that are combined using some operator (e.g., sum, max, min), and the final result is stored in a particular process.

Here is the MPI-based implementation of this scheme:

```
MPI_Comm_size(comm, &size);
MPI_Comm_rank(comm, &rank);
nlocal = n / size;

/* Compute partial dot-products */
for (i = 0; i < n; i++) {
    px[i] = 0.0;
    for (j = 0; j < nlocal; j++) {
        px[i] += a[i * nlocal + j] * b[j];
    }
}
```

As we can see, each process computes its part of the dot product and stores it in the local vector `px`. No communication is needed in this step.

To aggregate the local vectors into the final result x, MPI provides collective reduction operations such as `MPI_Reduce`:

```
MPI_Reduce(px, x, n, MPI_DOUBLE, MPI_SUM, 0, comm);
```

This function performs the following:

- `px` is the local array being sent (partial result from each process).
- `x` is the receive buffer in the root process (rank 0 in this example), where the final result will be stored.
- `n` is the number of elements in the arrays.
- `MPI_DOUBLE` specifies the data type.
- `MPI_SUM` is the operation used to combine data — in this case, addition.
- `0` is the rank of the root process where the result is stored.
- `comm` is the communicator.

MPI also provides other predefined operations such as:

- `MPI_MAX` — for computing the maximum of corresponding elements,
- `MPI_MIN` — for computing the minimum,
- `MPI_PROD` — for product,
- and many more.

In addition, MPI allows users to define **custom reduction operations** using the `MPI_Op_create` interface. This gives the programmer flexibility to implement and register user-defined functions that operate element-wise on arrays and pass them as operators in collective calls like `MPI_Reduce` or `MPI_Allreduce`.

Column-wise decomposition simplifies the local computation of matrix-vector multiplication but requires a global reduction to finalize the result. MPI's reduction operations such as `MPI_Reduce` help implement this efficiently and scalably.

In certain situations, it is desirable for the result of a reduction operation to be made available to **all** participating processes, rather than just a single root process. In such cases, one uses the collective communication:

```
MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)
```

Here, there is **no root process**, and the final reduced result is stored in the `recvbuf` of every process. This is especially useful in parallel applications where all processes need the final computed result (e.g., for norm calculations, convergence checks, etc.).

7.5.3 Scatter, Scatterv, Gather, Gatherv

Another essential collective communication is the `MPI_SCATTER` operation. In this pattern, a single process (typically rank 0) possesses a large buffer of data and wants to distribute **equal-sized chunks** of it to all processes in the communicator group.

```
MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount,
    recvtype, root, comm)
```

All processes must call this function collectively. The root process provides the data to distribute in `sendbuf`, and each process receives its chunk in `recvbuf`. This operation is widely used during the initialization phase of parallel programs when data is to be distributed from a central source.

When the data to be distributed to each process is of **different sizes**, we use the more general variant:

```
MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf,
    recvcount, recvtype, root, comm)
```

Here, `sendcounts` is an array indicating how many elements each process should receive, and `displs` gives the starting offset for each process in the `sendbuf`.

The `MPI_GATHER` collective communication operation is similar to `MPI_SCATTER`, but in the reverse direction. In this pattern, we want to gather data from all processes in the communication group into a single root process.

```
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
    recvtype, root, comm);
```

Here,

- Each process sends `sendcount` elements from its `sendbuf` of type `sendtype`.
- The root process receives all the data in its `recvbuf`, expecting `recvcount` elements from each process, all of type `recvtype`.
- `root` is the rank of the process that gathers the data.
- All processes must call this function, but only the root process needs to provide valid `recvbuf`.

The `MPI_GATHERV` operation is a more flexible variant of `MPI_GATHER`, where each process can send a different number of elements. This is particularly useful when the data sizes to be gathered from different processors are not uniform.

```
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
    displs, recvtype, root, comm);
```

Here,

- `recvcounts` is an array of integers specifying the number of elements expected from each process.
- `displs` is an array of displacements (i.e., starting indices in `recvbuf`) where data from each process should be placed.

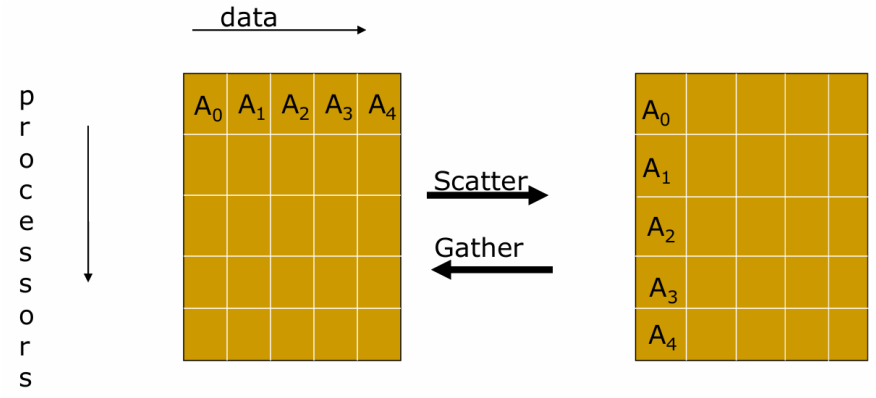


Figure 7.4: `Scatter` and `Gather` collective communication operations

- This allows variable-length contributions from each process to be gathered efficiently into the root's receive buffer.

Let us return to our example of column-wise matrix-vector multiplication. After all processes compute their local partial dot products and store them in `px`, we need to reduce them and distribute the final result. This can be achieved as follows:

```
/* Summing the dot-products */
MPI_Reduce(px, fx, n, MPI_DOUBLE, MPI_SUM, 0, comm);

/* Now all values of x are stored in process 0.
   Need to scatter them to all processes */
MPI_Scatter(fx, nlocal, MPI_DOUBLE, x, nlocal, MPI_DOUBLE, 0, comm);
```

Here, we first use `MPI_Reduce` to gather and sum all partial dot products into the buffer `fx` in process 0. Since we want the final vector x to be distributed across all processes, we follow it up with a `MPI_Scatter` to send n_{local} values of x to each process.

Alternatively, instead of gathering all values in process 0 and then redistributing them, we can use a clever method where each segment of the partial result `px` is reduced directly into the corresponding process:

```
for (i = 0; i < size; i++) {
    MPI_Reduce(px + i * nlocal, x, nlocal, MPI_DOUBLE, MPI_SUM, i,
        comm);
}
```

This results in each process directly receiving its portion of the reduced vector x, eliminating the need for an explicit scatter operation afterward.

Collective communications offer two major advantages:

1. **Reduced Programming Complexity:** Using one-line collective calls like `MPI_Allreduce`, `MPI_Scatter`, etc., avoids complex loops involving point-to-point communication.

2. **Performance Optimization:** When using collective calls, the MPI library can detect communication patterns and apply optimized algorithms (such as tree-based or ring-based communication) for better efficiency. In contrast, such optimization is not possible with arbitrary combinations of `MPI_Send` and `MPI_Recv`.

Up until **MPI-2**, only **blocking collective communication** routines were available. Starting from **MPI-3**, MPI introduced **non-blocking collective communication** routines. These behave similarly to their point-to-point counterparts like `MPI_Isend` and `MPI_Irecv`, allowing the program to initiate a communication and proceed with computation before explicitly waiting for its completion.

Just as `MPI_Reduce` is used for a blocking reduction, we can now use:

```
MPI_Ireduce(..., &request)
```

to perform a non-blocking reduction, where `request` is an `MPI_Request` object that can later be passed to `MPI_Wait` or `MPI_Test` to complete the communication.

Note the following important characteristics:

- Non-blocking collectives in MPI-3 support only the **standard mode** of communication. Unlike point-to-point routines, **buffered**, **synchronous**, or **ready** modes are *not supported* for collective routines.
- There is **no need for message tags** in collective communication. In point-to-point communication, message tags are used to provide scope or context to distinguish between multiple messages. However, collective operations are inherently well-scoped and synchronized across the communicator group. Therefore, the scope is implicit and tags are unnecessary.

We also observed that many collective communication routines have both a **simple variant** (where equal-sized data is involved for all processes) and a **vector variant** (where data sizes may differ across processes). For example:

- `MPI_Allgather` and `MPI_Allgatherv`
- `MPI_Scatter` and `MPI_Scatterv`
- `MPI_Gatherv`, `MPI_Reduce_scatter`, etc.

Further, it is important to distinguish between routines that require a **root process** and those that do not:

- Collectives like `MPI_Scatter`, `MPI_Gather`, and `MPI_Reduce` require a `root` process argument, which acts as the sender or receiver of data.
- On the other hand, collectives like `MPI_Allgather`, `MPI_Allreduce`, and `MPI_Bcast` do **not require** a root argument—they involve all processes equally.

Communication Pattern Classification:

- **One-to-all:** A single process sends data to all others.
  Example: `MPI_Bcast`, `MPI_Scatter`
- **All-to-one:** All processes send data to one process.
  Example: `MPI_Reduce`, `MPI_Gather`
- **All-to-all:** Every process sends and receives data from every other process.
  Example: `MPI_Allgather`, `MPI_Allreduce`, `MPI_Alltoall`

This classification helps in identifying the nature of communication and optimizing the parallel program based on the data flow requirement.

7.5.4 Barrier

We have already encountered the utility of barriers in earlier examples, particularly in the context of the grid solver, where synchronization between iterations was crucial. The `MPI_Barrier` function provides a mechanism to ensure that all processes in a communicator reach a common synchronization point before any of them can proceed further. This is especially useful in iterative algorithms where the correctness of the next iteration depends on the completion of the previous one.

When a process calls `MPI_Barrier`, it will be blocked until all other processes in the communicator have also called `MPI_Barrier`. Once all processes have reached this point, they are released simultaneously. This ensures coordinated progress across all processes.

Implementing a Barrier using Point-to-Point Communication Even though MPI provides `MPI_Barrier` as a collective operation, it's important to note that any collective communication can, in principle, be implemented using point-to-point primitives like `MPI_Send` and `MPI_Recv`.

One naïve way to implement a barrier might be as follows:

- Process P_0 sends a "finished" message to P_1.
- P_1 receives this message, performs its barrier work, then sends a message to P_2, and so on.
- Eventually, P_{n-1} receives from P_{n-2}, completes its work, and then sends an acknowledgment message back to all the previous processes.

This approach involves $n-1$ forward messages and $n-1$ backward acknowledgments, resulting in $2(n-1)$ sequential point-to-point communications.

Time Complexity: The time complexity of this naïve linear chain barrier is $O(n)$, where n is the number of processes. This becomes a performance bottleneck for large-scale parallel programs.

Can We Do Better? Butterfly Barrier (Eugene Brooks) Yes—we can reduce the time complexity from $O(n)$ to $O(\log_2 n)$ using a communication pattern called the **Butterfly Barrier**, proposed by Eugene Brooks.

This algorithm leverages the fact that $\log_2 n$ rounds of pairwise exchanges are sufficient for all processes to share synchronization status. In each round r, every process communicates with a partner process determined by flipping the r-th least significant bit of its rank. After $\log_2 n$ such rounds, all processes are guaranteed to have participated in a global barrier.

We will describe and illustrate this butterfly pattern in detail in the next subsection.

Butterfly Barrier Implementation

The butterfly barrier algorithm was proposed by Eugene Brooks II and is a classic example of an efficient synchronization method. For n processes, it requires at most $O(2\log_2 n)$ pairwise synchronizations in the worst case. This approach was originally designed for shared-memory multiprocessors but is widely applicable due to its scalability and simplicity.

Algorithm Idea: The core idea is that in each round k, process i synchronizes with process $i \oplus 2^k$, where $\oplus$ denotes the bitwise XOR operation. The algorithm proceeds in $\log_2 n$ rounds, with each round doubling the number of processes each participant is aware of having synchronized.

Let us consider an example with $n = 8$ processes (numbered 0 through 7). The synchronization proceeds as follows:

- **Stage 0:** Each process i synchronizes with $i \oplus 1$. This forms the pairs (0,1), (2,3), (4,5), and (6,7).
- **Stage 1:** Each process i synchronizes with $i \oplus 2$. Now, we get (0,2), (1,3), (4,6), and (5,7).
- **Stage 2:** Each process i synchronizes with $i \oplus 4$. This leads to (0,4), (1,5), (2,6), and (3,7).

At the end of each round, the synchronization knowledge propagates exponentially. For example, by the end of Stage 1, process 5 has directly synchronized with 4 and 7 and indirectly with 6 (via 7). By the end of Stage 2, every process has synchronized—directly or transitively—with all others. Therefore, for $n = 8$ processes, the barrier completes in $\log_2 8 = 3$ stages.

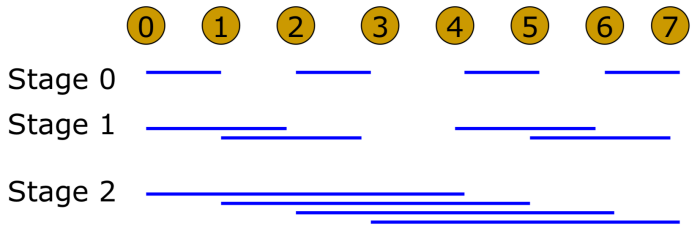


Figure 7.5: Butterfly barrier implementation for 8 processes

Shared Memory Implementation: In the original shared-memory setting, explicit communication was replaced by shared flags. For instance, process 0 sets a flag to indicate it has reached the barrier. Process 1, upon reading this flag, knows process 0 is ready, and vice versa. This read/write pattern continues in the butterfly shape across rounds, hence the name.

Non-Power-of-Two Case: The algorithm works optimally when n is a power of two. However, if n is not a power of two, we can emulate a power-of-two barrier by using the next higher power of two and assigning some processes to play the role of "dummy" partners. This leads to redundant communications and, in the worst case, increases the number of pairwise synchronizations to $O(2 \log_2 n)$.

The butterfly barrier is efficient, scalable, and well-suited to high-performance parallel computing systems. Compared to the naïve linear chain barrier (which takes $O(n)$ time), the butterfly barrier reduces synchronization complexity to $O(\log_2 n)$—a significant improvement for large-scale systems.

7.5.5 Broadcast

Another important collective communication operation in MPI is **broadcast**. In a broadcast operation, a single process (called the **root**) sends the same piece of data to all other processes within the communicator. This is often used in parallel programs when one process has input data that needs to be shared with all other processes before computation begins.

MPI provides a standard interface for broadcast:

```
MPI_BCAST(buffer, count, datatype, root, comm)
```

- `buffer`: The starting address of the data to be broadcast.
- `count`: Number of elements to broadcast.
- `datatype`: MPI datatype of the data.
- `root`: Rank of the process broadcasting the data.
- `comm`: Communicator involving all participating processes.

All processes in the communicator must call `MPI_BCAST`, regardless of whether they are the root or a receiver.

Although a naïve implementation would simply have the root process send the data individually to each of the $P-1$ processes (which takes $O(P)$ time), this is inefficient for large P. Instead, optimized implementations use a tree-based approach where communication happens in a hierarchical, staged fashion.

For example, using a binary tree scheme:

- In the first step, the root sends data to two processes.
- In the second step, those two send to two more, and so on.

This results in a time complexity of $O(\log_2 P)$ and significantly improves scalability.

MPI implementations typically use variations of binomial or pipeline trees for high efficiency, especially on large systems.

Important Notes:

- All processes must participate in the broadcast for it to succeed.
- The function is *synchronous*, in the sense that all processes wait for the operation to complete before proceeding.
- Unlike point-to-point communication, there is no need for tags in collective communications. The scope and context are implied by the collective function itself.

7.5.6 All-to-All

This operation is somewhat similar to `MPI_Allgather`, but with a key difference. In `MPI_Allgather`, each process sends data that is broadcast to all other processes, and all processes receive the same global result. However, in `MPI_Alltoall`, each process sends distinct data to every other process, and similarly receives distinct data from every other process. Thus, each process ends up with a unique collection of data from all other processes.

This pattern is particularly useful in scenarios where a distributed matrix needs to be transposed across processes, or when each process has some unique data for every other process. The result of this operation is conceptually equivalent to a transpose of a communication matrix. This is shown in figure 7.6, where each process has multiple data items, each destined for a different target.

```
MPI_Alltoall(sendbuf, sendcount, sendtype,
             recvbuf, recvcount, recvtype, comm)
```

The vector variant of this communication, where different amounts of data are sent to different processes, is:

```
MPI_Alltoallv(sendbuf, array_of_sendcounts, array_of_displs,
    sendtype,
              recvbuf, array_of_recvcounts, array_of_displs,
                  recvtype, comm)
```

Note that unlike collectives such as `MPI_Bcast` or `MPI_Reduce`, it is not possible to implement `MPI_Alltoall` with $O(\log_2 P)$ complexity. This is because each process must send a distinct message to every other process, which fundamentally requires $O(P)$ communications.

Naïve vs Optimized Communication Pattern:

A naïve implementation might look like the following:

```
// Naive All-to-All (sequential and bottlenecked)
for all processes i in order {
    if (i != my_proc) {
        send to process i;
        receive from process i;
    }
}
```

This approach introduces serialization and communication hotspots due to synchronization around a fixed global order. A better strategy is to "break symmetry" and have each process follow its own order in a round-robin fashion. This avoids bottlenecks and maximizes parallelism:

```
// Optimized All-to-All using ring pattern
for (i = 0; i < P; i++) {
    dest = (my_proc + i) % P;
    src  = (my_proc - i + P) % P;
    send to dest;
    receive from src;
}
```

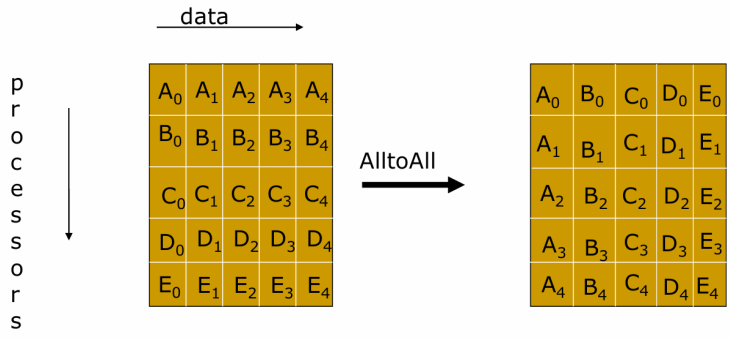


Figure 7.6: Schematic representation of All-to-All communication pattern

This circular scheduling pattern ensures full utilization of communication links and avoids contention, making the operation significantly faster on large systems.

7.5.7 ReduceScatter, Scan

The idea behind `MPI_Reduce_scatter` is to first perform a reduction across all processes and then scatter the results among the processes, as illustrated in Figure 7.7. This collective operation combines the functionalities of both `MPI_Reduce` and `MPI_Scatter`.

```
MPI_Reduce_scatter(sendbuf, recvbuf,
                   array_of_recvcounts, datatype, op, comm);
```

The `MPI_Scan` operation is known as a prefix sum or partial reduction operation. Unlike `MPI_Reduce`, where the final result is available at a single process, in `MPI_Scan`, each process receives the result of combining its data with the data from all lower-ranked processes. This is also called an **inclusive scan** or **inclusive prefix sum**. The behavior is illustrated in Figure 7.8.

```
MPI_Scan(sendbuf, recvbuf, count,
         datatype, op, comm);
```

For example:

- Process 0 has no prior process, so its result remains unchanged.
- Process 1 receives the combination (according to the given operator) of process 0 and process 1's data.
- Process 2 receives the combined result of processes 0, 1, and 2, and so on.

These kinds of operations are useful in many distributed algorithms, particularly in numerical computations, parallel prefix algorithms, and histogram computations.

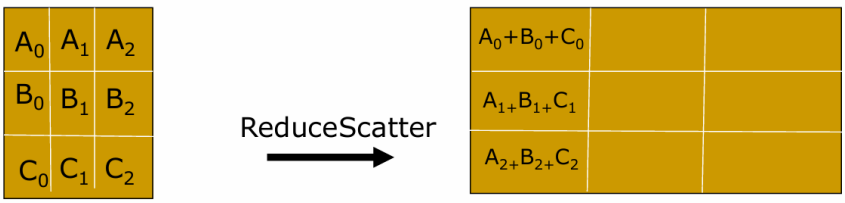


Figure 7.7: Conceptual diagram of `MPI_Reduce_scatter`

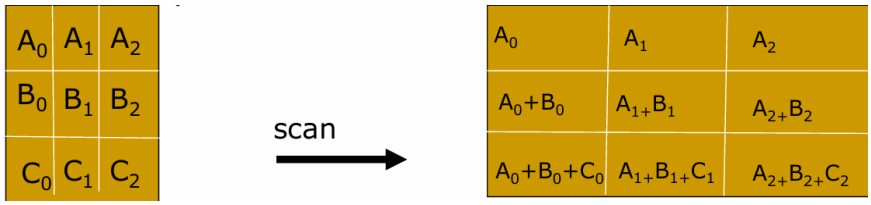


Figure 7.8: Inclusive prefix sum using `MPI_Scan`

Example of Ring-based `Allgather` Strategy:

In general, the efficiency of collective operations like `Allgather`, `Alltoall`, and `Scan` heavily depends on the hardware topology, communication latency, and network contention.

As an example, suppose we implement a ring-based `Allgather` operation. The idea is that in each stage, every process sends data to its right neighbor and receives data from its left neighbor. This process continues for $n-1$ steps if there are n processes, resulting in an $O(n)$ time complexity. At stage 0:

- Process 0 sends its data A_0 to Process 1.
- Process 1 sends its data A_1 to Process 2, and so on.

At the end of stage 0:

- Each process P_i now holds A_i and A_{i-1} (modulo n).

At stage 1:

- Each process sends what it received in the previous stage to its right neighbor.

This continues until all n pieces of data are circulated around the ring. Hence, it takes $O(n)$ stages. This is as shown in figure 7.9.

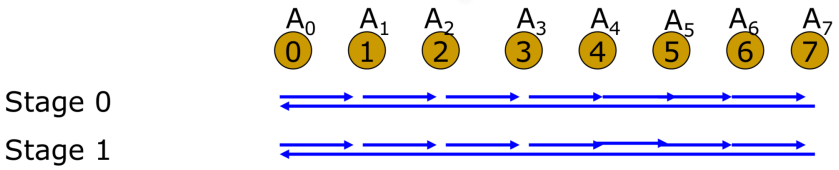


Figure 7.9: All gather operation for a ring-based topology

For a complete reference guide on MPI refer Snir [1998].

Chapter 8

Graphical Processing Units

A CPU typically has up to 32 cores, with each core supporting multiple threads—usually up to 8. It is well-suited for serial processing and is referred to as a multi-core architecture.

However, for highly parallel computations, we need architectures with significantly more cores. This is where Graphical Processing Units (GPUs) come into play. GPUs can have thousands of lightweight cores and are optimized for parallel processing. Consequently, they are often termed many-core architectures.

GPU cores are not as powerful as CPU cores individually, but they are designed to execute a vast number of simple operations simultaneously. In the early days, before 2014, NVIDIA GPUs were primarily used for gaming and video-related applications. These tasks involved multiple stages, such as pixel reading, shading, and rendering—each of which can be executed independently. For instance, the reading of one pixel does not depend on the reading of another, making it ideal for parallel execution. The high core count in GPUs allows such operations to be performed in real-time.

Because each GPU core is relatively lightweight and suited for simple computations rather than complex tasks, GPUs are used to accelerate operations that exhibit high degrees of parallelism. Typically, CPUs and GPUs coexist in a heterogeneous computing environment. GPUs are not used in a standalone manner. A program is usually executed on the CPU, which handles control and coordination, while delegating specific light-weight, data-parallel tasks to the GPU.

NVIDIA's GPU architecture supports CUDA (Compute Unified Device Architecture), a parallel computing platform and API developed by NVIDIA. CUDA enables developers to write programs in C/C++ that can offload computations to the GPU. It allows for efficient expression of data-parallelism and is widely used in domains requiring high-performance parallel computing.

Although GPUs were initially designed for graphical workloads, they have also proven to be highly efficient for numerical computations—particularly matrix operations. This makes them essential not only in gaming and graphics, but also in scientific computing, machine learning, and deep learning applications.

8.1 GPU Architecture

Figure 8.1: GPU Architecture

A schematic of the GPU architecture is shown in Figure 8.1. The architecture consists of the following major components:

- **TPC** – Texture Processing Clusters
- **SMX** – Streaming Multiprocessors
- **SP** – Single Precision Cores (GPU cores)
- **DRAM** – Device RAM (GPU memory)
- **ROP** – Render Output Units

GPU cores are referred to as **Streaming Multiprocessors** (SMX), while individual scalar cores inside an SMX are called **Single Precision** (SP) cores. Coarse-level parallel tasks are handled by the SMX units, which further distribute fine-grained parallel tasks to the SP cores. This structure supports both coarse-grain and fine-grain parallelism.

For instance, NVIDIA's Kepler architecture consists of 15 SMX units, each containing 192 SP cores, resulting in a total of $15 \times 192 = 2880$ SP cores. Although the clock speed of each core (approximately 745 MHz) is lower than that of a typical CPU core, the massively parallel structure of GPUs allows them to deliver high throughput for data-parallel tasks.

GPUs operate using the **SIMT** (Single Instruction Multiple Threads) model. Each SMX is equipped with its own 32-bit registers used to store the execution context of GPU threads. NVIDIA GPUs also feature a memory hierarchy including L2 cache, common memory, and local caches per SMX.

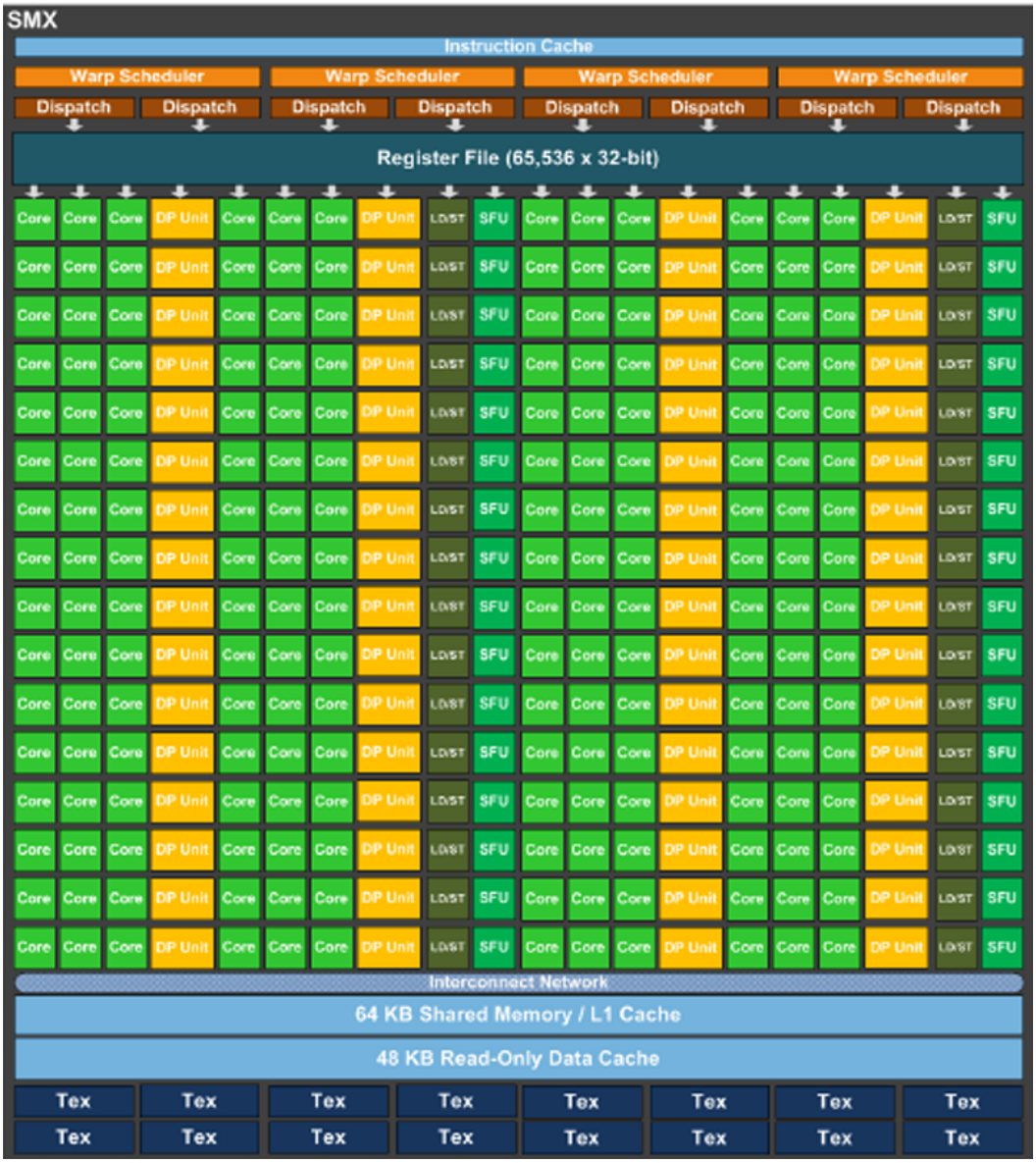


Figure 8.2: Architecture of a single SMX unit

Figure 8.2 illustrates the internal architecture of an SMX. It typically includes:

- 65536 registers
- 192 SP cores for single-precision computations
- 64 DP (Double Precision) cores
- 32 Special Function Units (SFUs) for evaluating transcendental functions
- 32 Load/Store units for data movement between global and shared memory
- 16 Texture Filtering Units
- Shared memory of size 64 KB (often configurable as 48 KB shared memory and 16 KB L1 cache, or vice versa)

The CUDA programming model provides explicit control over shared memory allocation. The programmer can allocate a portion of this 64 KB memory as shared memory (manually managed) and the rest as L1 cache (automatically managed by the hardware).

Computations are organised as blocks. Blocks are given to SMX. A block consists of many threads and threads are given to SP cores.

8.2 CUDA Memory Spaces

Consider the figure 8.3 of CUDA Memory Spaces. CPU pushes the relevant data to Global memory to be executed on GPU. It spawns GPU kernels/functions which are executed on SMX. Each thread in SMX loads data from global memory/device memory to shared memory.

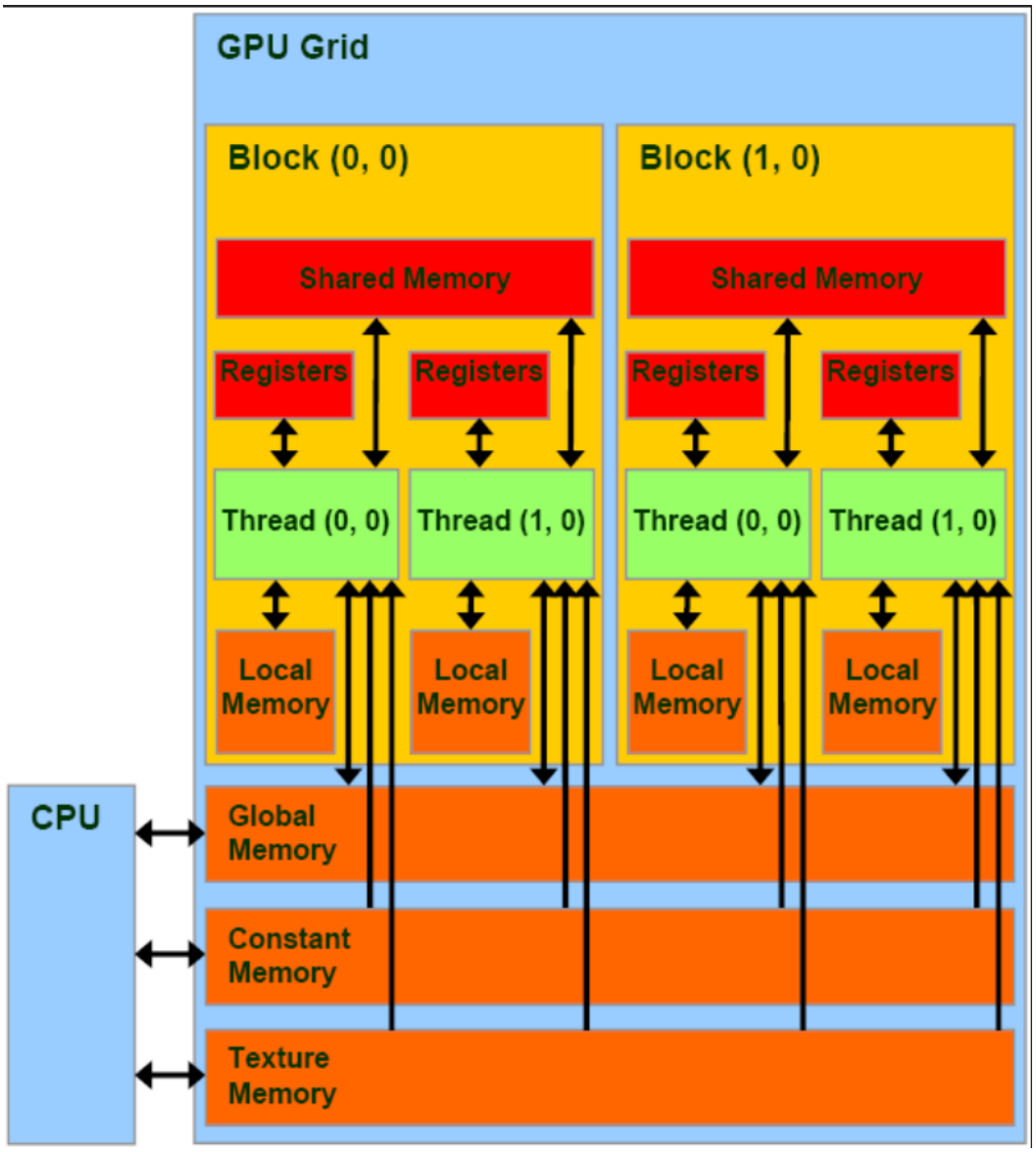


Figure 8.3: CUDA Memory Spaces

The memory spaces in CUDA are:

- **Global Memory** — It is the main memory of the GPU. It is accessible for read/write by all the threads in all the SMXs. It can also be accessed by host (CPU). For Kepler K40 it is around 12 GB. It has a latency of around 200–400 clock cycles (300 ns).
- **Constant Memory** — It is read-only memory. It is used to store constants that are used by all the threads in all the blocks.
- **Texture Memory** — It can be accessed by all threads. It is used to store textures, used for texture mapping. It is used to improve performance of reads that exhibit spatial locality among the threads.
- **Shared Memory** — Each SMX has its own individual memory. It is the memory shared by all the threads in a block executing in an SMX. It is faster than global memory. Shared memory has a latency of 20–30 clock cycles (5 ns). The threads can read/write to this memory.
- **Local Memory** — It is the memory local to each thread, used to store temporary variables. Each thread has read/write access to this memory.
- **Registers** — They are present in each SMX. It is the memory local to each thread. It is used to store the context of the thread. Kepler K40 has 64K registers in each SMX.

The host can read/write global, constant, and texture memory. GPUs prefer to access data from shared memory rather than device memory because of latency.

Difference between CPU and GPU Threads: There are a few differences between CPU and GPU threads regarding context switching. Context switching is much faster in GPUs because the state of a thread (or thread block) is stored in shared memory, and threads remain active until execution is complete.

Unlike in CPUs, where in the case of a context switch, memory related to that thread gets dumped to main memory or disk, in GPUs, this overhead is avoided. In CPUs, loading the data for the next thread requires retrieving it from main memory or disk, which is slow. Thus, context switching is faster in GPUs as there is no need to bring in any data—it is already loaded in shared memory.

Also, in the case of GPUs, the cache is explicitly managed by the programmer. The programmer has to explicitly bring frequently accessed data from device memory to shared memory. This differs from CPUs, where cache management is handled automatically by the hardware. For further details refer Nvidia [2020].

8.3 CUDA

A common way to program GPU architectures is by using the CUDA programming model. CUDA supports a hierarchical parallelism model, where computations in your program are structured as **grids**, which can be thought of as different phases of your program. These grids (or phases) execute one after another. Each grid consists of a number of **blocks**. This organization is specified by the programmer. That is, the programmer expresses computations in terms of grids and blocks.

The programmer launches a GPU **kernel** with a given number of blocks. A kernel corresponds to a grid. The blocks within a grid can be arranged in one, two, or

three dimensions. At runtime, the GPU scheduler assigns these blocks to **Streaming Multiprocessors (SMs)**. A block can be assigned to only one SM, but an SM can execute multiple blocks concurrently. For example, in many NVIDIA GPU architectures, a maximum of 16 thread blocks can be executed concurrently per SM.

Each block consists of **elements**, and each element is computed by a **thread**. In many architectures, the maximum number of threads per thread block is 1024. It is this thread — the **GPU thread** — that gets executed on a GPU core (also called an SP core). Similar to blocks, the elements within a block can be arranged in one-, two-, or three-dimensional layouts. This is illustrated in Figure 8.4.

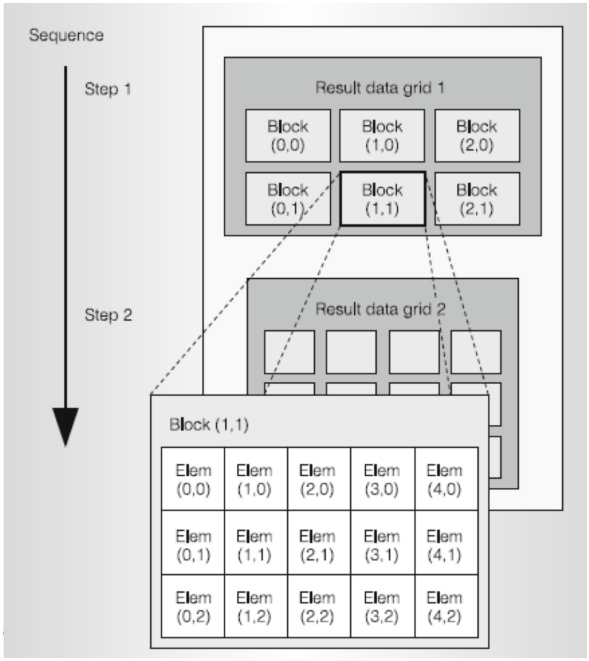


Figure 8.4: Decomposing result data into a grid of blocks partitioned into elements to be computed in parallel

In Figure 8.4, we see that the grids are two-dimensional and will execute sequentially. Inside a single grid, we see two-dimensional blocks, and each block, in turn, consists of multiple threads organized in two-dimensions.

8.3.1 Thread Blocks

A **thread block** is an array of concurrent threads that execute the same program (kernel) and cooperate to compute the result. Thread blocks can be arranged in one-, two-, or three-dimensional layouts (i.e., they have shape and dimensions).

Each thread within a thread block is uniquely identified by a `threadIdx` value, which may be one-, two-, or three-dimensional (e.g., `threadIdx.x`, `threadIdx.y`, `threadIdx.z`) depending on how the block is organized. These indices are essential for identifying and managing the work done by each thread within a block.

Importantly, all threads within a thread block execute on the same **Streaming Multiprocessor (SM)**, and therefore they can share fast, low-latency **shared memory**. This shared memory allows threads in the same block to cooperate and synchronize their execution efficiently.

8.3.2 Kernels

A **kernel** is a GPU function that is executed by a grid of thread blocks. In the CUDA programming model, a grid corresponds to a single invocation of a kernel. Just like the terms "element" and "thread" can often be used interchangeably in context, the terms "grid" and "thread blocks" are sometimes loosely interchanged when discussing GPU execution.

To compute a grid, we launch (or "call") the corresponding GPU kernel function. While launching, we specify various execution parameters: the number of blocks in the grid, how these blocks are arranged (in 1D, 2D, or 3D), and how each block itself is arranged in terms of threads or elements (also in 1D, 2D, or 3D). Once the kernel is launched with these parameters, the GPU scheduler handles the parallel execution of threads across the available SMs (Streaming Multiprocessors).

Each kernel call thus represents one phase of computation on the GPU, and multiple kernel calls can be used to structure a larger GPU program.

A **thread block** is a batch of threads that can cooperate with each other by:

- **Sharing data through shared memory.** Since each block is assigned to a single Streaming Multiprocessor (SM), all threads in a block reside within the same SM. SMs provide a shared memory space accessible to all threads of the block, which allows efficient data sharing and coordination.
- **Synchronizing their execution.** Threads in a block can use built-in synchronization mechanisms (such as `__syncthreads()`) to coordinate their execution, ensuring consistent results.

However, an important limitation is that threads from *different blocks* **cannot directly cooperate**. This is because thread blocks may be assigned to different SMs, and SMs do not share memory. As a result, there's no shared memory space available for inter-block communication.

> With advancements in GPU architecture, it is now possible for multiple kernels to be executed **simultaneously** on the GPU. By default, kernel launches are serialized, but modern GPUs support concurrent kernel execution if certain conditions are met and if the programmer uses the appropriate API (such as CUDA streams).

Execution of a CUDA program always begins on the CPU (host). From the host, we invoke GPU execution using specific CUDA kernel launch syntax. Before launching a kernel, we typically allocate and transfer data to the GPU's global (device) memory. Once the kernel completes its execution, data can be copied back from the device memory to the host.

While it is technically possible for threads from different blocks to communicate via **global memory** using locks or atomic operations, such communication is expensive and error-prone. Hence, it is best to design GPU programs so that **only threads within the same block** need to cooperate. This allows full use of the fast shared memory and efficient synchronization within an SM.

For an overview of CUDA memory spaces, refer to Figure 8.3. Each thread in CUDA has access to various memory spaces, with different scopes and performance characteristics:

- Read/write access to its own **per-thread registers**
- Read/write access to its own **per-thread local memory**
- Read/write access to **per-block shared memory**
- Read/write access to **per-grid global memory**
- **Read-only** access to **per-grid constant memory**
- **Read-only** access to **per-grid texture memory**

The host (CPU) can access the global, constant, and texture memories, all of which reside in device DRAM.

Given the high access latency of global memory (typically hundreds of cycles), it is crucial to minimize global memory accesses in performance-critical code. Instead, threads should make efficient use of **shared memory**—which resides within the SM and has significantly lower latency (around 10–20 cycles).

Global and shared memory are the most commonly used memory types in GPU computations. Other memory types—**local, constant, and texture memory**—are specialized for convenience or performance optimizations:

- **Local memory** is used by the compiler to store automatic array variables that cannot be held in registers.
- **Constant memory** is suitable for read-only data that is accessed uniformly by all threads. It is cached for fast access. See the CUDA Programming Guide Nvidia [2011] for more details.
- **Texture memory** is optimized for spatially coherent, read-only data with random access patterns. It is also cached and supports features like address clamping and wrapping.

The characteristics of these memory types are summarized in Table 8.1.

Memory	Location	Cached	Access	Scope ("Who?")
Local	Off-chip	No	Read/Write	One thread
Shared	On-chip	N/A	Read/Write	All threads in a block
Global	Off-chip	No	Read/Write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read	All threads + host

Table 8.1: CUDA Memory Spaces

CUDA is a programming language designed for expressing threaded parallelism on GPUs. It is a minimal extension of the C programming language, meaning most CUDA

programs consist of standard C code augmented with GPU-specific constructs—similar to what we saw with OpenMP and MPI.

A CUDA program typically begins execution on the CPU (host). At appropriate points, it launches GPU kernels—parallel functions that execute on the GPU (device). The main application runs on the CPU, while the compute-intensive kernels run across many parallel threads on the GPU. The programmer organizes these threads into a hierarchy of **thread blocks** and **grids**.

The parallel portions of the application are executed on the device as **kernels**. One or more kernels can be executed at a time, with many threads executing each kernel.

One of the key differences between CUDA threads and CPU threads is their lightweight nature. CUDA threads incur very little overhead for creation and context switching because register and shared memory data are persistent across switches. They do not get swapped out in the conventional sense, which makes switching between threads nearly instantaneous.

CUDA achieves efficiency by launching thousands of lightweight threads, whereas even multicore CPUs typically manage only a few heavyweight threads.

Note: From this point onward, we will refer to the GPU as the *device* and the CPU as the *host*. Kernels are special functions written to execute on the device.

8.4 CUDA C

We will now explore the fundamental CUDA C constructs and built-in variables that are essential for writing GPU kernels.

- `threadIdx.{x, y, z}` — Threads are arranged in one-, two-, or three-dimensional configurations within a block. To uniquely identify a thread within a block, we use the built-in variable `threadIdx` along with the appropriate dimension index.
- `blockIdx.{x, y, z}` — Similarly, to identify a thread block within a grid, we use `blockIdx` along with its dimensional coordinate(s). Each block in the grid has a unique ID.
- `blockDim.{x, y, z}` — This gives the dimension layout of the threads within a block. For example, if each block has 16×16 threads, then `blockDim.x = 16` and `blockDim.y = 16`.
- `gridDim.{x, y, z}` — This specifies how blocks are arranged within a grid. For example, if the grid has 8×8 blocks, then `gridDim.x = 8` and `gridDim.y = 8`.
- `kernel<<<nBlocks, nThreads>>>(args)` — This is the syntax to invoke a GPU kernel function. Unlike calling a CPU function, where one simply passes arguments to a function name, invoking a kernel requires special triple angle-bracket syntax `<<<...>>>` to indicate the number of thread blocks and number of threads per block. This lets the compiler know that a GPU kernel is being launched.

8.4.1 Example: Summing Up

Suppose we want to add the elements of two matrices A and B and store the result in matrix C. Assume that the matrices are of size $N \times N$ and stored as single-dimensional arrays in row-major order. The CPU version of the code looks like:

```
void addMatrix(float *a, float *b, float *c, int N)
{
    int i, j, idx;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            idx = j + i * N;
            c[idx] = a[idx] + b[idx];
        }
    }
}

void main()
{
    ...
    addMatrix(a, b, c, N);
}
```

The corresponding GPU implementation using CUDA would look like:

```
__global__ void addMatrixG(float *a, float *b, float *c, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int idx = i + j * N;

    if (i < N && j < N)
        c[idx] = a[idx] + b[idx];
}

void main()
{
    ...
    dim3 dimBlock(blocksize, blocksize);
    dim3 dimGrid(N / dimBlock.x, N / dimBlock.y);

    addMatrixG<<<dimGrid, dimBlock>>>(a, b, c, N);
    ...
}
```

In the `main` program, we define a `blocksize`, which specifies the number of threads per dimension in each thread block. Using this, we determine the number of blocks per grid dimension and define the `dimGrid` and `dimBlock` variables accordingly.

The kernel function `addMatrixG` is then launched using the special CUDA syntax `<<<dimGrid, dimBlock>>>`. Each thread computes a unique global index (i, j) using its `blockIdx`, `blockDim`, and `threadIdx` values. It then computes its corresponding linear index `idx` and performs the addition operation on that single element.

This is a form of SIMD (Single Instruction, Multiple Data) parallelism. In the GPU context, we often refer to this model as SIMT (Single Instruction, Multiple Threads) parallelism, where each thread executes the same instruction sequence but operates on different data elements.

8.4.2 Variable Qualifiers (GPU code)

The CUDA programming model provides several qualifiers to specify where variables reside in GPU memory. These qualifiers help in controlling data access scope, lifetime,

and performance:

- `__device__` – Declares a variable stored in global (device) memory. This memory is large but has high latency and no cache. Such variables are typically allocated using `cudaMalloc()` (the `__device__` qualifier is implied). These variables are accessible by all threads across all blocks. Their lifetime is the duration of the application.
- `__constant__` – Declares a variable in constant memory. It behaves like `__device__` memory but is cached and read-only for GPU threads. It must be initialized from the host using `cudaMemcpyToSymbol(...)`. The lifetime is also the duration of the application.
- `__shared__` – Declares a variable stored in shared memory, which resides on-chip and has very low latency. It is accessible by all threads in the same thread block. The lifetime is the duration of a kernel launch.

Variables that are scalars or built-in vector types without any qualifier are typically stored in registers. Arrays of more than 4 elements that are not explicitly shared or constant are stored in global device memory.

Typical CUDA Execution Steps

A standard CUDA program follows these steps:

1. Allocate memory on the device using `cudaMalloc()`.
2. Copy input data from the host (CPU) to the device (GPU) using `cudaMemcpy()`.
3. Launch the kernel on the GPU.
4. Optionally, perform additional computation on the host while the GPU kernel runs (asynchronous execution).
5. Copy the result back from the device to the host using `cudaMemcpy()`.
6. Free the device memory using `cudaFree()`.

By default, the CPU does not wait for the GPU kernel to finish execution. This allows for hybrid CPU–GPU computing, where while the GPU is executing, the CPU can simultaneously perform communication or computation.

Best Practices

To make effective use of CUDA:

- **Minimize** data transfer between CPU and GPU, as these transfers are expensive.
- **Maximize** parallelism by launching a large number of threads on the GPU.

Important CUDA Constructs

- `cudaMalloc(void **devPtr, size_t size)` – Allocates memory on the device.
- `cudaMemcpy(void *dst, const void *src, size_t count, cudaMemcpyKind kind)` – Copies memory between host and device.
- `cudaFree(void *devPtr)` – Frees the allocated memory on the device.
- `__syncthreads()` – Synchronizes all threads within a block (acts like a barrier).

8.5 Example: Matrix-Vector Multiplication

Consider the following CUDA code for computing the matrix-vector multiplication $Ax = y$:

```
__global__ void matvec_mul(int m, int n, double *A, double *x,
    double *y)
{
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    double sum = 0;

    if (row < m) {
        for (int col = 0; col < n; col++) {
            sum += A[row * n + col] * x[col];
        }
        y[row] = sum;
    }
}
```

Here, each entry of the output vector y can be computed independently. So, each thread is assigned a unique row of the matrix A using its global thread ID (computed from `blockIdx`, `blockDim`, and `threadIdx`). It performs a dot product of that row with the input vector x and stores the result in y.

The corresponding host code is as follows:

```
int main(int argc, char* argv[]) {
    int m = ...; // number of rows
    int n = ...; // number of columns

    size_t size_A = sizeof(double) * m * n;
    size_t size_x = sizeof(double) * n;
    size_t size_y = sizeof(double) * m;

    double *A = (double*) malloc(size_A);
    double *x = (double*) malloc(size_x);
    double *y = (double*) malloc(size_y);

    double *dA, *dx, *dy;

    // Allocate memory on the device
    cudaMalloc((void**) &dA, size_A);
    cudaMalloc((void**) &dx, size_x);
    cudaMalloc((void**) &dy, size_y);

    // Initialize A and x
```

```
    // ...

    // Initialize y to 0
    for (int i = 0; i < m; i++) y[i] = 0;

    // Copy data to the device
    cudaMemcpy(dA, A, size_A, cudaMemcpyHostToDevice);
    cudaMemcpy(dx, x, size_x, cudaMemcpyHostToDevice);

    // Launch the kernel
    int numThreadsPerBlock = 1024;
    int numBlocks = (m + numThreadsPerBlock - 1) /
        numThreadsPerBlock;

    dim3 dimGrid(numBlocks);
    dim3 dimBlock(numThreadsPerBlock);

    matvec_mul<<<dimGrid, dimBlock>>>(m, n, dA, dx, dy);

    // Copy the result back to host
    cudaMemcpy(y, dy, size_y, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(dA);
    cudaFree(dx);
    cudaFree(dy);

    // Free host memory
    free(A);
    free(x);
    free(y);

    return 0;
}
```

Note: In CUDA, you can only pass and manipulate one-dimensional arrays in device memory. Multidimensional arrays must be flattened manually. This is why in the kernel we access elements of A using `A[row * n + col]`.

Recall that after calling a GPU kernel, control immediately returns to the host CPU, and the subsequent lines of code execute without waiting for the GPU kernel to finish. However, if a `cudaMemcpy` function or another kernel that depends on the results of the previous kernel is called, then the GPU ensures that the earlier kernel completes before proceeding.

Thus, a dependency is implicitly created between the kernel call and `cudaMemcpy`, since we are copying the result vector y from device memory (which was computed by the kernel) back to the host memory. The `cudaMemcpy` call will block the CPU until the GPU kernel finishes execution and the data is available for transfer. Note that each thread in the kernel operates on a particular element of the output vector y.

8.5.1 Example 1, Version 2: Access from Shared Memory

Here, we present a modified version of the matrix-vector multiplication example that uses shared memory to improve cooperation among threads and reduce global memory access latency. Consider the following code:

```
__global__ void matvec_mul(int m, int n, double *A, double *x,
    double *y)
{
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    double sum = 0;

    // Declare shared memory for x
    __shared__ double sx[BLOCK_SIZE];

    // Each thread loads one element of x into shared memory
    if (threadIdx.x < n)
        sx[threadIdx.x] = x[threadIdx.x];

    __syncthreads(); // Ensure all threads have loaded their x values

    if (row < m) {
        for (int col = 0; col < n; col++) {
            sum += A[row * n + col] * sx[col];
        }
        y[row] = sum;
    }
}
```

This version assumes that the number of columns n is less than or equal to the block size (typically ≤ 1024 for many NVIDIA architectures). This is because each thread in the block is responsible for loading one element of x into shared memory.

This optimization improves performance because all threads now access the vector x from shared memory (`sx`) instead of repeatedly accessing the global memory. Shared memory resides on-chip and has much lower latency compared to global memory.

Note that in this version, the host (CPU) code remains the same as in the previous example.

8.6 Example 2: Matrix-Matrix Multiplication

Suppose we want to compute the product $C = A \cdot B$, where:

- A is of size (h_A, w_A)
- B is of size (w_A, w_B)
- Hence, C is of size (h_A, w_B)

The idea is to decompose matrix C into blocks, and assign each block to a thread block in two dimensions. Each thread block will:

1. Load a sub-matrix of A and a sub-matrix of B into shared memory.
2. Compute a partial result for the corresponding block of C.
3. Iterate this process across the width of A (or height of B), updating its partial result.

Each thread in a thread block computes one element of a sub-matrix of C. The element is computed as a dot product between a row of A and a column of B. This is as shown in figure 8.5.

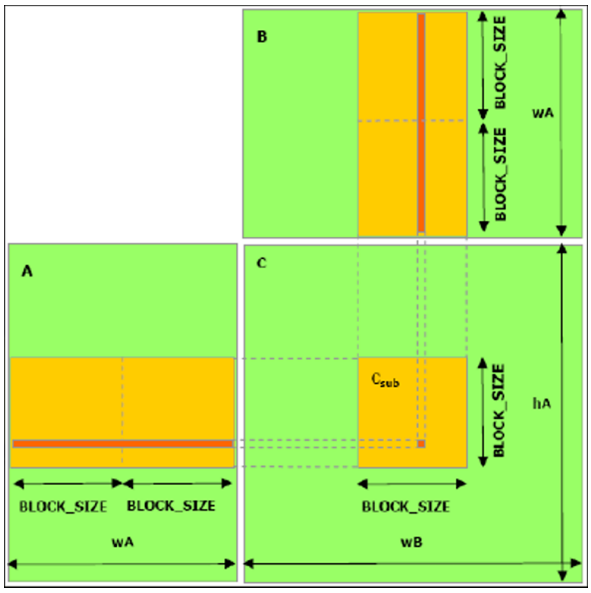


Figure 8.5: Matrix multiplication

Host Code

```
void Mul(const float *A, const float *B, int hA, int wA, int wB,
    float *C)
{
    float *Ad, *Bd, *Cd;
    size_t size;

    // Allocate and copy A
    size = hA * wA * sizeof(float);
    cudaMalloc((void**) &Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);

    // Allocate and copy B
    size = wA * wB * sizeof(float);
    cudaMalloc((void**) &Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

    // Allocate C
    size = hA * wB * sizeof(float);
    cudaMalloc((void**) &Cd, size);

    // Set up execution configuration
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);

    // Launch kernel
    Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);

    // Copy result back to host
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(Ad); cudaFree(Bd); cudaFree(Cd);
}
```

Kernel Code

```
__global__ void Muld(float *A, float *B, int wA, int wB, float *C)
{
    //Setup aBegin, aend, aStep, bBegin, bStep based on Block index
       and Block size

    // The element of the block sub-matrix that is computed by the
       thread
    float Csub = 0;
    // Loop over all the sub-matrices of A and B required to compute
       the block sub-matrix
    for(int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b +=
       bStep){

        // Shared memory for the sub-matrices of A and B
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SZIE][BLCOK_SIZE];

        // Load the matrices form gloabl memory to shared memory;
           each thread loads on element of each matrix
        As[ty][tx] = A[a+wA *ty +tx];
        Bs[ty][tx] = B[b+wB*ty+tx];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();

        // Multiply the two sub-matrices
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += As[ty][k] * Bs[k][tx];

        __syncthreads();
    }

    //Write the block sub-matrix to global memory; each thread
       writes own element
    int c = wB * BLOCK_SIZE *by+BLCOK_SIZE *bx;
    C[c+wB *ty+tx] = Csub;
}
```

Note:

- We assume that matrix dimensions are divisible by `BLOCK_SIZE`.
- Each thread computes one element of the output matrix C.
- Shared memory is used to load tiles (blocks) of A and B, reducing global memory traffic.

8.7 Example 3: Reduction

We now study a binary tree-based reduction algorithm. Given an array, we wish to compute a scalar value — such as the sum, minimum, or maximum of all its elements.

Basic Idea

The reduction process involves iteratively combining pairs of values in the array using a binary tree pattern. At each level:

- $\frac{N}{2}$ threads add adjacent pairs of values.
- In the next pass, $\frac{N}{4}$ threads operate on results from the previous pass.
- This continues until a single thread computes the final result.

At each level, threads must synchronize to ensure all intermediate computations are complete before the next level begins. This requires explicit synchronization using `__syncthreads()`.

Case 1: Fits in One Block

If the number of elements N is less than or equal to the number of threads allowed in a single block (typically 1024), then the entire reduction can be carried out within a single thread block. In this case:

- Synchronization across threads can be done using `__syncthreads()`.
- Shared memory can be used for fast communication among threads.
- This is an efficient and straightforward implementation.

Case 2: Too Large for One Block

If N is larger than the number of threads per block, multiple thread blocks are launched. Each block performs a partial reduction and writes its result to global memory. However, the final result is not yet available — further reduction on these partial results is needed.

Challenge: Synchronizing Across Blocks

CUDA does not allow synchronization across thread blocks within a single kernel launch, since blocks are scheduled independently and may run concurrently on different SMs. Thus, `__syncthreads()` only synchronizes threads within a block.

To solve this:

- One option is to use **atomic operations** and global flags to track completion. However, this is expensive and error-prone.
- A better solution is to use **multiple kernel launches**. Since CUDA guarantees that kernels are executed sequentially by default, each kernel launch acts as a global synchronization point.

Multiple Kernel Strategy

1. Launch a kernel where each thread block reduces a portion of the array and stores one partial result in global memory.
2. Launch a second kernel to reduce these partial results.
3. Repeat until the number of results is small enough to be reduced in a single thread block.

> In this multi-kernel strategy, all data remains on the GPU. There is no need to copy data back and forth between the host and device across kernel calls. The kernels read/write from the global memory allocated on the device, improving performance.

Global Synchronization via Kernels

This approach works because:

- CUDA guarantees that kernel calls are **blocking** on the host — the second kernel starts only after the first completes.
- This provides a convenient global synchronization mechanism between thread blocks.

Hence, the reduction is carried out level by level (as in a binary tree), with each level corresponding to a separate kernel invocation. This process continues until a final result is computed by a single block using intra-block reduction and synchronization.

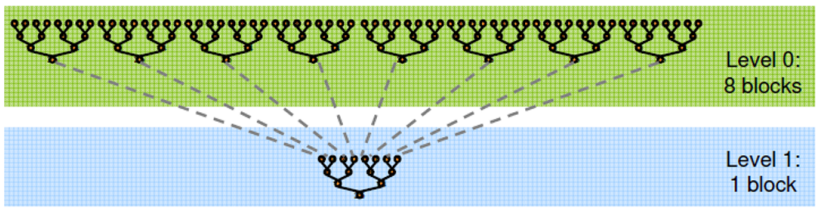


Figure 8.6: Reduction using multiple kernel launches for global synchronization

Let us now look at the host code for performing the reduction using multiple kernel calls:

```
int* h_idata, h_odata; /* host data*/
int* d_data, d_odata; /* device data*/

/* copying inputs to device memeory*/
cudaMemcpy(d_idata, h_idata, bytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_odata, h_idata, numBlocks*sizeof(int),
    cudaMemcpyHostToDevice);

int numThreadsperBlock = (n,maxthreadsperBlock) ?
   n:maxthreadsperBlock;
int numBlocks = n/numThreadsperBlock;
```

```
dim 3 dimBlock(numThreads, 1, 1); dim3 dimGrid(numBlocks, 1, 1);
reduce<<<dimGrid, dimBlock>>>(_data, d_odata);
int s=numBlocks;
while(s>1){
    numThreadsperBlock = (s<maxThreadsperBlock)?s:maxThreadsperBlock;
    numBlocks = s/numThreadsperBlock;
    dimBlock(numThreads, 1, 1); dimgrid(numBlocks, 1, 1);
    educe<<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata);
    s =s/numthreadsperBlock;
}
```

We will now look at the GPU kernel code used for binary tree-based reduction:

```
__gloabl__ void reduce(int*g_idata, int*g_odata)
{
    extern __shared__ int sdata[];
    //load shared mem
    unsigned int tid = threadidx.x;
    unsigned int i = blockIdx.x * blockDimx.x + threadiIdx.x;
    sdata[tid]=g_idata[i];
    __synchthreads();

    // do reduction in shared mem
    for (unsigned int s = 1; s < blockDimx.x; s *= 2){
    if (tid%(2*s)==0)
        sdata[tid]+=sdata[tid+s];
        __synchthreads();
    }
    // write result for this block to global mem
    if (tid==0) g_odata[blockIdx.x]=sdata[0];
}
```

Note that in the device code, we have utilized shared memory to perform the intermediate computations during the reduction process. We do not copy data back to global memory until we are certain that the reduction is complete. This strategy avoids frequent transfers between device and host memory, which are typically time-consuming and should be minimized for efficiency.

At each step of the reduction, data is copied from global memory into shared memory. Shared memory offers much lower latency compared to global memory and allows for significantly faster computation due to its on-chip nature. After computing the partial results, the outcome from shared memory is written back to global memory.

Furthermore, after each kernel invocation, the output array is copied into the input array to serve as the input for the next reduction iteration. This process continues until a single value remains, representing the final reduction result.

8.7.1 Example 4: Prefix Sum (Scan)

We now present a parallel algorithm for prefix sum (also called *scan*) computations on arrays using a divide-and-conquer strategy.

Given an input array X of size n, the goal is to compute the prefix sum array S such that:

$$S[i] = \sum_{j=0}^{i} X[j], \quad \text{for } 0 \leq i < n.$$

Step 1: Partitioning The array X is partitioned into P subarrays, one for each processor (assuming n is divisible by P for simplicity). Each processor P_i is assigned a subarray X_i of size n/P.

Step 2: Local Prefix Computation Each processor computes the prefix sum of its local subarray X_i independently and in parallel. This produces local prefix sum arrays:

$$S_i[j] = \sum_{k=0}^{j} X_i[k], \quad \text{for } 0 \leq j < n/P.$$

Step 3: Compute Block Sums Each processor writes the final value of its local prefix array (i.e., the sum of its block) to a separate auxiliary array Y of size P, where:

$$Y[i] = \sum_{j=0}^{n/P-1} X_i[j].$$

Step 4: Global Prefix of Block Sums Using a sequential or parallel prefix sum algorithm (e.g., binary tree), the prefix sum of the array Y is computed to get array Z, where:

$$Z[i] = \sum_{k=0}^{i-1} Y[k], \quad \text{with } Z[0] = 0.$$

This array Z stores the cumulative sum of all blocks that precede block i.

Step 5: Final Adjustment Each processor P_i adds $Z[i]$ to every element of its local prefix array S_i:

$$S_i[j] \leftarrow S_i[j] + Z[i], \quad \text{for all } j.$$

Result After this step, the local arrays S_i together represent the final global prefix sum array S.

Time Complexity

- Local prefix sums: $O(n/P)$ time (done in parallel).
- Global prefix sum of Y: $O(\log P)$ time using a binary tree algorithm.
- Adjustment step: $O(n/P)$ time.

Hence, the total parallel time is:

$$T(n, P) = O\left(\frac{n}{P} + \log P\right).$$

This is efficient and scalable for large n and moderate P.

8.8 Prefix Sum (Scan) using CUDA

We now present an efficient algorithm for computing the **prefix sum (scan)** in parallel using CUDA. This is a fundamental building block for many parallel algorithms such as stream compaction, sorting, histogram generation, etc.

Definition: The all-prefix-sums operation takes a binary associative operator $\oplus$ and an input array of n elements:

$$[a_0, a_1, a_2, \ldots, a_{n-1}]$$

and returns the array:

$$[a_0, (a_0 \oplus a_1), (a_0 \oplus a_1 \oplus a_2), \ldots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1})]$$

We assume $\oplus$ to be an associative binary operation such as addition, multiplication, or bitwise OR, AND, etc.

Approach: We solve this using a binary tree method with two main phases:

1. **Up-Sweep (Reduce) Phase:** A bottom-up traversal of the binary tree structure that computes partial sums at internal nodes.
2. **Down-Sweep Phase:** A top-down traversal that uses the partial results to compute the final prefix sums.

This algorithm has a parallel time complexity of $O(\log_2 n)$ using $O(n)$ work, assuming n is a power of two.

Up-Sweep Phase (Reduce)

In the up-sweep phase, we perform pairwise addition of elements with increasing stride. Each level in the tree reduces the number of active threads by half, while the distance between elements being combined doubles.

Initially, each thread performs:

$$X[i] = X[i] + X[i - d], \quad \text{where } d = 2^k \text{ in the } k\text{-th iteration}$$

The process continues until the final element contains the sum of the entire array. This phase stores partial results needed for the down-sweep.

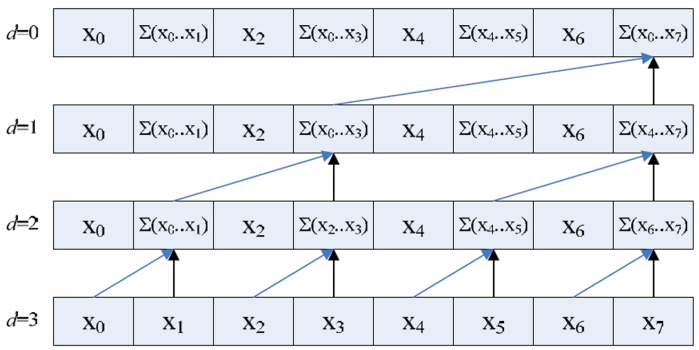


Figure 8.7: Up-Sweep Phase of Prefix Sum

Down-Sweep Phase

In the down-sweep phase, we traverse the binary tree from root to leaves. The idea is to propagate prefix values down the tree and replace each node with its prefix value, using the intermediate values from the up-sweep phase.

We begin by setting the last element to zero, as the identity element for addition:

$$X[n-1] \leftarrow 0$$

At each level, we swap the partial sum and update:

$$\text{temp} \leftarrow X[i-d]; \quad X[i-d] \leftarrow X[i]; \quad X[i] \leftarrow \text{temp} + X[i]$$

This step guarantees the prefix property across the array.

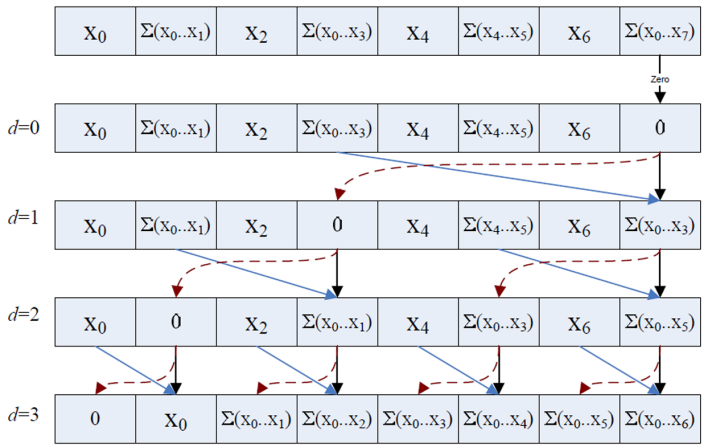


Figure 8.8: Down-Sweep Phase of Prefix Sum

CUDA Implementation Notes:

- The input array is typically stored in `shared memory` for fast access.
- `__syncthreads()` is used to synchronize threads at each level of both up-sweep and down-sweep phases.
- This algorithm performs an *exclusive scan* by default. For an inclusive scan, minor post-processing adjustments are made.
- Works best when the number of elements is a power of two. For non-power-of-two sizes, zero-padding is used.

Time Complexity: $O(\log_2 n)$ using n threads.

Work Complexity: $O(n)$, matching the sequential version, hence this method is cost-optimal.

Host Code

```
int main(){
    const unsigned int num_threads = num_elements/2;
    /* cudaMalloca d_idata and d_odata */
    cudaMemcpy( d_idata, h_data, mem_size, cudaMemcpyHostToDevice);

    dim3 grid(256, 1, 1); dim3 threads(num_threads, 1, 1);
    scan<<<grid, threads>>>(d_odata, d_idata);
    cudaMemcpy( h_data, d_odata[i], sizeof(float)*num_elements,
       cudaMemcpyDeviceToHost)
    /* cudaFree d_idata and d_odata */
}
```

Device Code

```
__global__ void scan_workefficient(float* g_odata, float* g_idata,
   int n)
{
    //Dynamically allocated shared memory for scan kernels
    extern __shared__ float temp[];
    int thid = threadIdx.x; int offset = 1;

    //Cache the computational window in shared memroy
    temp[2*thid] = g_idata[*thid];
    temp[2*thid +1] = g_idata[2*thid +1];
    // build the sum in place up the tree
    for (int d=n>>1; d>0; d>>=1)
    {
        __syncthreads();

        if(thid <d)
        {
            int ai = offset * (2*thid +1) - 1;
            int bi = offset * (2 * thid + 2) - 1;

            temp[bi] += temp[ai];
        }
        offset *= 2;
    }
    // scan back down the tree
    // clear the last element
    if (thid == 0) temp[n-1] = 0;
    // traverse down the tree building the scan in place
    for (int d = 1; d < n; d *= 2)
    {
        offset >>= 1;
        __syncthreads();
        if (thid < d)
        {
            int ai = offset*(2*thid + 1) - 1;
            int bi = offset*(2*thid + 2) - 1;
            float t = temp[ai];
            temp[ai] = temp[bi];
            temp[bi] += t;
        }
```

```
40      }
41
42      __syncthreads();
43
44      //write results to global memory
45      g_odata[2*thid] = temp[2*thid];
46      g_odata[2*thid +1] = temp[2*thid +1]
47  }
```

For implementation details and optimization strategies, refer to Nvidia [2011].

Chapter 9

Parallel Algorithms

9.1 Parallel Sorting

Let us first consider the problem of sorting in parallel.

Problem Definition

The input is a sequence of size N that is initially distributed across P processors. Each processor holds a local subset of the global sequence.

The objective is to sort the global sequence such that:

- The elements in each processor P_i are sorted locally.
- All elements in processor P_i are greater than all elements in processor P_{i-1} and less than all elements in processor P_{i+1}, for all valid i.

9.1.1 Parallel Quick Sort

One naive approach to parallel sorting is a parallel variant of the quicksort algorithm.

Algorithm Overview

1. Initially, a single processor holds the entire dataset. It chooses a pivot element and partitions the array into two sub-arrays:
   - Elements less than the pivot.
   - Elements greater than the pivot.
2. These two sub-arrays are then passed to two processors (including the original one), and each processor recursively performs local quicksort on its assigned sub-array.
3. This process continues, doubling the number of sub-arrays (and processors involved) at each level, until we reach P sub-arrays—one per processor.

Disadvantages

1. **Inefficient Utilization of Processors:**
   - In the early stages—where the most computation is performed—only a few processors are active. For example, only one processor is used for the full array in the first step, two in the second, and so on.
   - Consequently, most processors remain idle during the compute-heavy phases, resulting in poor load balancing and underutilization of parallel hardware.
2. **Single Processor Memory Bottleneck:**
   - The algorithm requires one processor to initially hold the entire dataset, which may not be feasible for large problem sizes.
   - This violates the definition of a distributed input, and limits the scalability of the algorithm on distributed-memory systems.

9.1.2 Parallel Quicksort with Full Processor Utilization

This algorithm improves over the naive parallel quicksort by involving **all processors at every iteration**. It avoids the underutilization of processors seen earlier.

We assume the input array is **evenly divided** across all P processors, each holding N/P elements.

Overview of the Algorithm

1. Processor P_0 selects a pivot element and **broadcasts** it to all other processors.
2. Each processor P_i partitions its local array into:
   - L_i: Elements **less than** the pivot.
   - G_i: Elements **greater than or equal to** the pivot.
3. Let $L = \sum_i |L_i|$ and $G = \sum_i |G_i|$. The processors are now **split into two subgroups**:
   - The first $\frac{P \cdot L}{N}$ processors handle all L_i.
   - The remaining $\frac{P \cdot G}{N}$ processors handle all G_i.
4. The computation continues recursively within each subgroup. Each subgroup selects a new pivot, partitions its elements again into L_i and G_i, and the processors are recursively split again.
5. This process continues until there are exactly P subgroups, each of size 1. At that point, each processor performs a **local quicksort** on its portion of the array.

Shared Memory Implementation

In a shared memory system:

- The L_i subarrays are placed in the **beginning portion** of a shared memory array, and the G_i subarrays are placed in the **remaining portion**.
- Each processor must know where to write its L_i and G_i elements in the shared memory.

To compute the starting write positions efficiently, we use the **prefix sum** operation on the sizes of the L_i and G_i subarrays. The prefix sum provides:

- The total number of L elements before processor P_i, giving it an exact memory offset.
- A similar offset for G_i within the second half of the shared memory.

This avoids the need for locks or atomics, and allows processors to write concurrently without conflict. Prefix sums can be computed in $O(\log P)$ time and are highly efficient on parallel systems.

Recursive Step: In the next step, each subgroup (those handling L_i and G_i respectively) selects a new pivot, repeats the partitioning, and continues recursively as described above. When the number of subgroups reaches P, each processor holds its final partition and performs a local sort.

Final Output: This approach guarantees that each processor P_i contains a sorted subarray, and:

$$\text{All elements in } P_{i-1} < \text{All elements in } P_i < \text{All elements in } P_{i+1}$$

achieving the final goal of global sorting in a distributed manner.

Message Passing Implementation

In a distributed memory setting (e.g., using MPI):

- Each processor must know **where to send** its L_i and G_i elements.
- This requires a **distributed prefix sum** to determine:
  - Global indices in the L or G partition.
  - The destination processor based on cumulative counts and group sizes.
- For instance, the destination processor for a chunk of L_i is determined by:

$$\text{Processor index} = \left\lfloor \frac{\text{prefix sum of } L_i \text{ elements}}{\text{number of } L \text{ elements per processor}} \right\rfloor$$

- The same applies for G_i.

This communication step can involve **all-to-all communication**, and in the worst case takes $O(N/P)$ time. The process is then repeated recursively on subgroups as before.

Final Local Sort

Once each processor is assigned its final portion of the array, it performs a local sort. The complexity is:

$$O\left(\frac{N}{P}\log_2\frac{N}{P}\right)$$

This variant of quicksort—called **Parallel Quicksort**—uses all processors efficiently and achieves good load balancing throughout the sorting process.

9.1.3 Complexity Analysis

The total time complexity of the parallel quicksort algorithm can be broken down into several components:

- The number of partitioning steps (recursive levels) is $O(\log_2 P)$, since at each step, the processor groups are divided roughly in half until we reach P groups.
- At each step, the following operations are performed:
  - **Broadcast of pivot:** Can be implemented using a binary tree in $O(\log_2 P)$ time.
  - **Allreduce (to count L and G elements):** Also costs $O(\log_2 P)$ time.
  - **Prefix sum:** Needed to compute offsets for data redistribution. Costs $O(\log_2 P)$.
  - **All-to-all communication:** Used to redistribute L_i and G_i values among processors. This requires $O(N/P)$ time in the worst case.

  Therefore, the cost per partitioning step is:

  $$O(\log_2 P + N/P)$$

  And across $O(\log_2 P)$ levels, the total cost becomes:

  $$O(\log_2^2 P + \frac{N}{P}\log_2 P)$$

- Once the partitioning completes, each processor performs a local sort on its portion. In the best case, each processor has roughly N/P elements, giving:

  $$O\left(\frac{N}{P}\log_2\frac{N}{P}\right)$$

Total Complexity:

$$O\left(\frac{N}{P}\log_2\frac{N}{P}\right) + O\left(\log_2^2 P + \frac{N}{P}\log_2 P\right)$$

Drawbacks

The major drawback of parallel quicksort lies in the potential for **load imbalance** and **underutilization of processors**. Unlike merge sort or radix sort, quicksort depends heavily on the choice of pivot.

In the worst case, the pivot may divide the array very unevenly, leading to:

- Some processors receiving significantly more data than others.
- Unequal distribution of work during the final local sorting stage.

Thus, even though the algorithm scales well theoretically, its practical performance can suffer due to this inherent imbalance. There is no guaranteed strategy for choosing a pivot that results in equal-sized partitions every time. Therefore, quicksort, although fast sequentially, is often considered **not ideal for parallel sorting** due to its unpredictable behavior with respect to load balancing.

For more details on parallel quicksort refer Kumar et al. [1994].

9.2 Breadth-First Search (BFS)

Graph traversal algorithms are critical in analyzing large datasets where relationships among data objects are represented as graphs. Breadth-First Search (BFS) is a fundamental algorithm used to explore graphs level by level and is widely used for applications such as finding shortest paths, social network analysis, web crawling, and more.

9.2.1 Level-Synchronized Algorithm

We begin with a level-synchronized parallel BFS algorithm. This approach proceeds level-by-level starting from a source vertex v_s. The **level** of a vertex v is defined as its graph distance from the source.

This is a **frontier-based** algorithm: in each iteration, the current *frontier* (the set of vertices at the current level) is expanded to discover the next set of vertices. The algorithm operates in the **Bulk Synchronous Parallelism (BSP)** model where threads synchronize after each level expansion.

Key design question: How should we decompose the graph (vertices, edges, adjacency matrix) among processors?

Distributed BFS with 1D Partitioning

In this approach, each processor is assigned a subset of the vertices and their outgoing edges. This is equivalent to performing a 1D row-wise partitioning of the graph's adjacency matrix.

Each vertex v has an edge list defined by the indices in row v of the adjacency matrix A. At each BFS level l, each processor owns a subset F of the *frontier vertices*.

Each processor:

- Traverses the edges of vertices in F.
- Forms a set N of neighboring vertices.

- Filters N to determine which of its members are owned by other processors.

- Communicates these vertices to their owner processors.

- The owners then include the received vertices in their own local frontiers.

For further details refer Yoo et al. [2005] (pages 1-7).

Algorithm 1 Distributed BFS with 1D Partitioning

Initialize level array $L_{v_s}(v) = \begin{cases} 0, & \text{if } v = v_s \text{ (source)} \\ \infty, & \text{otherwise} \end{cases}$
for $l = 0$ to ∞ **do**
  $F \leftarrow \{v \mid L_{v_s}(v) = l\}$ (Local frontier)
  if $F = \emptyset$ on all processors **then**
    break ▷ *Traversal complete*
  end if
  $N \leftarrow$ neighbors of vertices in F
  for each processor q **do**
    $N_q \leftarrow \{v \in N \mid \text{owned by } q\}$
    Send N_q to processor q
    Receive $\tilde{N}_q$ from processor q
  end for
  $\tilde{N} \leftarrow \bigcup_q \tilde{N}_q$ ▷ *Combined new frontier*
  for each $v \in \tilde{N}$ such that $L_{v_s}(v) = \infty$ **do**
    $L_{v_s}(v) \leftarrow l + 1$
  end for
end for

BFS on GPUs

A GPU-based BFS assigns a thread to each vertex. A kernel is launched at each level with one thread per vertex. Only threads corresponding to current frontier vertices become active.

This implementation uses global memory arrays for visited flags and level distances. It may result in load imbalance, since some threads may be idle while others are active, depending on the graph structure.

Note: Atomics (e.g., `atomicCAS`) may be required to avoid race conditions when multiple threads attempt to write to the same memory location.

Algorithm 2 GPU BFS Kernel (Frontier-based)

```
1: procedure BFS_KERNEL(currLevel)
2:     v ← blockIdx.x × blockDim.x + threadIdx.x
3:     if dist[v] == currLevel then
4:         for each n ∈ neighbors(v) do
5:             if visited[n] == 0 then
6:                 dist[n] ← dist[v] + 1
7:                 visited[n] ← 1
8:             end if
9:         end for
10:    end if
11: end procedure
```

Challenges with GPU BFS:

- **Thread divergence**: Threads assigned to non-frontier vertices will idle.
- **Load imbalance**: High-degree vertices cause some threads to do significantly more work.
- **Race conditions**: Need atomics for concurrent updates to shared arrays (e.g., `dist`, `visited`).

For further details refer Harish and Narayanan [2007] (pages 5-8). Several advanced techniques exist to mitigate these issues:

- Work-efficient BFS (e.g., queue-based).
- Direction-optimizing BFS (push vs pull model).
- Warp-centric and load-balanced kernel launches.

9.3 Gaussian Elimination

Recall that in gaussian elimination we are required to zero out the elements below the diagonal by adding multiples of row i to later rows. Let us look at the following sequential code for it:

```
    for i = 1 to n-1
        for j = i+1 to n
            for k = i to n
                A(j,k) = A(j,k) - A(j,i)/A(i,i) * A(i,k)
```

Note that we don't need to compute $A(j,i)/A(i,i)$ from inner loop every time. So we modify the code to do as follows:

```
for i = 1 to n-1
    for j = i+1 to n
        m = A(j,i)/A(i,i)
            for k = i to n
                A(j,k) = A(j,k) - m * A(i,k)
```

Note that in each iteration we know that $A(j,i) = 0$. Thus no need to compute what we already know. Hence we start the innermost loop with $i + 1$. Thus, we further modify the code as:

```
for i = 1 to n-1
    for j = i+1 to n
        m = A(j,i)/A(i,i)
            for k = i+1 to n
                A(j,k) = A(j,k) - m * A(i,k)
```

We further modify to store multiplies m below diagonals.

```
    for i = 1 to n-1
        for j = i+1 to n
            A(j,i) = A(j,i) / A(i,i)
                for k = i+1 to n
                    A(j,k) = A(j,k) - A(j,i) * A(i,k)
```

Let us now look at the runtime of this algorithm. Note that in each iteration in the innermost loop k goes from $i + 1$ to n where i goes from 1 to $n - 1$. Thus, the total number of times we compute:

$$\sum_{i=1}^{n-1}(n-i)^2 = O\left(\frac{n^3}{3}\right)$$

Let us now look at the Parallel Gaussian Elimination:

Chapter 10

PRAM Model

10.1 Introduction

PRAM stands for **Parallel Random Access Machine**. It is an abstract computational model designed to help researchers and developers formulate parallel algorithms without being constrained by the underlying hardware limitations. The model assumes an idealized parallel computer in which:

- There are an arbitrary number of processors.
- All processors operate synchronously in lockstep, i.e., they follow a global clock.
- All processors have uniform access to a shared memory in constant time.

This abstraction allows algorithm designers to focus purely on the parallelism inherent to the algorithm, avoiding details like message passing, memory latency, synchronization overhead, or communication costs. Thus, PRAM enables the study of algorithmic efficiency in an idealized setting, decoupled from real-world hardware complexity.

10.2 Benefits of PRAM

There are several significant benefits of using the PRAM model for designing and analyzing parallel algorithms:

1. **Architectural Independence:** PRAM ignores machine-specific characteristics such as network topology or memory hierarchies. This allows for the creation of portable high-level parallel algorithms that can later be adapted to a wide variety of parallel architectures.
2. **Upper and Lower Bounds:** The model provides a clean theoretical framework to establish asymptotic upper and lower bounds on time and work (i.e., total number of operations across all processors), which helps in understanding the inherent difficulty of problems and guides optimal parallelization strategies.

3. **Simplified Design Space:** By abstracting away low-level concerns, PRAM simplifies the process of algorithm design and analysis. Once a theoretically efficient PRAM algorithm is developed, it can be transformed (possibly with some compromises) into a practical algorithm suited for real hardware.

4. **Basis for Real Machines:** Many ideas from PRAM algorithms inspire actual parallel programming techniques. The PRAM model can serve as a high-level blueprint for designing efficient parallel programs on real systems, such as multi-core CPUs, shared memory systems, or GPUs.

5. **Inspiration for Architecture Design:** PRAM algorithms often reveal the theoretical maximum parallelism possible for a given problem. This can drive hardware designers to develop architectures (e.g., SIMD processors, GPU thread hierarchies) that attempt to approximate such ideal parallelism.

6. **Applicability to Modern Architectures:** Although no machine can truly realize the PRAM model due to communication and synchronization limitations, many modern architectures like GPUs can emulate aspects of PRAM using shared memory and thousands of lightweight threads. Hence, PRAM remains relevant and influential in the design of efficient GPU-based algorithms.

10.3 PRAM Architecture Model

The PRAM architecture consists of the following core components:

- **Control Unit:** Coordinates and synchronizes the actions of all processors. All processors follow the same instruction in lockstep, adhering to the SIMD (Single Instruction, Multiple Data) paradigm.
- **Global Shared Memory:** A single memory space accessible by all processors. Each memory cell can be accessed in constant time, and all processors share this memory space.
- **Unbounded Set of Processors:** The PRAM model assumes an unlimited number of processors, each having a small amount of private memory for intermediate computations.

Each active processor operates in the following cycle:

1. **Read** a value from global memory.
2. **Perform** a local computation using its private memory.
3. **Write** the result back to global memory.

10.3.1 Memory Access Models

The main distinction between different types of PRAM models lies in how they handle memory access conflicts, especially when multiple processors try to read from or write to the same memory location simultaneously.

- **EREW (Exclusive Read Exclusive Write):** No two processors can access the same memory cell for reading or writing in the same time step. This is the most restrictive and realistic model.
- **CREW (Concurrent Read Exclusive Write):** Multiple processors are allowed to read from the same memory location simultaneously, but only one processor can write to a memory location at any given time.
- **CRCW (Concurrent Read Concurrent Write):** Multiple processors can read from and write to the same memory location in the same time step. The model defines additional rules to resolve write conflicts:
  - **COMMON:** All processors attempting to write to a memory cell must write the same value.
  - **ARBITRARY:** An arbitrary processor among the contenders is chosen to perform the write.
  - **PRIORITY:** The processor with the lowest index (or priority) among those attempting the write wins and its value is written.

10.3.2 Model Equivalence

Although these models differ in power and ease of programming, they are computationally equivalent in the theoretical sense. That is, an algorithm designed for one PRAM model can be simulated on another model with at most a logarithmic time overhead.

For instance, a CRCW-PRIORITY algorithm can be simulated on an EREW-PRAM with an overhead of at most $O(\log P)$ time, where P is the number of processors. This allows algorithm designers to choose a simpler model for the design phase (e.g., CRCW) and later convert or adapt it for implementation in a more restrictive model (e.g., EREW or CREW).

10.4 Steps in PRAM Algorithm

PRAM algorithms typically follow two broad phases:

- **Phase 1: Processor Activation.** A sufficient number of processors are activated based on the size of the input and the nature of the task.
- **Phase 2: Parallel Computation.** The activated processors execute computations in parallel, according to the model's access rules (e.g., EREW, CREW, or CRCW).

10.4.1 Example: Binary Tree Reduction

Binary tree-based reduction is a classic example of a PRAM algorithm. The goal is to reduce an array of n elements using an associative operation (e.g., sum, min, max) into a single value.

In this algorithm, the input array is treated as the leaves of a binary tree. In each step, $n/2$ processors are used to pair and combine adjacent elements. The result is

stored in a smaller array of size $n/2$, and the process repeats on this reduced array. After $\log_2 n$ steps, the final result is obtained at the root.

This reduction can be efficiently implemented using the EREW PRAM model, as each processor reads from and writes to distinct memory locations in every step.

10.4.2 Example: Prefix Sum Calculation

Another important operation in parallel computing is the prefix sum (also called the scan operation). Given an array A of n elements, the prefix sum array B is defined as:

$$B[i] = \sum_{j=0}^{i} A[j], \quad \text{for } 0 \leq i < n.$$

Prefix sum is a building block for several applications including:

- Array partitioning based on conditions (e.g., separating odd/even numbers).
- Memory allocation.
- Lock-free synchronization in shared-memory parallel architectures.

We present a CREW PRAM algorithm for computing prefix sums in $O(\log_2 n)$ time using n processors.

CREW PRAM Algorithm

We assume an input array A of size n. The figure 10.1 illustrates the algorithm where, in each step, processors concurrently read two elements from the array (concurrent read) and write the sum to the corresponding location (exclusive write). This satisfies the CREW PRAM access constraints.

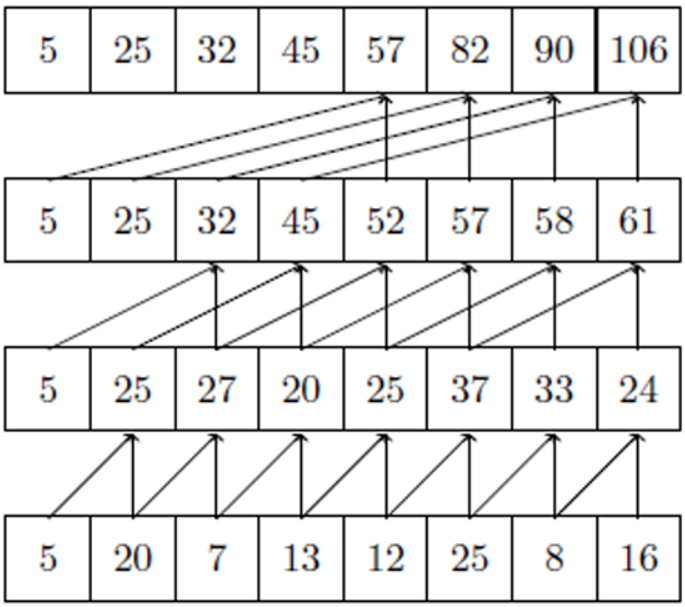


Figure 10.1: CREW PRAM-based Prefix Sum Computation

The key idea is that at each level of the computation tree, the distance between the elements being added doubles, and the same number of processors are used to compute. The total number of parallel steps required is $O(\log_2 n)$.

10.4.3 Example: Merging Sorted Lists

Most PRAM algorithms achieve reduced time complexity by leveraging parallelism at the cost of performing more total operations than an optimal RAM (sequential) algorithm. This trade-off is a common theme in the design of parallel algorithms.

Sequential Case (RAM Model). Consider merging two sorted lists A and B, each containing $n/2$ elements. A standard merge operation proceeds in linear time by maintaining two pointers, one for each list, and repeatedly selecting the smaller of the two current elements. This requires at most $n - 1$ comparisons, yielding a time complexity of $O(n)$, and is work-optimal in the RAM model.

Parallel Case (CREW PRAM Model). In contrast, we now describe a parallel merging algorithm under the CREW PRAM model using n processors — one assigned to each element of the two lists. The basic idea is as follows:

- Each processor P_i is assigned to element x_i from either list A or B (assume a merged virtual list of size n).
- The processor knows the index of x_i in its own list.
- To determine the final position of x_i in the merged output, the processor performs a binary search in the *other* list to find how many elements in that list are smaller than x_i.
- The sum of the two ranks (local index and the binary search result) gives the final position of x_i in the merged list.

Each binary search operation takes $O(\log_2 n)$ time, and since there are n processors performing independent binary searches, the total number of operations is $O(n \log_2 n)$.

Time Complexity.

$$T(n) = O(\log_2 n) \quad \text{(parallel time per processor)}$$

Work Complexity.

$$W(n) = O(n \log_2 n)$$

Cost Optimality. A parallel algorithm is said to be *cost-optimal* if the total work done by all processors matches the best known sequential time complexity. Here, the best sequential algorithm for merging takes $O(n)$ time, but our PRAM algorithm performs $O(n \log_2 n)$ work. Therefore, this algorithm is **not cost-optimal**.

Remarks. While the algorithm achieves polylogarithmic time performance, it does so by increasing the total computational work. This is a characteristic trade-off in many PRAM algorithms — achieving low parallel time at the cost of higher overall computation.

More advanced parallel merging algorithms exist that reduce the total work, for example, by using sampling and divide-and-conquer strategies. However, they typically involve more complex coordination among processors.

10.4.4 Example: Enumeration Sort

Enumeration Sort is a comparison-based sorting algorithm that determines the final position of each element in the sorted array by counting the number of elements that are smaller than it.

Sequential Version. In a traditional sequential implementation, for each element a_i, we count the number of elements a_j such that $a_j < a_i$. The count gives the rank or final position of a_i in the sorted array. This requires $O(n^2)$ comparisons in total, and thus the sequential complexity is $O(n^2)$.

Parallel Version: Special CRCW PRAM Model. We now consider a parallel implementation using a special CRCW (Concurrent Read, Concurrent Write) PRAM model. This version can compute the sorted order in constant time $O(1)$ using n^2 processors.

- Each processor $P_{i,j}$ is assigned to compare elements $a[i]$ and $a[j]$ for $i, j \in \{0, 1, \ldots, n-1\}$.
- If $a[i] > a[j]$, the processor writes a 1 to a shared memory location representing $position[i]$, otherwise it writes 0.
- We assume a special CRCW variant in which if multiple processors write to the same memory location simultaneously, the **sum** of all written values is stored (i.e., a summation write conflict resolution).
- Thus, $position[i]$ will store the number of elements less than $a[i]$, which corresponds to its rank in the sorted array.

This means that all n^2 comparisons can happen in parallel in one step, and all writes to the $position$ array are resolved via concurrent summation, yielding the correct sorted positions in constant time.

Time Complexity.

$$T(n) = O(1)$$

Work Complexity.

$$W(n) = O(n^2)$$

Cost Optimality. This algorithm is **not cost-optimal** because it performs $O(n^2)$ total operations, whereas the best-known comparison-based sequential sorting algorithms such as Merge Sort or Quick Sort perform $O(n \log_2 n)$ comparisons.

Summary. Enumeration Sort demonstrates the power of PRAM models in reducing parallel time complexity through massive processor usage. However, such algorithms may be impractical for real-world machines due to their unrealistic assumptions and high processor demands, and they are not cost-efficient compared to optimal sequential or practical parallel algorithms.

10.5 Summary

In summary, PRAM algorithms serve primarily as theoretical constructs that abstract away architectural constraints such as memory latency, bandwidth limitations, and synchronization overhead. Despite being idealized, they are immensely valuable for the following reasons:

- They provide a simple and clean model for designing and analyzing parallel algorithms.
- They allow us to establish upper and lower bounds on the parallel time complexity of problems.
- They guide the design of efficient parallel algorithms that can later be adapted for practical systems such as multicore CPUs or GPUs.
- They help in identifying parallelism in a problem, which is crucial in the early design phase.
- They can even inspire the architecture of specialized parallel machines or hardware accelerators.

While PRAM algorithms may not be directly implementable on real-world systems due to their ideal assumptions (e.g., unlimited processors and zero-cost synchronization), they remain a foundational tool for both education and research in parallel computing. For further details refer Quinn [1994] pages 25-32, 40-42, 256.

Bibliography

Randal E Bryant and David R O'Hallaron. Computer systems, 2016.

David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel computer architecture: a hardware/software approach*. Elsevier, 1998.

Pawan Harish and Petter J Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *International conference on high-performance computing*, pages 197–208. Springer, 2007.

Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing*, volume 110. Benjamin/Cummings Redwood City, CA, 1994.

Lawrence Livermore National Laboratory. Openmp tutorial, 2021. URL https://hpc.llnl.gov/tutorials/openMP. Accessed: 2025-07-27.

Lawrence Livermore National Laboratory. Introduction to parallel computing, 2023. URL https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing. Accessed: 2025-07-20.

NPTEL. Introduction, flynn's taxonomy, and memory-based classification, 2020. URL https://www.youtube.com/watch?v=1jYOBr-orpk&list=PLm6ShqSrKDxPEU6TDUUxE02M_sMc1mn0B. Lectures 1–4 from the NPTEL High Performance Computing Course.

CUDA Nvidia. Nvidia cuda c programming guide. *Nvidia Corporation*, 120(18):8, 2011.

NVIDIA Nvidia. A100 tensor core gpu architecture. *Unprecedented Acceleration at Every Scale*, 2020.

OpenMP Architecture Review Board. The official openmp website. https://www.openmp.org/, 2025. Accessed: 2025-07-27.

Michael J Quinn. *Parallel computing theory and practice*. McGraw-Hill, Inc., 1994.

V Rajaraman. Ieee standard for floating point numbers. *Resonance*, 21:11–30, 2016.

Marc Snir. *MPI–the Complete Reference: the MPI core*, volume 1. MIT press, 1998.

Intel Software. Openmp lectures - youtube playlist. https://m.youtube.com/watch?v=nE-xN4Bf8XI&pp=ygUVT3Blbk1QIGxlY3R1cmVzIGludGVs, 2014. Accessed: 2025-07-27.

Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 25–25. IEEE, 2005.