Notes on Parallel Computing

Nihar Shah

July 19, 2025

Contents

Chapter 1

Computer Architecture

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

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

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

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:

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

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: $000\dots0001110$ (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: $111\ldots1110010$

- **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: $1.011_2 = 1.375$

So the final number is:

$$-1 \times 1.375 \times 2^2 = -5.5$$

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	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

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 signifcand 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.

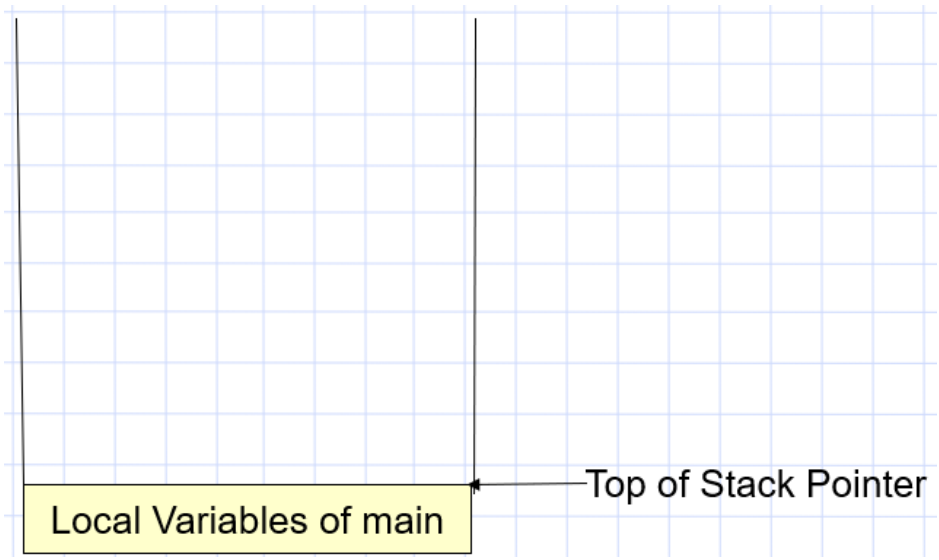


Figure 1.1: Memory Layout of a Program

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()`.

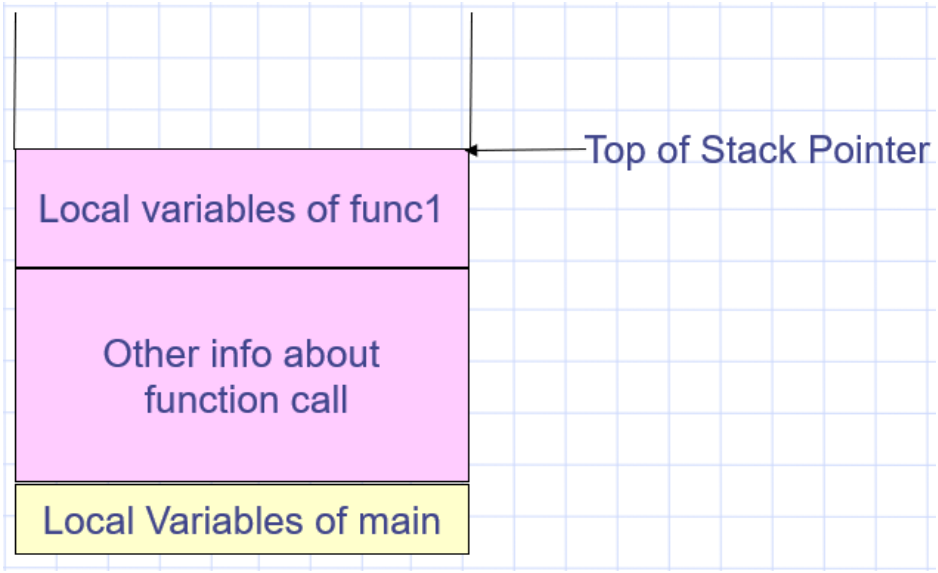


Figure 1.2: Memory Layout after a Function Call to `func1()`

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.

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.

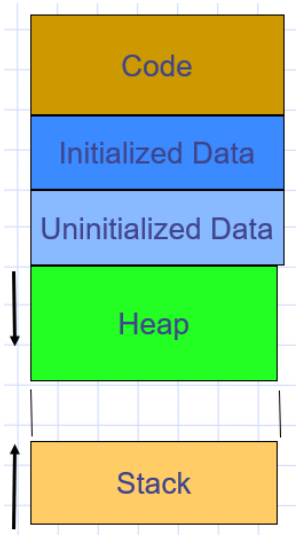


Figure 1.3: Program and Data Memory Layout

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 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.

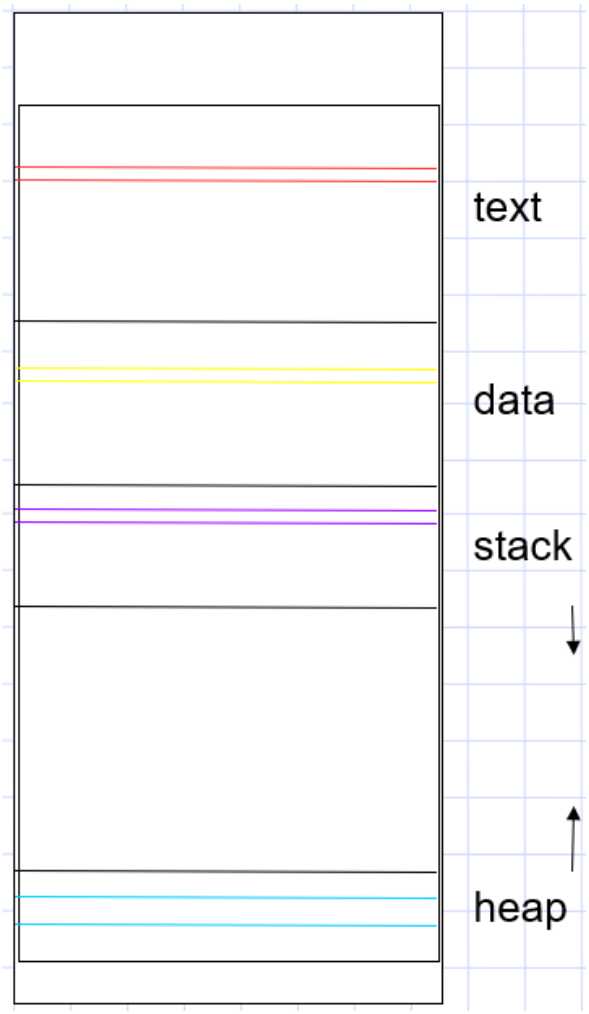


Figure 1.4: Program Memory Layout

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.

- **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.

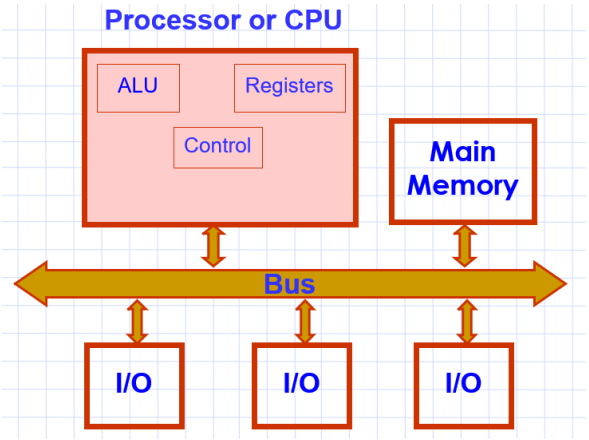


Figure 1.5: Basic Computer Organization

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 $100\times$ 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.

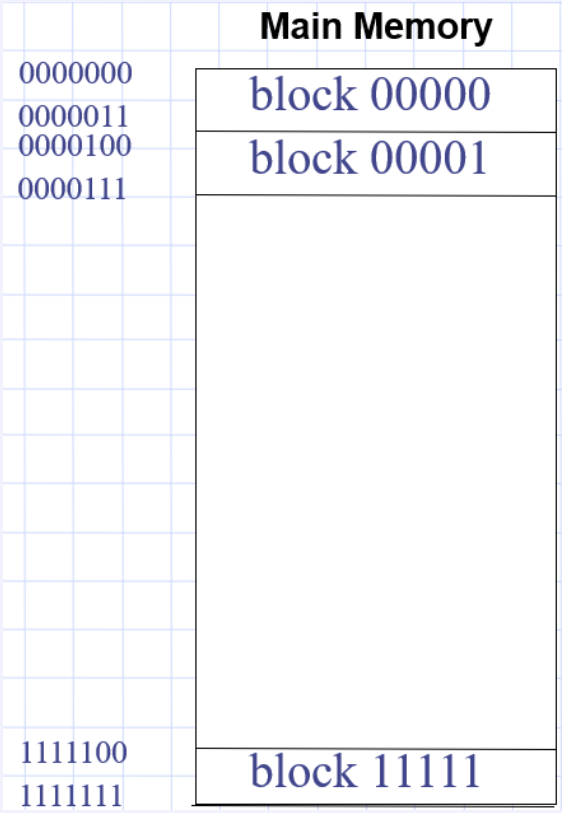


Figure 1.6: Memory Address Breakdown

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.

1.2.3 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

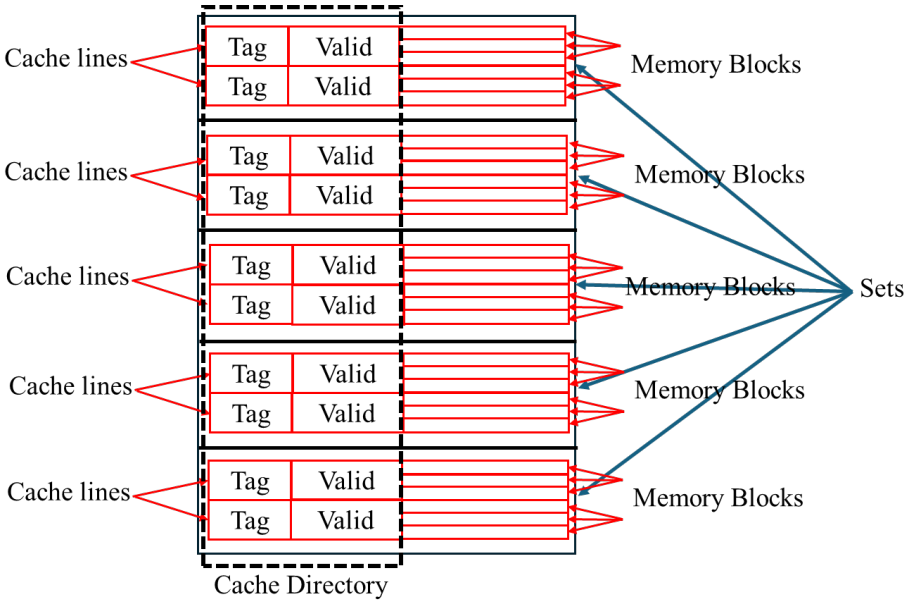


Figure 1.8: Cache Memory

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.

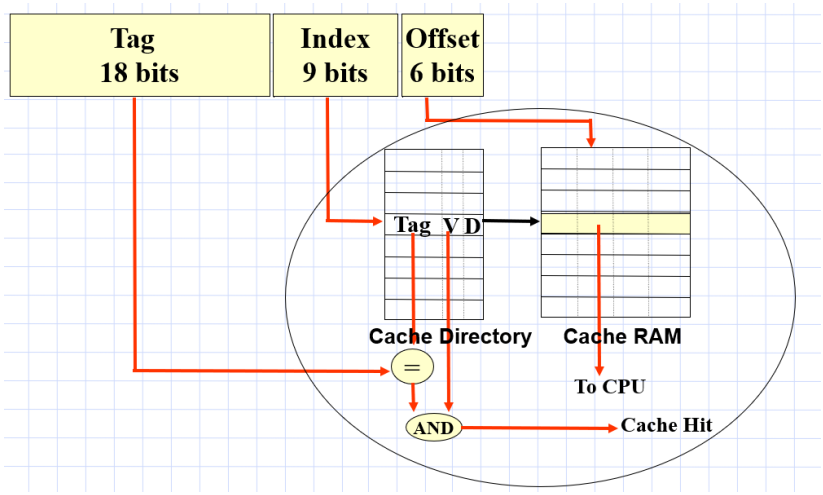


Figure 1.9: Cache Lookup and Access

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.

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.

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.

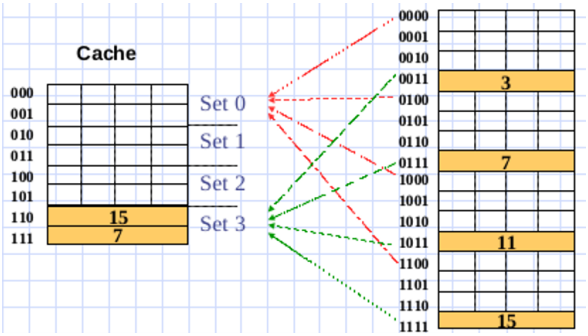


Figure 1.10: Set Associative Mapping

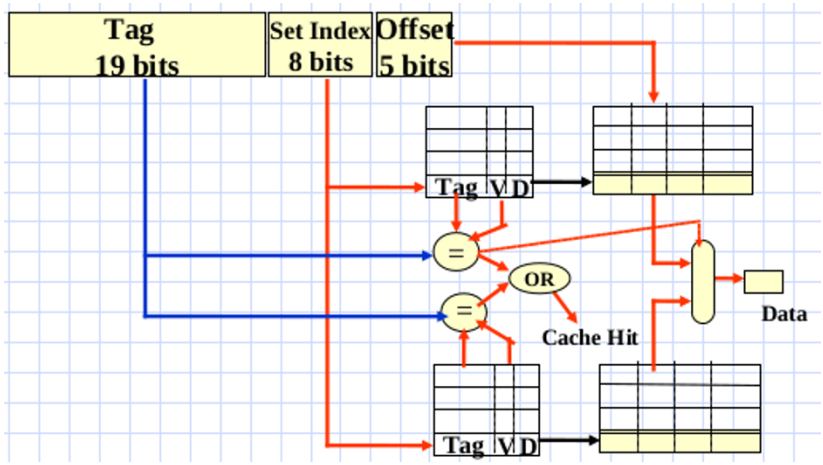
Cache Access Mechanism

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 = 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.

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].

1.3 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.

1.3.1 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.

Listing 1.1: Vector Sum Reduction

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

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:

Listing 1.2: Preloading the Array for Temporal Locality

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

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.

1.3.2 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.

Listing 1.3: Vector Dot Product

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

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 $A[0]$:

```
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:

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

1.3.3 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];
```

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

1.3.4 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.

1.3.5 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:

Listing 1.4: Optimized Matrix Multiplication using temporary variable

```
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;
        }
}
```

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:

Listing 1.5: Matrix multiplication with loop order kji

```
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;
    }
```

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:

Listing 1.6: Matrix multiplication with loop order ikj

```
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];
    }
```

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:

Listing 1.7: Matrix multiplication with loop order jik

```
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;
    }
```

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:

Listing 1.8: Matrix multiplication with loop order jki

```
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;
    }
```

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:

Listing 1.9: Matrix multiplication with loop order kij

```
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];
    }
```

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:

Listing 1.10: Loop Unrolling

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

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

Listing 1.11: Unrolled Once

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

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:

Listing 1.12: Blocking or Tiling

```
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;
            }
        }
```

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.

1.4 Vector Operations

1.4.1 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.

1.4.2 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.

1.4.3 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.

1.4.4 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.

1.4.5 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.

1.4.6 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 Architecture

2.1 Introduction

Moore's Law states

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

This law has been true for the past 50 years. This has led to the increase in the number of cores in a processor.

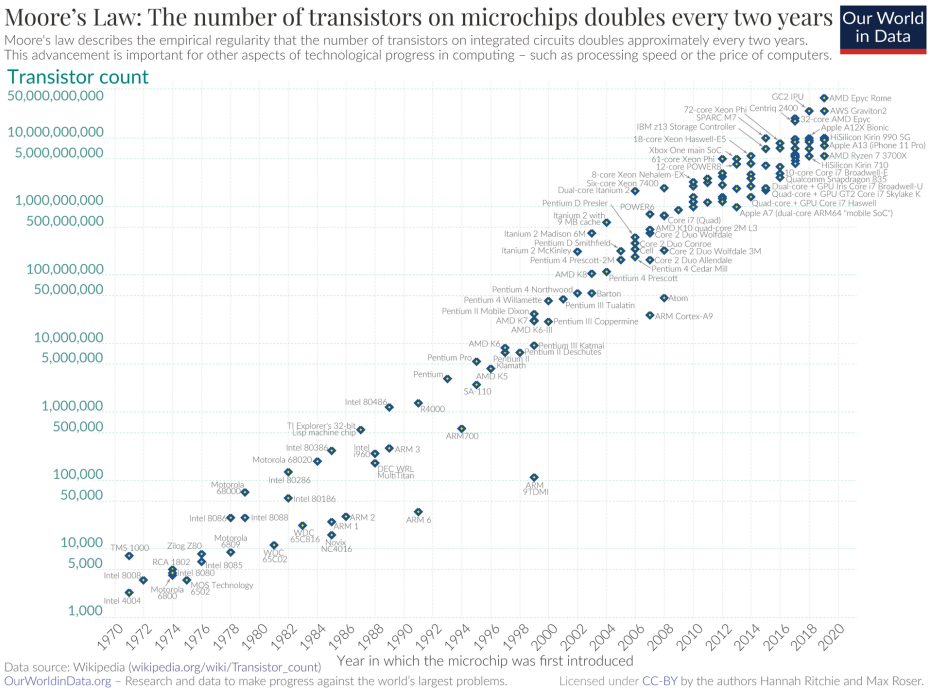


Figure 2.1: Moore's Law Graph

But computing power is saturating. The power consumption is increasing and the heat dissipation is becoming a problem. Moore's Law is not sustainable. The solution to this problem is parallel computing. Parallel Computing is the use of multiple processors to perform a computation. It is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided

into smaller ones, which are then solved concurrently. It leads to Faster execution times from days or months to hours or seconds. Example: Climate Modelling, Bioinformatics, Computational Fluid Dynamics, etc. The large amount of data can be processed in parallel. Example: Google, Facebook, process billions of requests per day. Parallelism cannot be achieved in all problems. It is not always possible to divide the problem into smaller problems. But it is more natural for certain kinds of problems. Example: Climate Modelling.

Because of the Computer Architecture trends the CPU Speeds have saturated and thus slow memory bandwidth is the bottleneck. Parallelism helps in data transfer Overlap.

2.2 Classification of Architectures - Flynn's Taxonomy

Flynn's Taxonomy is a classification of computer architectures based on the number of instruction streams and data streams.

- SISD - Single Instruction, Single Data (Serial Computers)
- SIMD - Single Instruction, Multiple Data
- MISD - Multiple Instruction, Single Data
- MIMD - Multiple Instruction, Multiple Data

2.2.1 SIMD - Single Instruction Multiple Data

It uses vector processors and processor array. In this, a single instruction is executed on multiple data elements simultaneously.

- Vector Processors - They are capable of executing a single instruction on multiple data elements simultaneously.
- Processor Array - It is a collection of processors that work in parallel on different data elements.

For example, CM-2, Cray-90, Intel Xeon Phi, Intel -vector instructions (1028 bytes), etc.

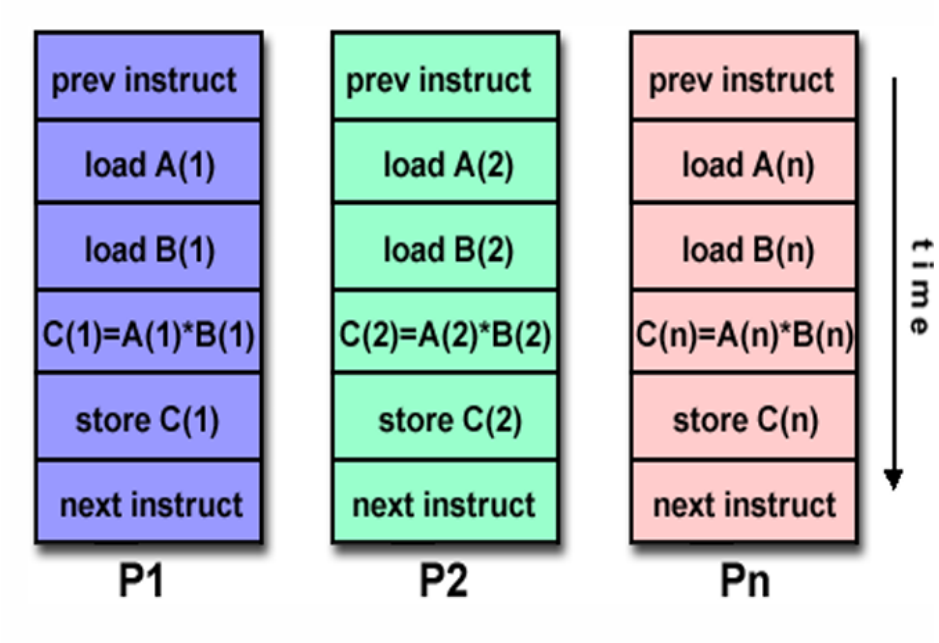


Figure 2.2: SIMD Architecture

Link to Parallel Computing Tutorial This link provided by LLNL provides a good tutorial on parallel computing for more information. Consider the figure 2.2. Say two vectors A and B are given of size n (i.e. n elements). The operation is to perform a element-wise multiplication between the two given vectors. We are given n processors (from P1,...,Pn) Then, in case of SIMD architecture we will pass say ith element of the vector to the ith processor i.e. the first processor will get the first element of the vector, the second processor will get the second element of the vector and so on. Then the processor will simultaneously with time start executing the same instruction on the data element that it has been given. So as shown in the figure 2.2, all the processors will simultaneously load the data element i.e. ith processor will load the ith element of the vector A. P1 will load A(1), P2 will load A(2) and so on Pn will load A(n). Then they will simultaneously execute the next instruction to load the data element of the vector B. Thus, the ith processor will load the ith element of the vector B i.e. P1 will load B(1), P2 will load B(2) and so on Pn will load B(n). Then, once the data has been loaded, the processors will simultaneously execute the next instruction to multiply the data elements that they have loaded. Thus, P1 will multiply A(1) and B(1), P2 will multiply A(2) and B(2) and so on Pn will multiply A(n) and B(n). Then, the processors will simultaneously execute the next instruction to store the element in the result vector C. The ith processor will store the result in the ith element of the vector C. Thus, P1 will store the result in C(1), P2 will store the result in C(2) and so on Pn will store the result in C(n). Thus, the SIMD architecture is used to perform the element-wise multiplication of two vectors in parallel. This is how the SIMD Architecture works.

2.2.2 MISD - Multiple Instruction Single Data

This is not very common. It is used in fault-tolerant systems. In this, multiple instructions are executed on the same data. Some of the examples are:

- Fault-tolerant systems
- Redundant systems

- N-modular redundancy
- Cryptography - Multiple algorithms to crack same data

2.2.3 MIMD - Mutliple Instruction Multiple Data

In this, multiple instructions are executed on multiple data elements. It is the most common architecture. Some of the examples are IBM SP, most supercomputers, cluster, Computational grids, etc.. At a given time step different processors can execute different instructions on different data elements.

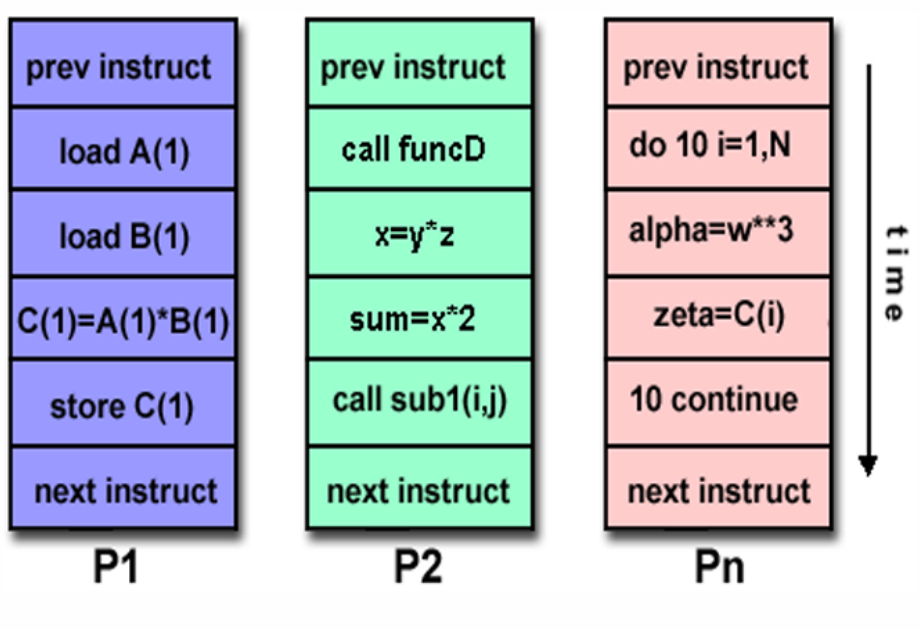


Figure 2.3: MIMD Architecture

As shown in figure 2.3, each of the processor P1, P2, ..., Pn can execute different instructions on different data elements simultaneously in time. P1 is doing element wise multiplication, P2 is running a completely different instruction simultaneously on completely different data elements and so on. Thus, MIMD architecture is used to perform different instructions on different data elements in parallel. This is how the MIMD Architecture works.

2.3 Classification based on Memory

- Shared Memory
- Distributed Memory

2.3.1 Shared Memory

In this, all the processors share the same memory. It requires special computer architecture for memory management (explained later). It is easier to program.

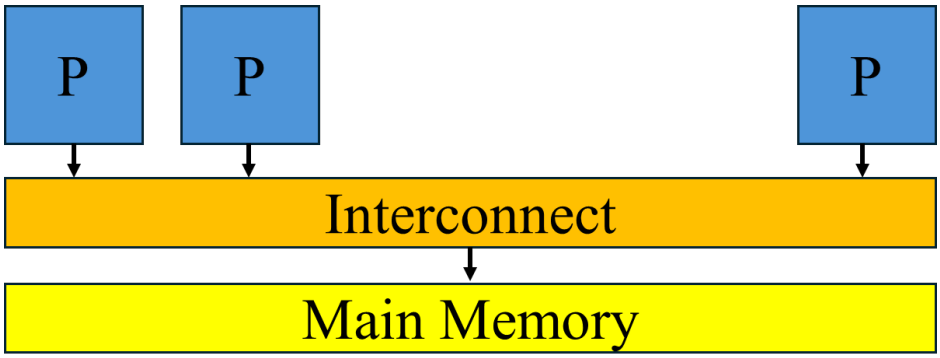


Figure 2.4: Shared Memory Architecture

As shown in figure 2.4, in this the n processors share the same physical address space i.e. say the address is 0x7FFF5FBFFD98 then, whenever a processor accesses the address 0x7FFF5FBFFD98 then it refers to the same physical address location in the memory for all the processors P1, P2, ..., Pn i.e. thus address 0x7FFF5FBFFD98 is at the same physical location for all the processors. Thus, any of the processor can access the address 0x7FFF5FBFFD98 or the address 0x7FFF5FBFFD98 refers to the same physical location for all the processors. This is called sharing the same physical address space. It will be cleared further when you will see Distributed Memory Architecture. It will be explained in detail later how the shared memory is managed in the computer architecture and thus will clear the doubts about how the shared memory is managed. This leads to certain problems in Read and Write and Maintaining Cache Coherence (explained later) Thus, any communication between the processors can be done thorough this shared memory. There are two types of shared memory:

- Uniform Memory Access (UMA) - All processors have equal access time to all memory locations.

- Non-Uniform Memory Access (NUMA) - Access time depends on the location of the memory.

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

Uniform Memory Access (UMA)

In this, all the processors have equal access time to all memory locations. It is easier to program.

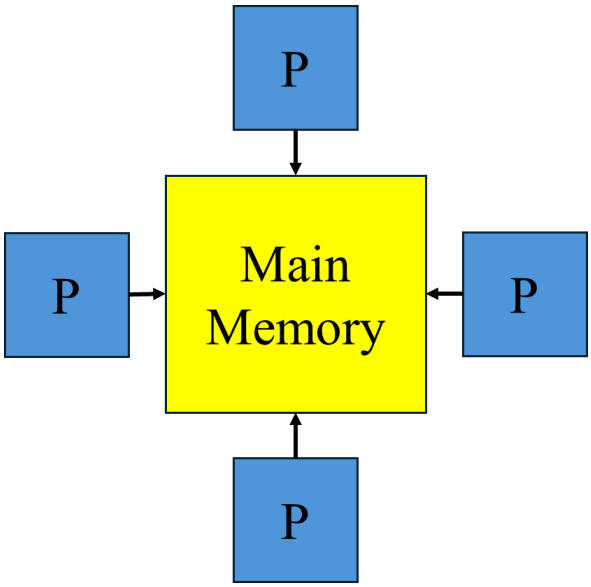


Figure 2.5: Uniform Memory Access Architecture

As shown in figure 2.5, all the processors have equal access time to all the memory locations i.e. the time taken by the processor to access a memory location is same for all the processors. All can be thought of as being at the same distance from the memory location. Thus, all the processors have equal access time to all the memory locations.

Non-Uniform Memory Access (NUMA)

In this, the access time depends on the location of the memory. It is difficult to program.

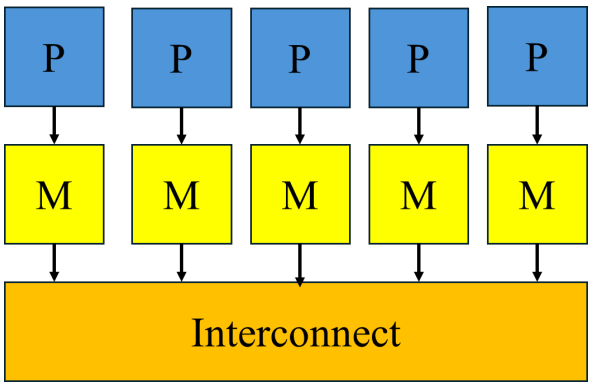


Figure 2.6: Non-Uniform Memory Access Architecture

As shown in figure 2.6, each processor might have some portion of shared physical address space that is physically close to it and therefore accessible to it in less time. Thus, the time taken by the processor to access a memory location depends on the location of the memory.

2.3.2 Distributed Memory Architecture

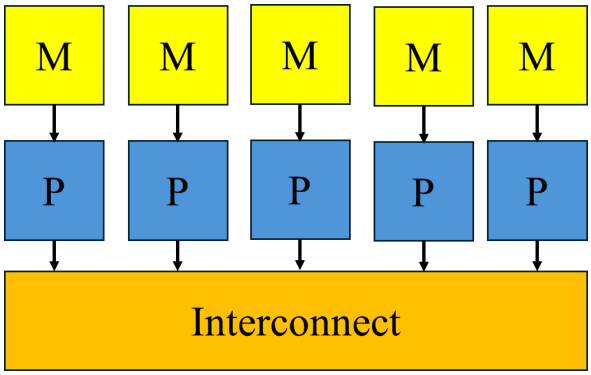


Figure 2.7: Distributed Memory Architecture

As shown in figure 2.7, If two processors access the same memory address say 0x7FFF5FBFFD98 then they will get different data since the physical address space is not shared. Thus, the address 0x7FFF5FBFFD98 refers to different physical locations for different processors. The only way to access data between processor is by sending and recieving data i.e. message passing. This is generally preferred when we wish to use very large number of cores. It is difficult to program.

2.3.3 Shared Memory Architectures: Cache Coherence

Now consider the figure 2.4. As shown in the figure the processors are connected to the Main Memory via Interconnect. Note that even in case of shared memory architecture individual processors have their own caches. Now suppose there is a variable X in the Main Memory and the value of X is 5. Now suppose the processor P1 tries to access the variable X, it will be a Cache miss and hence the value of X will be loaded into the cache of the processor P1. Now suppose the processor P2 tries to access the variable X, it will be a Cache miss and hence the value of X will be loaded into the cache of the processor P2. Now suppose the processor P1 changes the value of X to 10, it will be a Cache hit and hence the value of X will be updated in the cache of the processor P1. Now suppose the processor P2 tries to access the variable X, it will be a Cache hit and hence the value of X used will be the stale value 5. This is called Cache Coherence Problem. The value of X is not consistent across the caches of the processors. There are two ways to solve this problem:

- Write Update Protocol - Propagate Cache lines to other processors on every write to a processor as well as Global memory. Write operation in write update takes longer time because it has to send the data to all the processors. In Read operation, write update is better as the update data is there in local caches of the processor, hence its a faster read.
- Write Invalidate Protocol - Whenever one of the Processor updates the data it sends a signal to other processor to invalidate the data in the cache. Thus, next time if the processor requires that data it will be a cache miss and the data loads from the main memory. Each processor gets the latest data from the main memory. In update protocol, the write updates are faster as it is required to send only an invalidate signal to the processors. In write invalidate protocol, read operation will be a cache miss: thus requires fetching data from memory.

In write update protocol, if the data is present in another processor it will get updated regardless of whether that data is going to be used on not by that processor. This does not happen in write invalidate protocol. Thus, more traffic in write update protocol. This is why most of the systems use write invalidate protocol. NOTE: The specific steps of protocols vary from implementation to implementation.

State Transition Diagram

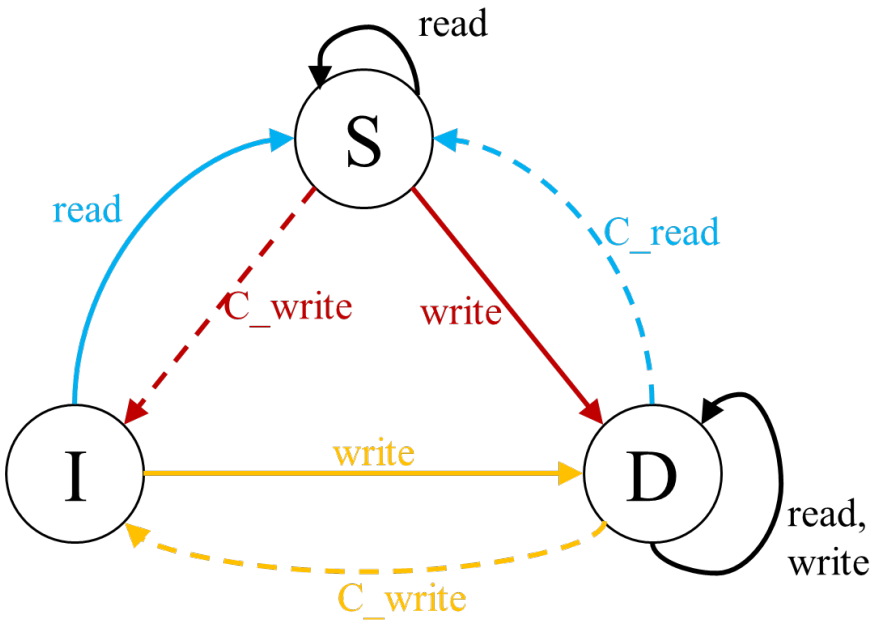


Figure 2.8: State Transition Diagram for invalidate protocol

A cache line can be in either of the three states -Shared (S) - a variable shared by 2 caches, Invalid (I) - another Processor has updated the data, Dirty (D) - the data has been updated by the processor. In the figure shown in 2.8, the state transition diagram for the invalidate protocol is shown. The solid lines indicate the Primary action and the dashed lines indicate the Cache coherence protocol action.

Now consider the figure 2.8. Suppose the cache line is in the shared state S i.e. the data is shared by at least two caches (i.e. in different processors). Then if it is read by either of the processors, it will remain in the shared state S indicated by the solid black line.
If it is written by one of the the processor, it will go to the dirty state D from shared state S for the processor that writes the data as shown by the solid red line from S to D, and because of the cache coherence protocol action, it will go to the invalid state I for the other processor as shown by dashed the red line in the figure 2.8 from S to I.
If the cache line is in the dirty state D i.e. the data has been updated by the processor, then if it is read/write by the processor, it will remain in the dirty state D as shown by solid black line in the figure 2.8 from D to D.
If the cache line is in the invalid state I i.e. the data has been updated by another processor, then, if it is read by the processor, it will be a cache miss and hence the data will be loaded

from the main memory and the cache line will go to the shared state S as shown by the solid blue line in the figure 2.8 going from I to S. Because of this read, the cache coherence protocol action will be to go from the Dirty state D to the shared state S for the other processor as shown by the dashed blue line in the figure 2.8 from D to S.
If the cache line is in the invalid state I i.e. the data has been updated by another processor, then, if it is written by the processor, it will go to the dirty state D as shown by the solid yellow line in the figure 2.8 from I to D. Thus, as a result the processor will send an invalidate signal to all the processors as part of the cache coherence protocol, the cache line which was in Dirty state D for the other processor will thus become invalid. Thus, the dirty state D will go to Invalid state I for the other processor as shown in figure 2.8 by a dashed yellow line from D to I.

2.3.4 Implementation of Cache Coherence Protocols

Snooping Protocol

It is generally used for Bus based architectures. Only one operation can be done at a time. All the memory operations are propagated over the bus and snooped. The idea is that each processor has a specialised cache controller which always snoops onto the bus. Thus, any write operation in any of the cache is publicly broadcasted onto the bus and thus is known by all the other cache controllers of the processors hearing on the bus. Thus, any information related to the cache line present in their cache is invalidated. But this has a problem that only one operation can be done at a time thus, it is slow and not scalable. It requires specialised cache controllers for each processor which increases the cost of architecture. All the data write/read are costly because all the processors are continuously writing on bus and other processors are snooping on the bus. To overcome this problem, the directory based protocol is used.

Directory Based Protocol

It is generally used for Network based architectures. It is scalable and faster. It is used in large scale systems. In the snooping protocol all the other processors even which do not have that data are also snooping on the bus and are listening even though they are not concerned with data. Hence, a mechanism is required where the invalidate signals are sent only to certain processors. This is done by maintaining a directory for each cache line which contains the information about which processors have the data. Thus, the invalidate signals are sent only to the processors which have the data. Thus, instead of broadcasting memory operations to all the processors, cache coherence operations are propagated only to the relevant processors. A centralised directory maintains states of all the cache lines and the processors which have the data. Consider it has M cache lines and P processors, thus a matrix of $M \times P$ is maintained where each row represents a cache line and each column represents a processor. The data about which processor contains that cache line is maintained by presence bits (1 and 0s) in the matrix. wherever the presence bit is 1, it means that the processor has the data and wherever the presence bit is 0, it means that the processor does not have the data. Thus, any invalidate signal will be sent only to the processor with presence bit as 1. The directory also maintains who is the current owner of the cache line. Note that this is called a *Full Bit Vector Scheme* where the number of presence bits is of the $\mathcal{O}(M \times P)$. Huge number of cache lines/processors requires huge

amount of memory for storing directory and less memory for storing the actual data. Thus, requires lots of swaps to memory thus, losing the advantage of parallelism. Ideally, at all times only some of the cache lines will be of interest to only some of the processors. Thus, the matrix $M \times P$ is sparse. Thus, a *Sparse Bit Vector Scheme* is used where the number of presence bits is of the $\mathcal{O}(M + P)$. Here the number of cache lines (m) is much lesser than the total number of cache lines(M) $(M << m)$ and the number of processors (p) is much less than the total number of Processors (P). $(p << P)$. Thus, a limited number of cache lines and processors are of interest at any given time.

2.3.5 False Sharing

A cache is made up of multiple cache lines. All the cache coherence protocols, such as write invalidate, write update deal with the granularity of cache lines and not at the scale of individual variables. Thus, if a variable is in one of the cache lines of caches of two processors and if one of the processors updates some other variable belonging to the same cache line, then even though the variable in the other processor was not updated, since the entire cache line is invalidated, the variable in the other processor will be invalidated. This is called False Sharing.

Let us understand this with an example, Consider modern day caches, a modern day cache lines are of 64 bytes in size. Now consider the data to be stored be of double type which takes 16 bytes. Thus, we can store 4 double type variables in a cache line. Now consider two processors P1 and P2. Suppose a cache line is shared by the processors P1 and P2. Now, say P1 updates the first variable out of the four variables stored in that cache line. Now consider a situation where P2 does not ever required to read or write the first variable but it requires to read or write the second variable. But, since P1 updates the first variable of the cache line and as the cache coherence protocols deal with granularity of cache lines and not of the individual variables, this will lead to the entire cache line being invalidated for the processor P2. Thus, even though the processor P1 deals only with the first variable and P2 deals with only the second variable, the entire cache line will be invalidated for P2. This is called False Sharing. This leads to unnecessary cache misses and thus, the performance of the system is degraded. Thus, even though these two processors are accessing different variables of the same cache line i.e. it is not true sharing, since processor P1 deals only with the first variable and processor P2 deals with only the second variable, but still any updating to either of the variable will lead to invalidating the entire cache line for the other processor. Thus, it is called False sharing. It unnecessarily introduces cache coherence protocols even though the processors are not sharing the same variable or the same element.

2.4 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 2.9. The diagram on the left shows a Static Network and the diagram on the right shows Indirect Network.

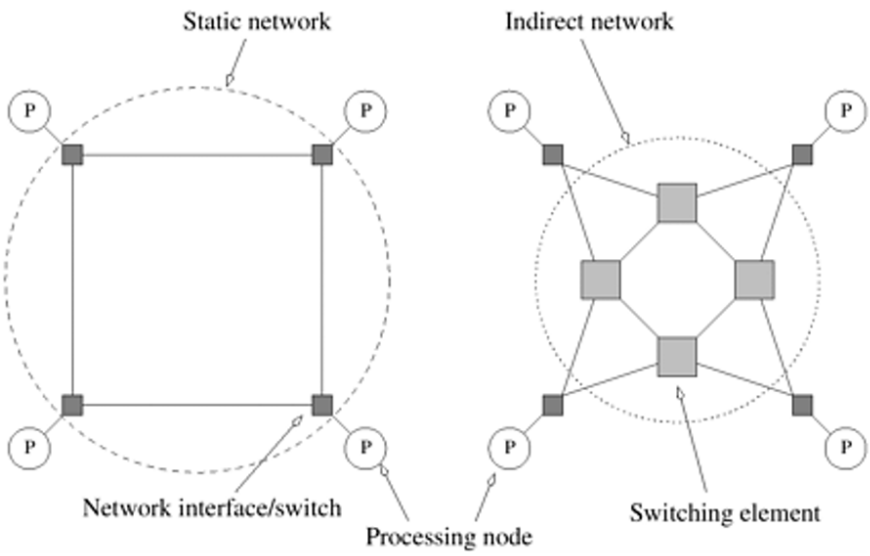


Figure 2.9: Classification of Interconnection Network

2.4.1 Network Topology

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

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 $\mathcal{O}(p)$. The distance between any two nodes in the network is constant $\mathcal{O}(1)$. 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 2.10.

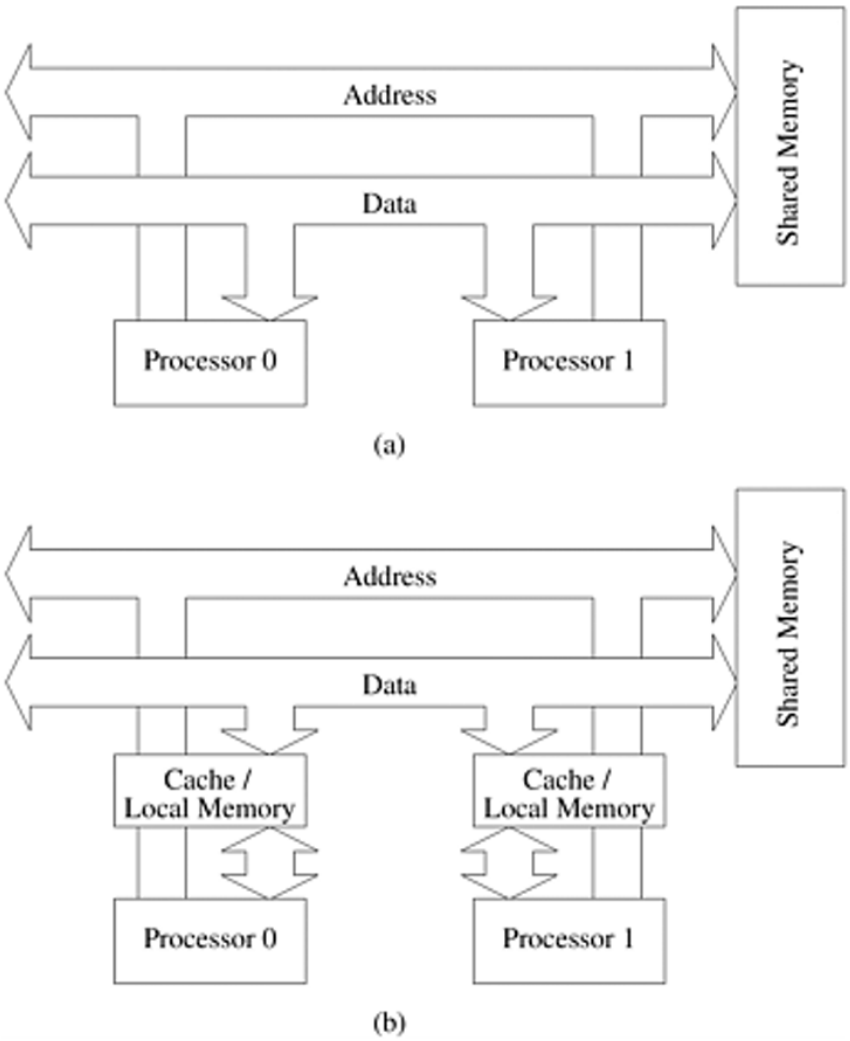


Figure 2.10: Bus-based Network

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.

Cross-bar Networks

Consider the figure 2.11. 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.

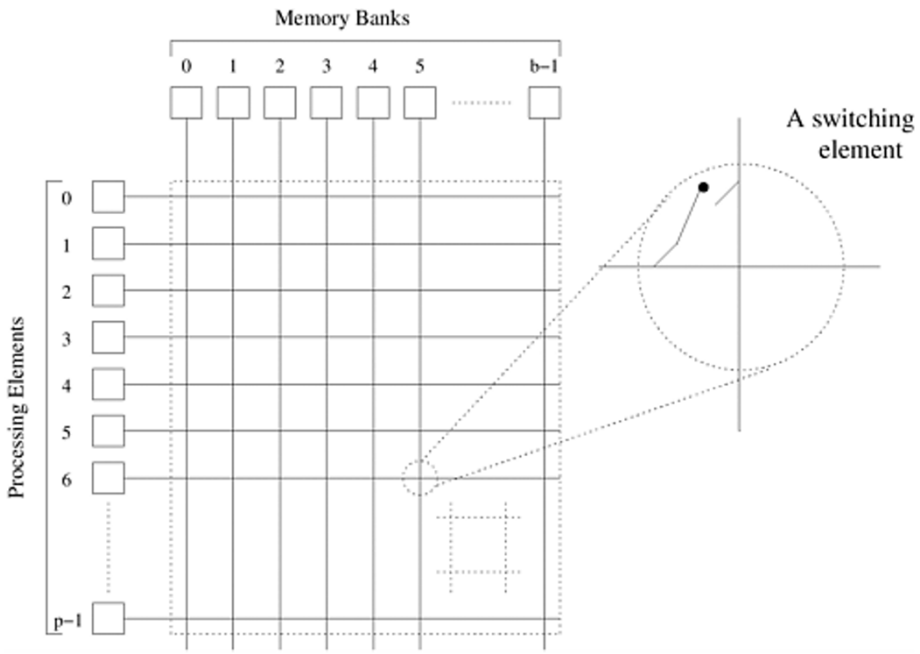


Figure 2.11: Cross-bar Network

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.

Multistage Network

Consider the figure 2.12. 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.

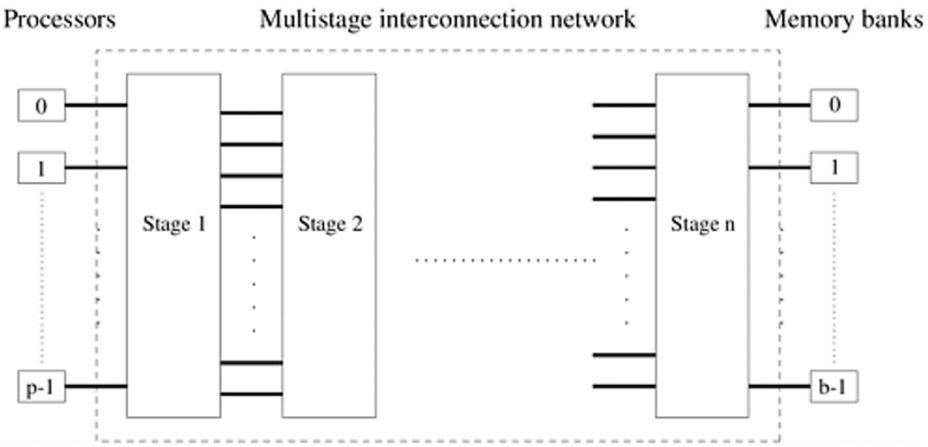


Figure 2.12: Multistage Network

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:

$$\mathrm{j} = \begin{cases} 2i & 0 \leq i \leq p/2 - 1 \\ 2i + 1 - p & p/2 \leq i \leq p - 1 \end{cases} \tag{2.2}$$

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 2.13. The figure shows a perfect shuffle interconnection for eight inputs and outputs.

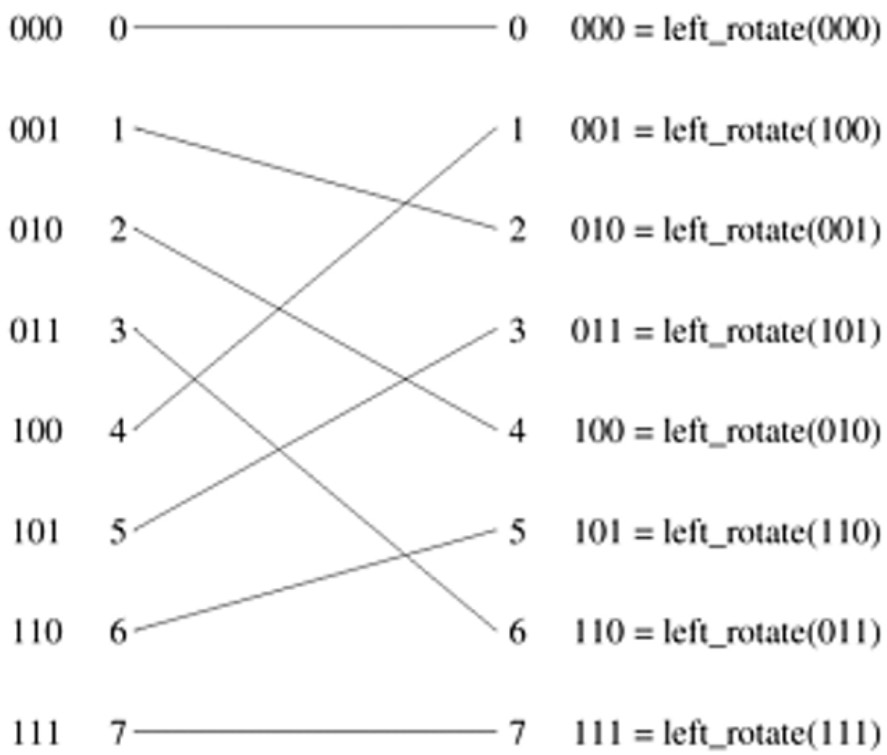


Figure 2.13: A perfect shuffle interconnection for eight inputs and outputs

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 2.14: Pass-through and Cross-over configuration of Switching Nodes

In figure 2.14 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 2.15. We now understand the routing scheme of the omega network.

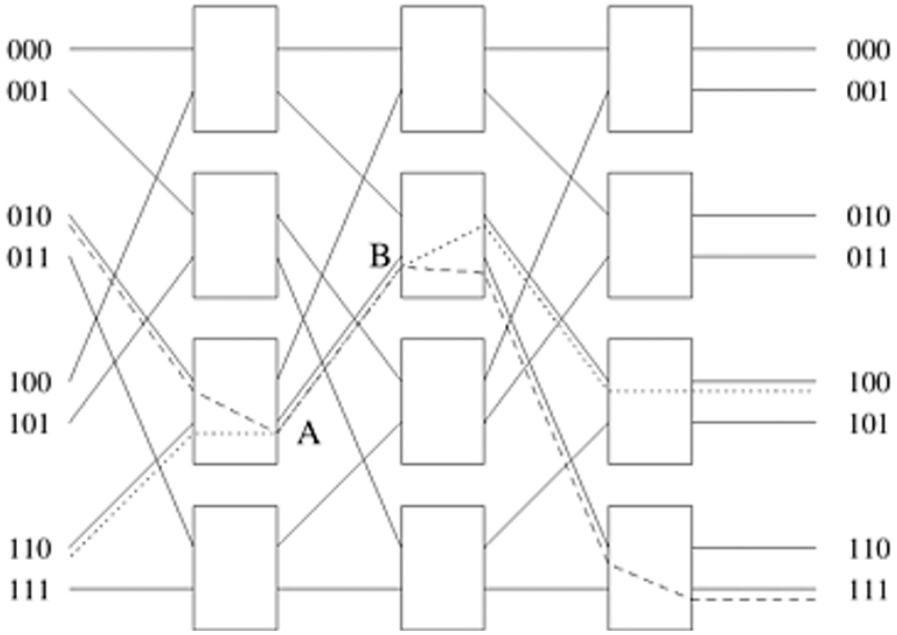


Figure 2.15: Omega Network

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 2.15 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 2.15 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 2.15 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 2.15. This property is called **blocking networks**.

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 2.16.

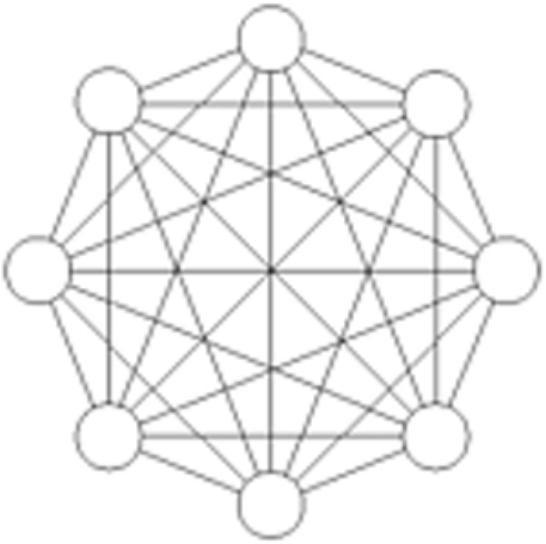


Figure 2.16: Completely Connected Network

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.

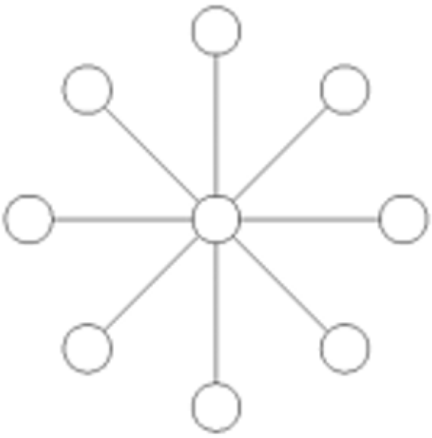


Figure 2.17: Star Connected Network

As shown in the figure 2.17. 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.

Linear Arrays, Meshes and k-d Meshes

Linear arrays are as shown in figure 2.18. It is a one-dimensional array of processing nodes. Each node is connected to its immediate neighbours.

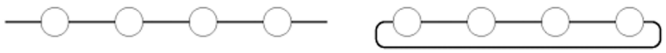


Figure 2.18: Linear Array

As shown in the figure 2.18 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 2.19

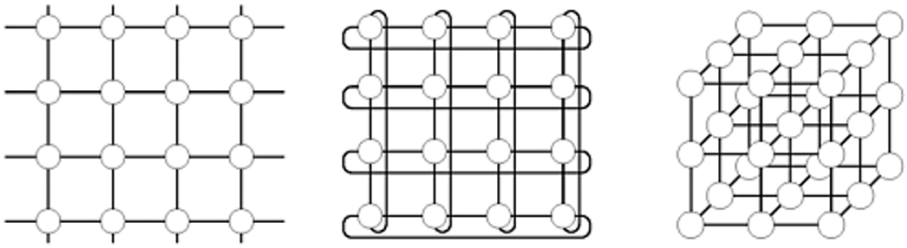


Figure 2.19: Mesh

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 2.20.

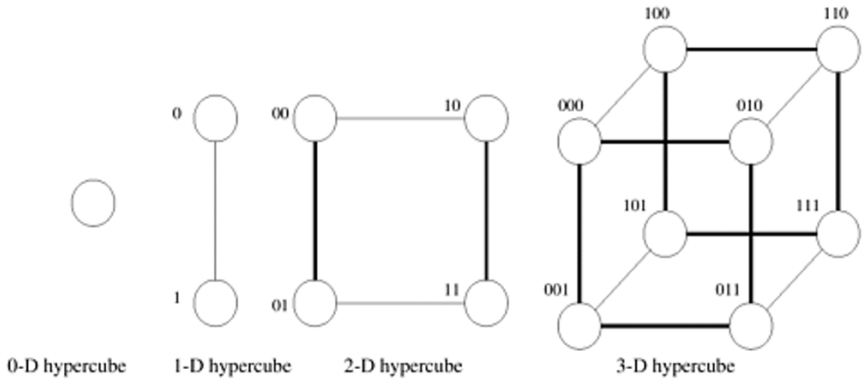


Figure 2.20: Hypercube

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 2.21

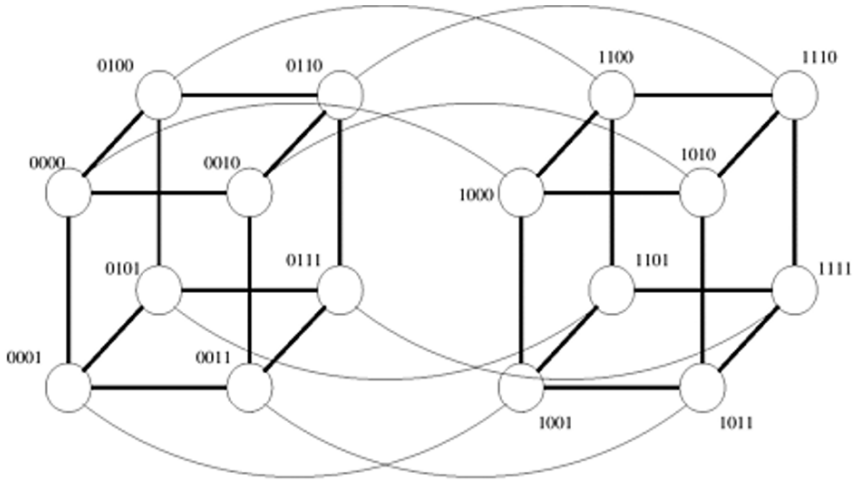


Figure 2.21: 4D Hypercube

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 2.22.

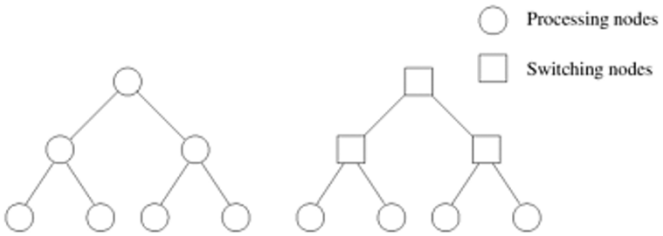


Figure 2.22: Tree Based Network

Note that there is only one path between any pair of nodes. IN the figure 2.22, 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 2.23 by increasing the number of connection links and switching nodes close to the root.

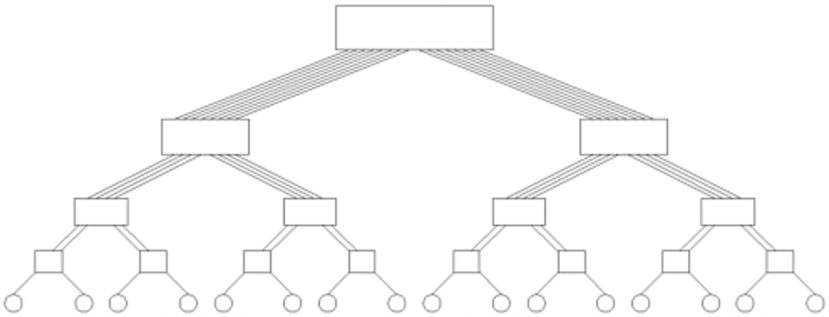


Figure 2.23: Fat Tree with 16 Processing Nodes

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 2.23 a two level fat tree has a diameter of four.

2.4.2 Evaluating Static Networks

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

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 seperate subtree of the root node and a message might have to travel all the way to the root and then down the other subtree.

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.

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,

$$\text{Channel bandwidth} = \text{Channel Width} \times \text{Channel rate} \tag{2.3}$$

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

$$\text{Bisection bandwidth/Cross section bandwidth} = \text{Bisection Width} \times \text{Channel width} \tag{2.4}$$

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.

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.

All of this has been summarise in the following table 2.1.

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 2.1: Summary of Static Networks

2.5 Graphical Processing Units

A CPU has at max of 32 Cores and 8 threads per core. It is good for serial processing. Thus, they are called Multi-core architectures. But for parallel processing, we need more cores. Thus, we need to use GPUs. GPUs have 1000s of cores and are good for parallel processing. Thus, are called Many-Core Architectures. GPUs consist of a large number of light-weight cores which are not as powerful as CPU Cores. In the early, days before 2014 NVIDIA GPUs were used for gaming and video applications for graphical displays. Theses can be done in stages with each stage consisting of independent computations like reading pixels, shading, etc. which can be done in highly parallel manner. Reading of one pixel is independent of reading of other pixel. Processing/rendering of pixels can happen simultaneously. Higher no. of cores of GPU helps in processing in real time. Thus, GPU many-core architectures consisting of light-weight cores as they can perform only simple computations and not very heavy computations are used. Typically GPU and CPU coexists in a heterogeneous setting. GPUs do not exists in standalone setting. A program runs on a CPU(coarse-grain parallelism) and CPU offloads some of the computations to the GPU for light-weight computations (fine-grain parallelism). NVIDIA's GPU architectures consists of CUDA (Compute Unified Device Architecture) which is a parallel computing platform and application programming interface (API) model created by NVIDIA. CUDA programming model is based on C/C++ and is used to program NVIDIA GPUs. It is used to write parallel programs that run on the GPU.

GPUs were initially used for gaming (possible to divide up monitor pixels to process things in parallel). They are also efficient in Matrix calculations.

2.5.1 GPU Architecture

Figure 2.24: GPU Architecture

A GPU Architecture is shown in figure 2.24. The GPU Architecture consists of five main components:

- TPC - Texture Processing Clusters

- SMX - Streaming Multiprocessors

- SP - Single Precision (GPU Core)

- DRAM - Device RAM (GPU RAM)

- ROP - Render Output Unit

GPU cores are called Streaming Multiprocessors (SMX). SP cores are called Single Precision (GPU Core). Coarse level parallelism tasks execute on SMx which gives fine level parallel tasks to execute on SP. Thus, promotes coarse and fine level parallelism. For example, the latest NVIDIA Architecture Kepler consists of 15 SMX with each of 192 SP cores = 2880 SP cores in total. The speed of each of the processors is 745 MHz which is much less than that of a CPU. GPus follow SIMT (Single Instruction Multiple Threads) model. NVIDIA GPU has L2 cache, common memory and individual caches. Each SMX has its own 32 bit registers for storing the context of GPU threads. Consider the figure 2.25 of one SMX, the GPU has 65536 registers, 192 SP cores - Single Precision Cores for single precision calculations, 64 DP cores - Double Precision Cores for double precision calculations, 32 Special Function Units (SFUs) - for certain special function, 32 load/store units for loading data from overall GPU memory to shared memory within each SMX, 16 Texture filtering units, Shared Memory of size 6 KB. CUDA gives the programmer freedom to use some of it as shared memory and some of it as L1 cache. Shared memory is controlled by the programmer and L1 cache is taken care by architecture.

Computations are organised as blocks. Blocks are given to SMX. A block consists of many threads and threads are given to SP cores.

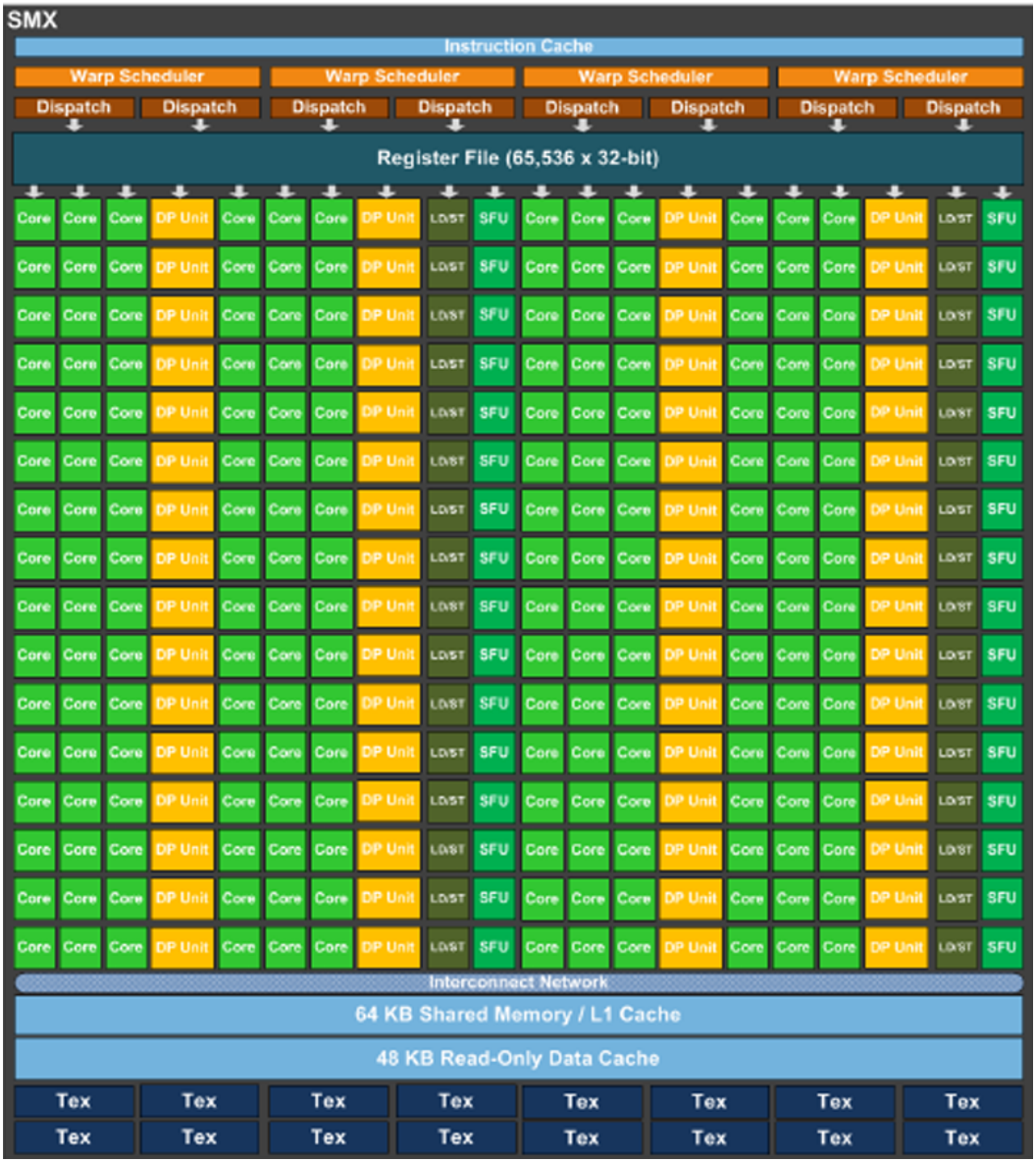


Figure 2.25: SMX Architecture

2.5.2 CUDA Memory Spaces

Consider the figure 2.26 of CUDA Memory Spaces. CPU pushes the relevant data to Global memory to be executed on GPU. It spawns GPU kernels/function which are executed on SMX. Each thread in SMX loads data from global memory/device memory to shared memory.

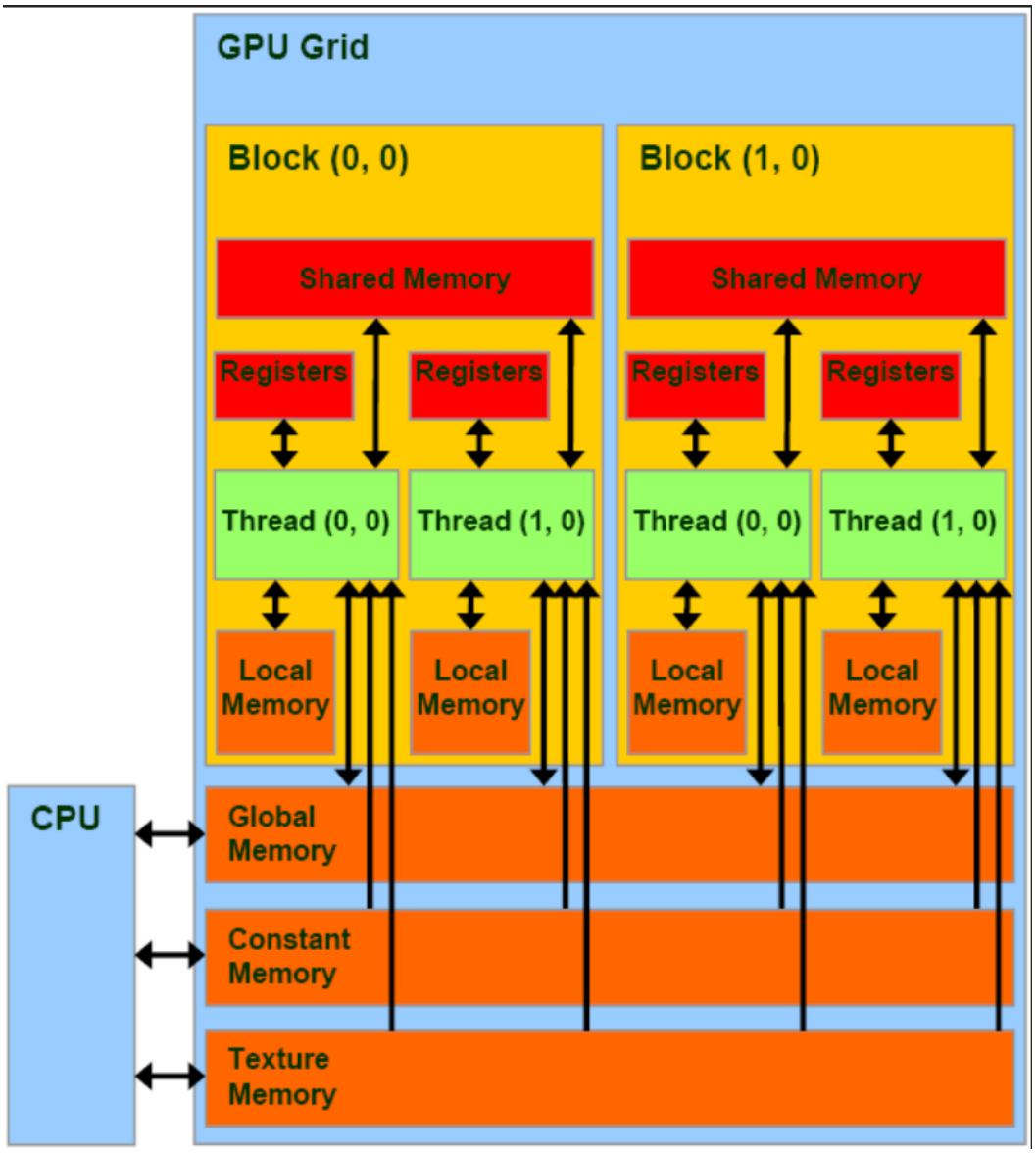


Figure 2.26: CUDA Memory Spaces

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 cycle (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 a 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 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 state of thread (thread block) stored in shared memory and threads stays till execution completion. Unlike in CPUs where in case of context switch memory related to that thread gets dumped to main memory or disk. Thus, loading the data for the thread next requires loading data 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 as it already is loaded in the shared memory. Also, in case of GPUs the cache is explicitly managed by the programmer. The programmer will have to explicitly bring the frequently accessed data from the device to the shared memory. Unlike in CPUs where the cache is managed by the hardware.

2.6 Parallelization Principles

Parallel programs incur overheads which are not present in sequential programs. They are:

- Communication Overhead - Time taken to communicate between processors.
- Synchronization Overhead - Time taken to synchronize between processors.
- Idling Overhead - Time taken by a processor to wait for other processors.

A good parallel program tries to minimise these overheads.

2.6.1 Evaluation of Parallel Programs

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

- Execution Time (T_p) - Time taken by the parallel program to execute.
- Speedup (S) - It is the ratio of 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 time taken by the best sequential program and $T(p,n)$ is the time taken by the parallel program with p processors. Here, n indicates the data size. If we employ p processors we generally expect the speedup of p i.e. the time taken for execution in parallel to be reduced by p times. But, usually, $S(p,n) < p$. This is because of the overheads in parallel programs. The overheads are as discussed due to communication, synchronization and idling.

  In some cases we may even get $S(p,n) > p$. This is called Super linear Speedup. A large data in sequential program may not fit in cache, thus there will be a lot of cache misses. But in parallel program, the data is distributed among the processors and thus the data fits in the cache and there are less cache misses. Thus, decomposing the problem to p processor then each will require a small part of the data and thus the entire data could fit into caches of all the individual processors and thus there will be less cache misses and thus the Speedup for that parallel program will be more than p. This is called Super linear Speedup when the speedup is more than p (expected speedup).

- Efficiency - Now generally, we expect the speed up to be of p times. But, speedup alone does not tell the full picture since the same speedup can be achieved by using different number of processors (Say a speedup of 3 can be achieved by using 3 processors or by using 100 processors). Thus, we normalize the speedup by the number of processors to get efficiency. Efficiency is the ratio of speedup to the number of processors. It gives the idea of how efficiently our program uses the processors.

A low efficiency implies that the overheads are not handled properly and thus are dominating the execution time.

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

Generally, Efficiency is less than 1 but sometimes it can be more than 1 because of the super linear speedup.

- Scalability - It is the ability of the program to maintain efficiency as the number of processors increases. This is important because we want to be able to use more processors to solve larger problems. If the efficiency decreases as the number of processors increases then the program is not scalable. Thus, the analysis of how the program behaves with increasing the number of processors p or increasing the problem size n is called scalability analysis. Scalability thus tell limitations of the program and how well the program scales in relation to number of processors and problem size.

  Ideally, we would like the Speedup to be linear i.e. Speedup should increase linearly with increase in the number of processors. But, in reality, the speedup is sub-linear. Similarly, we would like the efficiency to remain constant with increase in the number of processors but, in practical it decreases with increase in the number of processors. This can be explained by Amdahl's Law.

  Amdahl's Law: The performance improvement to be gained from using some faster mode of executing is limited by the faction of the time the faster mode can be used. For the context of parallel programming, it states that the speedup of a program is limited by the fraction of the program that cannot be parallelized. Thus, overall speedup is given in terms of fractions of computation time with and without enhancement. Consider a sequential program which takes t_s time, which has f_s part of the program which cannot be parallelized and let f_p be the part of the program that can be parallelized. Thus, in an ideal case the time taken by the parallel program will be addition of $f_s t_s$ i.e. time taken by the fraction of the sequential program and $\frac{f_p t_s}{p}$ time taken by the parallelized part of the program. Hence, the total time taken by the parallel program is $f_s t_s + \frac{f_p t_s}{p}$. Thus, speedup is given by the time taken by the ratio of the time taken by the sequential program to the time taken by parallel program. It is given by:

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

  where, f_s is the fraction of the program that cannot be parallelized i.e. serial and f_p is the fraction of the program that can be parallelized. Thus, the speedup is limited by the fraction of the program that cannot be parallelized. This is because not all the operations can be parallelized and some operations are dependent on the previous operations.

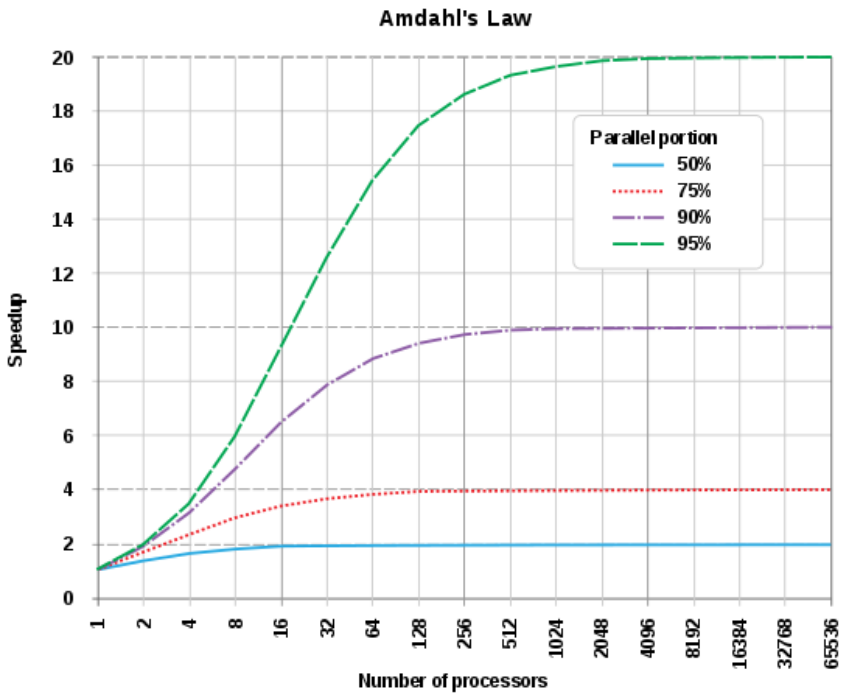


Figure 2.27: Amdahl's Law

It can be seen in the figure 2.27 that even for highly parallelizable programs ($f_p = 0.95$) the speedup curve starts to flatten out as the number of processors increases. Thus, it explains that the speedup curves starts flattening out as the number of processors increases.

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 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 keep the overall time constant and not that the the problem size should be increased linearly with the number of processors.
Strong Scaling: The scaling keeping the problem size constant (Amdahl's law) is called strong scaling.
Weak Scaling: The scaling due to increasing the problem size (Gustafson's Law) is called weak scaling.

- Isoefficiency - Generally, efficiency decreases with increasing number of processors P and increases with increasing problem size N. Isoefficiency is the ability of the program to maintain efficiency as the number of processors increases and the problem size increases. For example, consider the formula for parallel Gaussian elimination in LAPACK package for solving Linear Algebra.

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

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

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

where, t_f is the time taken for a floating point operation, t_v is the time taken for a vector operation and t_m is the time taken for a memory operation. Thus, we can see that from the dominant term in the efficiency equation that as P is increased, N should be increased by $\mathcal{O}(\sqrt{P})$ to keep the efficiency constant. Now in Gaussian Elimination function the amount of computations to be carried out is $\mathcal{O}(N^3)$, thus the isoefficiency function is $\mathcal{O}(P\sqrt{P})$. The isoefficiency function is the ability of the program to maintain efficiency as the number of processors increases and the problem size increases. Thus upon increasing the problem size it is the number of processors that should be increased to keep the efficiency constant. Thus, the lesser the number of processors to increase with a large increase in the problem size the better the program isoefficiency. Thus, the isoefficiency should be as small as possible. Smaller isoefficiency functions imply higher scalability. Consider two parallel programs with isoefficiency functions $W1 = \mathcal{O}(P)$ and $W2 = \mathcal{O}(\sqrt{P})$. Then the second algorithm is considered to be more scalable since only small amount of processors need to be added. Similarly, an algorithm with an isoefficiency function of $\mathcal{O}(P)$ is highly scalable while an algorithm with quadratic or exponential isoefficiency function is poorly scalable.

2.7 Parallel Programming Classification and Steps

2.7.1 Parallel Program Models

Classification based on the way the program and Data is written and executed:

- Single Program Multiple Data (SPMD): We write a single program and run it on multiple processors. Each processor runs the same program but on different data using processor ids to execute certain parts of data. It is the most common model used in parallel programming.
- Mutiple Program Multiple Data (MPMD): Different processors run different programs on different data. It is used in distributed memory systems. For example, used in climate modelling. It is not a very popular model.

2.7.2 Programming Paradigms

Classification based on the way the parallel program uses shared memory model or message passing:

- Shared memory Model - If there is a common global memory to access certain data. For example - Threads, OpenMP, CUDA
- messaage Passing model - If there is no global memory and the processes explicitly send data whenever required. For example, MPI (Message Passing Interface)

2.7.3 Parallelizing a Program

There is no general rule as it all depends on the application. Given a sequential program/algorithm, how to go about producing a parallel version. There are four steps to parallelize a program are:

- Decomposition - Identifying parallel tasks with large extent of possible concurrent activity; splitting the problem into tasks that can be executed in parallel.
- Assignment - Assign the tasks to the processors and grouping the tasks into processes with best load balancing. i.e. amount of work performed by each of the processes must be equal. This is not necessarily done in Decomposition task. It specifies how to group tasks together for a process in order to have balance workload, reduce communication and management cost. This is done by structured approaches such as code inspection (parallel loops) or understanding of application. Sometimes the grouping is done statically and sometimes dynamically (i.e. the grouping is done at runtime) based on the load.

  Both decomposition and assignment steps are usually independent of architecture or programming model, but cost and complexity of using primitives may affect decisions.
- Orchestration - It is the process of managing the execution of the parallel program. It involves managing the synchronization and communication between the processes.

- Mapping - Mapping the processes to the processors. It is the process of mapping the processes to the processors. It is done in such a way that the processes are mapped to the processors in such a way that the communication overhead is minimized.

2.7.4 Data Parallelism and Data Decomposition

Data Parallelism: Given data is divided across the processing entites. **Owner computes own:** Thus, each process own and computes a portion of the data, it decides which data to communicate between other processes. This is called owner computes rule. The data is divided into chunks and each chunk is assigned to a process. This is called data parallelism where the data is divided across the processing entities and thus the parallelism is dictated by the data.

Domain Decomposition: Multi-dimensional domain in simulations divided into sub-domains equal to processing entities is called domain decomposition. The given P processes are arranged in multi-dimensions forming a process grids. Consider the figure 2.28.

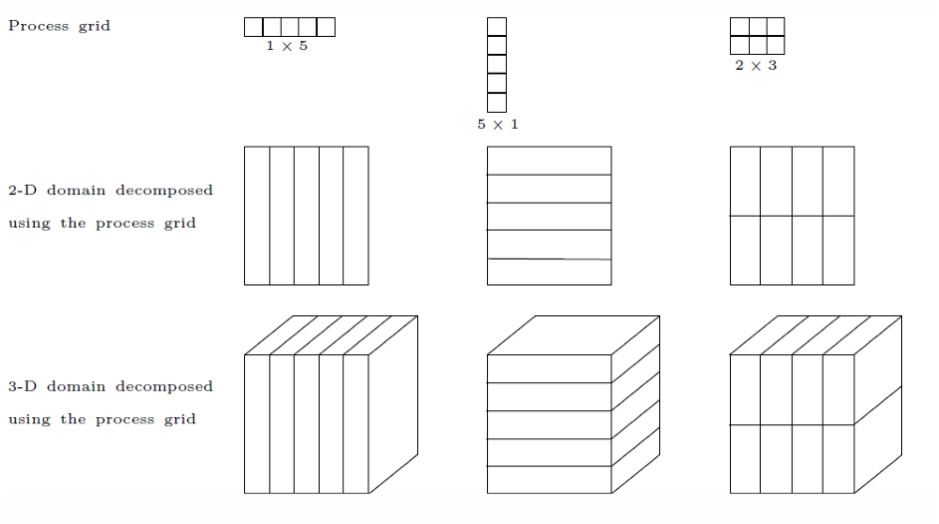


Figure 2.28: Domain Decomposition

It shows that if the process grid is arranged as a 1×5 grid than the 2D domain can be decomposed using the process grid as shown and the domain is divided into 5 equal parts. Each process is assigned a part of the domain. The domain decomposition is done in such a way that the communication between the processes is minimized. Similarly, same concept can be applied for a 3D domain as well as shown in the figure 2.28 shows many such ways for decomposing the domain (data) based on different arrangements of process grids. For a given process gird we decompose the domain so as to map the domain onto process grid such that the communication between the processes is minimized.

2.7.5 Data Distributions

For dividing the data in a dimension using the processes in a dimension, data distribution schemes are followed. Common data distributions are:

- Block Distribution - The data is divided into blocks and each block is assigned to a process. It is the simplest form of data distribution. It is used when the data is uniformly distributed.

- Cyclic Distribution - The data is distributed in a cyclic manner. It is used when the data is not uniformly distributed.
- Block-Cyclic Distribution - It is a combination of block and cyclic distribution. It is used when the data is not uniformly distributed.

For example, consider the figure 2.29.

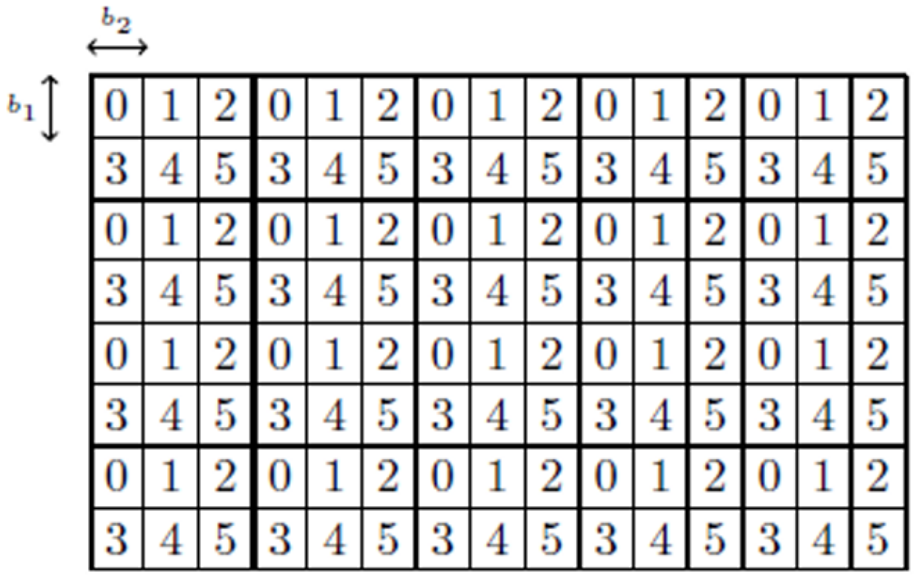


Figure 2.29: Data Distribution

Here, it can be seen that the process grid is arrange as a 2×3 grid and the blocks of data are distributed in a cyclic manner.

2.7.6 Task Parallelism

We decompose into tasks that can be executed in parallel. It is used when the data is not uniformly distributed. The independent tasks are identified and mapped to different processors. The tasks are grouped by a process called mapping. Here, we are trying to achieve two objectives - balance the groups and minimize inter-group dependencies. This is represented as a Task graph. For example, consider the task dependency graph shown in figure 2.30.

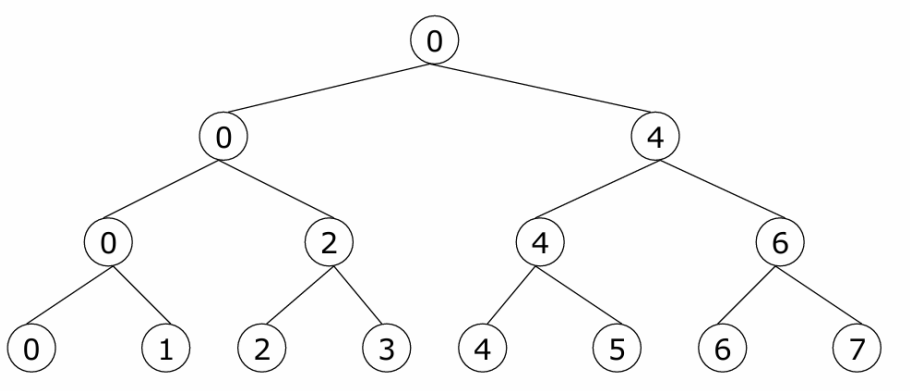


Figure 2.30: Task Dependency Graph

Each node shown represents a task. In general a task can be a statement or a block of statements in a program. Suppose, independent tasks follow a tree like parallelism as shown in figure 2.30. The 8 tasks at the bottom are independent and then they are merged to form 4 tasks and then 2 tasks and then 1 task. Suppose we wish to map this tasks to 8 processors. So, we can map the 8 tasks to 8 processors. Then to merge the tasks we assign it to one of the processor as shown in figure 2.30, this allows as one of the processor already has the data and the other processor can directly access the data from the processor which has the data. Thus, the communication overhead is minimized and similarly for the other tasks. In general, assigning a task graph to processors is an NP complete problem. Thus, we use heuristics to assign the tasks to the processors.

2.7.7 Orchestration

It is the process of managing the execution of the parallel program. It involves managing the synchronization and communication between the processes.

Maximizing data locality:

It is the process of maximizing data locality in a processor and avoid unnecessary communication. Thus, it is about **minimizing the volume of data exchange**. (like not communicating intermediate results) or **Minimizing the frequency of interactions - packing**. For example, say to compute parallel dot products. Say we have two vectors A and B and we wish to find dot product between two. It requires multiplying the corresponding elements of the two vectors and then summing them up. Thus, we can do the multiplication part in parallel across various processes or various threads. Now the communication part can be done in two ways - either we can communicate the intermediate results to the master processor and then sum them up or we can sum them up in the individual processors and then communicate the final result to the master processor. The second approach is better as it minimizes the volume of data exchange. Thus, the orchestration is about minimizing the volume of data exchange and **minimizing the frequency of interactions**. Thus, structuring the communication in such a way that the communication overhead is minimized. Point to point communication between two processors involves latency and bandwidth. The latency is the time taken to establish the connection between the two processors and the bandwidth (measured in MB/s) is the rate at which the data can be communicated between the two processors. By minimizing the frequecny of interactions we can minimize the overall delay i.e. latency. For example, consider that a processor is required to send 10 arrays to another processor, then in order to minimize the frequency of interactions we can pack all the 10 arrays into one packet and send it to the other processor. In sending 10 arrays together to another processor we require to establish connection only once, thus the delay because of latency occus only once but in case if we send it as 10 different arrays, we would be required to establish communication between the processors each time which would mean delay due to latency each time. Thus, Packing will minimize the frequency of interactions and thus the overall delay. This idea is called **packing**. Note that in this case the volume of data remains the same in both the cases but the frequency of interactions is minimized in the second case.

Minimizing contention and hotspots:

(Avoiding communication bottlenecks) Do not use the same communication pattern with the other processes in all the processes. For a given communication pattern, following a certain order will incur a communication bottleneck while following a different order will not incur a communication bottleneck and this can affect the performance. For example, consider a situation where every process communicates data with every other process. This is as shown in table 2.2.

Process	A	B	B	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 2.2: Communication Pattern

In the table 2.2, the communication takes place according to the rule that for each of the processor at time=t, it communicates with t if ($t \neq id$). Thus, starting from time $t = 0$, for Processor A ($id = 0$), it won't communicate to anyone else since its $t = id$, for processor B ($id = 1$), at time $t = 0$ it will communicate (since $t \neq id$) with Processor A. Similarly for processor C, at time $t = 0$, it will communicate (since $t \neq id$) with processor A and so on for Processors D and E, at time $t = 0$ they will communicate with Processor A. Similarly, at $t = 1$, Processor A will communicate with Processor B, for processor B, it will not communicate since ($t = id$), for Processor C it will communicate with Processor B at time $t = 1$ and so on for other processors. This is followed for all the processors at all the time steps. Now see that at some timestep t all the processors will communicate to processor with $id = t$. This will result in communication bottleneck, although it looks like the processors are communicating in parallel, but at some timestep t all the processors will communicate to the same processor and thus the communication bottleneck will occur. Even at the hardware level because of the switches and common network links, it can receive only one package at a time. Now as the number of processes increase say to a 1000, then at some time step t all 999 processes will try and communicate with one of the process at some time step t. Thus leading to communication bottleneck. The solution is to come up with an algorithm where each process follows a different order in which it communicate with the processes so that at no time step all of them end up communicating with the same processor. This is called **minimizing contention and hotspots** which refers to avoiding use of the same communication pattern with the other processes in all the processes.

Overlapping computations with interactions

One of the solutions is **Overlapping computations with interactions**. The idea is to split communication into phases into two parts: one part in which the computations depend on the communicated data (type 1) and second is computations that do not depends on communicated data (type 2). While communicating for type 1 perform computations for type 2. This can be done especially on the receiver side which do not depend on the communicated data and thus can perform the computation. In other words, the computations are being overlapped with the communications.

Replicating data or computations

Balancing the extra computation or storage cost with the gain due to less communication. For example, consider two processes 0 and 1. 0 computes a matrix A by doing an outer product of two vectors (say a and b) and then distributed this matrix A to process 1. Then processes computes with their own part of the matrix. Instead of his approach, we can replicate the vectors a and b to process 1 and then process 1 can compute the matrix A by itself. This will reduce the communication overhead since it will be a lightweight communication compared to sending a part of the Matrix A. But, this will increase the computation overhead since the process 1 will have to compute the matrix A by itself. Now after both of them compute the matrix A, they can work on their respective parts of the matrix. Depending on the vector sizes instead of computing half of the matrix which could be very huge in size, we are communicating only the vectors thereby reducing the communications. This is called **Replicating data or computations** as we are replicating the data and computations to reduce the communication overhead. Here, both of the processes end up replicating the data and computations by computing the matrix A to reduce the communication overhead. Ultimately, it all depends on the computation cost and communication cost and the trade-off between them.

2.7.8 Mapping

Mapping the processes to the processors. Which process runs on which particular processor.

- **Network topology and communication pattern:** It can depend on network topology and communication pattern of processes. The aspect of mapping the communication pattern to the network topology in an intelligent way plays an important role in the performance of the parallel program. For example, consider a 2D mesh network topology. The processes are arranged in a 2D grid and the communication pattern is such that each process communicates with its neighbours. Then we would like to do a natural mapping i.e. map the processes to the processors in such a way that the processes which communicate with each other are mapped to the processors which are close to each other. This will reduce the communication overhead as the communication between the processes will be faster.

- **Heterogeneous systems:** On processor speeds in case of heterogeneous systems (different processors have different speeds). In case of homogeneous systems all the processors have same speed and communication cost is same. In case of heterogeneous systems the communication cost as well as processor speed is different for different processors and thus the mapping can depend on the communication cost. For example, consider a system with 2 processors A and B. Processor A is faster than processor B. Now, if the communication cost between processor A and B is less than the computation cost of processor B then it is better to map the process which communicates with processor B to processor A. This is because the communication cost is less than the computation cost of processor B. Thus, we wish to map the processes to processors in such a way that heavy compute tasks are mapped to faster processors and light compute tasks are mapped to slower processors.

All data and task parallel strategies generally follow static mapping i.e. at the beginning of the program the mapping is done and it remains constant throughout the program. But,

in some cases dynamic mapping is done i.e. the mapping is done at runtime. For example, consider a situation where the load on the processors is not uniform and the load on the processors keeps changing. In such cases, dynamic mapping is done where the mapping is done at runtime based on the load on the processors by the master processor. The idea here is that a process/global memory can hold a set of tasks. The master processors distributed some tasks to all the processes. Once a processor completes its tasks, it asks the coordinator process for more tasks. This is referred to as self-scheduling or work stealing, where the processes are scheduled dynamically based on the load on the processors. Here we don't fix the tasks onto the processors but let the processes decide dynamically which tasks to take up. Thus, even in case of heterogeneous processors, the processes working on a high speed processor can finish tasks quickly and will then steal more work from the slower processors. This is called work stealing.

To summarise the various steps they are as shown in the table 2.3

Step	Architecture-Dependent?	**Description**
Decomposition	Mostly no	Identifying parallel tasks with large extent of possible concurrent activity but not too much
Assignment	Mostly no	Balance workload; reduce communication volume; Assigning the tasks to the processors
Orchestration	Yes	Reduce non inherent communication via data locality; reduce communication and synchronization cost as seen by the processor; reduce serialization at shared resources; schedule tasks to satisfy dependencies early; Managing the execution of the parallel program
Mapping	Yes	Put related processes on the same processor if necessary; exploit network topology; Mapping the processes to the processors

Table 2.3: Summary of Steps

As shown in the table 2.3, the decomposition and assignment steps try to identify independent tasks and balance the workload. Once these tasks are fixed and the communication patters are fixed, the orchestration step tries to minimize the communication overhead and the mapping step tries to map the processes to the processors in such a way that the communication overhead is minimized. The Orchestration and Mapping steps are architecture dependent as they depend on the network topology and the communication pattern of the processes. The first two steps Decomposition and Assignment are mostly independent of the architecture and are done at the programming level.

2.7.9 Example

Given a 2d array of float values, repeatedly average each elements with immediate neighbours until the difference between two iterations is less than some tolerance value. Consider a 2d domain with some grid points, we start with some initial values at this grid point and we keep updating the values over time. For example, it could be solving a heat equation where the heat spreads over a plate, thus the grid points represent temperature at different points on the plate and thus averaging the value reduces the temperature at that point. So we are looking at how the values of these temperatures evolve over time.

```
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 2.1: Parallel Program for Averaging

Consider the figure 2.31 which shows the grid points and the neighbours of the grid points.

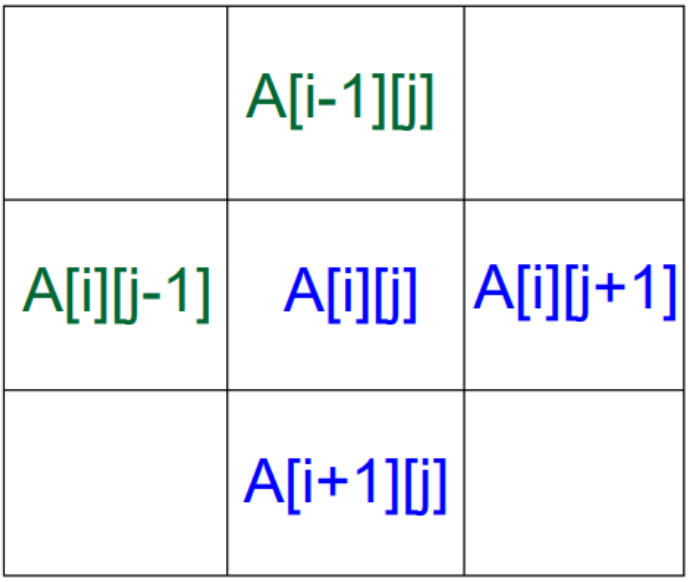


Figure 2.31: Grid Points

In this we go over all the values of the grid points and update the value of the grid point by averaging the values of the neighbours. We keep doing this until the difference of global variable diff between the two iterations is less than some tolerance value. There are two methods to solve this problem: one is Jacobi method where we take the values of the neighbour from the previous time step and the other is Gauss-Seidel method where we take the values of the neighbour from the current time step whenever available (for the lower index values we use the latest values and for the higher index values we use the previous values). Suppose we want to parallelize the program. For the purpose of parallelizing the program we wish to decompose the problem into tasks that can be executed in parallel.

1. One way is to consider each element in parallel. A parallel process or a parallel thread for each of the element i.e. concurrent task for each element update: which would require a max concurrency of n^2 tasks (for n*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 number of 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 context switching between the processes which can thus affect the performance.

2. Another way is to consider tasks for elements in anti-diagonal. Note that values in a particular diagonal depends on the values from the previous diagonal but not among the elements in that diagonal. Now we consider a particular diagonal, for some element in the diagonal for using the Gauss-Siedel method we require the values of the neighbours from the previous diagonal at latest time step and the value from the next diagonal at previous time step. This works out, since we are computing diagonal by diagonal starting from the left top corner and moving towards the right bottom corner. Thus, the values of the neighbours are available from the previous diagonal at the latest time step and the next diagonal at the previous time step. Also note that the values of the elements in the diagonal are independent of each other and thus can be computed in parallel. Thus, we can consider the tasks for the elements in the anti-diagonal and compute the values of the elements in the anti-diagonal in parallel. This way we can reduce the number of tasks to $2n - 1$ tasks (for n*n elements in the grid). This is a better approach than the previous approach as the number of tasks is reduced and the tasks are independent of each other and can be computed in parallel. Note that for the diagonals which are at the ends, will have small number of elements, thus the parallelism will also be very small. This also will have a synchronization const because the processes assigned to a particular diagonal cannot start executing until the earlier diagonals have finished executing. Thus, the parallelism will be limited by the number of elements in the diagonal and the synchronization cost.

3. What we follow is block distribution of the data. Consider the figure 2.32 which shows the block distribution of the data.

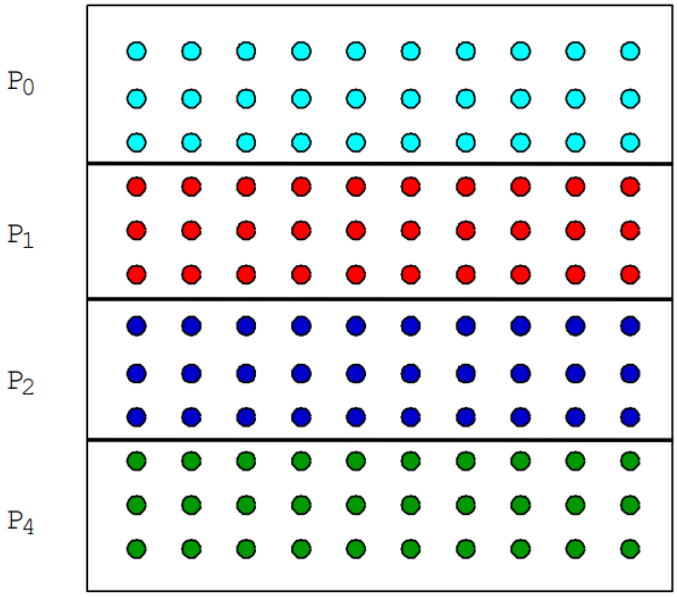


Figure 2.32: Block Distribution

In this we have divided the 2d grid across the rows. First set of rows is assigned to P0, the second set of rows assigned to P1 and so on. Each processor will be responsible

for updating the values of the elements in the rows assigned to it. Now for orchestration step, we try to find out what are the synchronization and communication requirements and we try to ensure correctness and minimize communication and synchronization calls. It depends on the programming/model architecture - shared memory model/message passing model.

Shared address space/Shared memory model

Now we consider Shared Address space (SAS) version of the above program as shown in lstlisting 2.2.

```
int n, nprocs; /* matri: (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);
    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 2.2: Shared Address Space Version

Note that here we are using $n+2$ processors in order to take care of the boundary processors. Here, the array A will be shared and the main process creates the different processes and then it is solved. Let us now look at the psuedo code 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 /* outermost loop over sweeps*/
        diff=0; /*initialize difference to 0*/
            Barriers(barrier1, nprocs);
        for (i=mybeg to myend) do /*sweep for all points of grid*/
            for (j=1 to n) do
                temp=A[i,j]; /*save old value of element*/
                A[i,j]=0.2*(A[i,j]+A[i,j-1]+A[i-1,j]+A[i,j+1]+A[i+1,j])
                diff+=abs(A[i,j]-temp);
            end for
        end for
        if(diff/(n*n) < TOL) then done =1;
    end while
```

```
end procedure
```

Listing 2.3: Solve Function

Now this particular function is going to be executed by all the processes. But depending on the process id different processes will end up processing different parts of the grid. Thus, it uses Single Program Multiple Data Model. It uses the same single program but executed by different processes on different data. The $mybegin = 1 + (n/nprocs) * pid$ and $myend = mybegin + (n/nprocs)$ are the starting and ending points of the grid that the process is responsible for and thus it is calculated based on the process id for each of the processor. Now all the processes enter the while loop. Here, we have a global variable diff and matrix A which is shared by all the processes. At the beginning of the loop all the processes synchronize as shown in the code by Barriers. Each of the processes then execute the statement 11. Although they will be executing the same statement but the beginning and the ending row will differ depending on the process id, thus different processes will be executing this code for different sets of rows. Now, the processes will be updating the values of the elements in the grid and then calculating the difference between the new value and the old value. After taking the average note that they update the value of diff, but since diff is a global variable it gets updated by all the processes which is then later on compared with tolerance to determine the end of the loop. This was just an overview of the program. For understanding the benefits of computing in parallel, we need to understand the importance of parallel statements like barrier. A barrier is a function call that achieves synchronization calls between the processes. When a particular process calls a barrier, it keeps waiting in the barrier call until all the other processes have reached the barrier call. Thus, the barrier call ensures that all the processes have reached the barrier call before any of the processes can proceed further. This is important in the above program as we need to ensure that all the processes have completed the computation of the elements in the grid before we can proceed to the next iteration. This avoids the situation where some of the processes have completed the computation and some of the processes have not completed the computation and thus the processes which have completed the computation will be waiting for the other processes to complete the computation instead of moving with the computation so as to avoid the use of stale values which are still not updated by the other processors.

The matrix A is a global variable and is thus shared by all the processes. But during execution note that the processes are updating the values of the elements in the matrix A. Thus, in case when the elements at the boundary of two processors are updated, it may lead to a problem since it may happen that the processor may not yet have updated some of the values at the boundary but when the other processor updates the element under its control it may use the updated value of the element in some cases and stale value of the element in some cases. Thus, the updates of the elements at the boundary of the processors should be done in a synchronized way. This can be achieved in Case of Jacobi method by maintaining a temporary array which is local to each processor, that stores values for the current time step based on the values stored in the global matrix (which contains values at the previous time step). Once all the processes have reached the end of the iteration which can be found using a barrier call. Then the temporary matrix will be copied to the global matrix and the next iteration will start. This way the updates of the elements at the boundary of the processors will be synchronized in case of Jacobi method. Here, we are ignoring this case for the sake of simplicity and are assuming that this problem is somehow handled internally.

Then we have a waitforend followed by solving matrix A in the code in 2.2. The waitforend is for the main process waiting for all the processes to finish which can be done by making all the process send a synchronization message to the first process which is an All to one communication. Note that diff variable is a global variable thus is a shared variable which is read and written to by all processes. Thus, it may result in garbage values since all the processes try to simultaneously read and change a shared variable. This is because of the race conditions because of context switching at the assembly level language which may result in garbage values. The solution to this is to use locks. A lock is a mechanism that allows only one process to access the shared variable at a time. Thus, when a process is accessing the shared variable, it locks the shared variable and then accesses the shared variable and then unlocks the shared variable. This way only one process can access the shared variable at a time. In computer science, a lock or mutex (from mutual exclusion) is a synchronization primitive. A lock is designed to enforce a mutual exclusion concurrency control policy. This means that only one thread at a time may acquire the lock and all other threads attempting to acquire the lock are blocked until the thread that acquired the lock releases it. In other words, in order to protect the shared variables by means of this locking mechanisms of mutex locks (mutual exclusion locks) where only that process which holds the lock will go ahead and execute these statements and it is only when the process releases this lock that the other processes can go ahead and execute the statements. This is called mutual exclusion because the processes are mutually excluding each other and one process gets exclusive access to that part of the code which is called a critical section. Note that this is also happening with the matrix A which is also a global variable and thus a shared variable. Thus, especially at the boundary values of the processors, where the values may be updated and read simultaneously, it is important to use locks to protect those boundary values when you are using shared arrays.

Another way to resolve these is by using temporary matrices which are local to each processor and then copying the values of the temporary matrix to the global matrix after all the processes have completed the computation. This way the updates of the elements at the boundary of the processors will be synchronized. This is called double buffering.

Note that in the case we are using locking mechanisms, which are very costly and need to be used sparingly, since each time after the processes updates the value, for updating the value in the diff variable, the processes are waiting for the lock to be released by the other process and then only they can proceed further. Note that this happens for each of the grid point updates. This can lead to a lot of waiting time and thus the performance of the program can be affected. So then the question arises, how to reduce the waiting time and the contention for the lock. Note that diff is needed only for calculating the final global value and to find out whether the convergence condition is achieved or not. Hence, instead of each process updating this global diff variable on every update of the grid point, we can have a local diff variable for each process which is updated by each process and then at the end of the iteration the local diff variables of all the processes are added to get the global diff variable. This way the contention for the lock is reduced and the waiting time is reduced since it is only at the time when all the local variables are added to the global diff variable that the lock is required. This is called reduction operation. Thus, we have reduced the locking overhead from $\mathcal{O}(n^2)$ for each grid point update to $\mathcal{O}(p)$ where p is the number of processes. This is a significant improvement in the performance of the program.

Now the updated solve psuedo code is as shown in the listing 2.4.

```
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 /* outermost loop over sweeps*/
        mydiff=diff=0; /*initialize difference to 0*/
            Barriers(barrier1, nprocs);
        for (i=mybeg to myend) do /*sweep for all points of grid*/
            for (j=1 to n) do
                temp=A[i,j]; /*save old value of element*/
                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;
        unlock(diff-lock)
        barrier(barrier1,nprocs);
        if(diff/(n*n) < TOL) then done =1; /* checking the terminating co
        Barrier(barrier1,nprocs);
    end while
end procedure
```

Listing 2.4: Updated Solve Function

Note that now in the pseudo code a new local vairable mydiff has been introduced for each of the processes which is updated by each of the processes and then at the end of the iteration the local mydiff variables of all the processes are added to get the global diff

variable using the locking mechanism as shown in the code 2.4. See that the statement lock(diff-lock) and unlock(diff-lock) are used to lock and unlock (or releasing the lock) the global diff variable respectively This way the contention for the lock is reduced and the waiting time is reduced since it is only at the time when all the local variables are added to the global diff variable that the lock is required. Also note that the barrier call is used to synchronize the processes at the end of the iteration. This is important as we need to ensure that all the processes have completed the computation of the elements in the grid and added their value from mydiff is added to global diff before checking the terminating condition in order to proceed to the next iteration.

Now there is another barrier call after we check the terminating condition. This is because in case if the terminating condition is not met then one of the process may start executing for the next iteration and see that in the beginning of the iteration set diff=0. Thus, while the other processes are still checking the terminating condition, they may encounter diff=0 in the terminating condition which is wrong. Thus, we add another barrier for letting all the processes check the terminating condition with the correct diff value and once all the processes are done, then proceed to the next iteration.

Then, the done=1 condition at the convergence point is updated redundantly by all the processes. Note that the code that does the update is identical to the sequential program.

This is also one of the advantages of a shared memory program over a message passing program. In a shared memory program, the code that is written is identical to the sequential program and the parallelism is achieved by adding the parallel constructs like locks and barriers. Thus, the code is easier to write and debug as compared to a message passing program where the code is completely different from the sequential program and the parallelism is achieved by sending messages between the processes.

Message Passing Version

Here we cannot assume A to be a global shared array which all the processes can access. Because there is no concept of shared variable in message passing model. Each process will have to maintain its part of the matrix. The only way to receive values update by some other processor is by using the explicit message passing between these tow processes. Thus, we need to send the data to the processes.

We do a domain decomposition to decompose the 2D domain into different processes and each process follows the owner compute rule. Here we have more communication and on top of that we will be maintaining local data structures because there is no concept of global data structures. Here, we use something called as ghost rows. This can be understood as follows. Consider the figure 2.33.

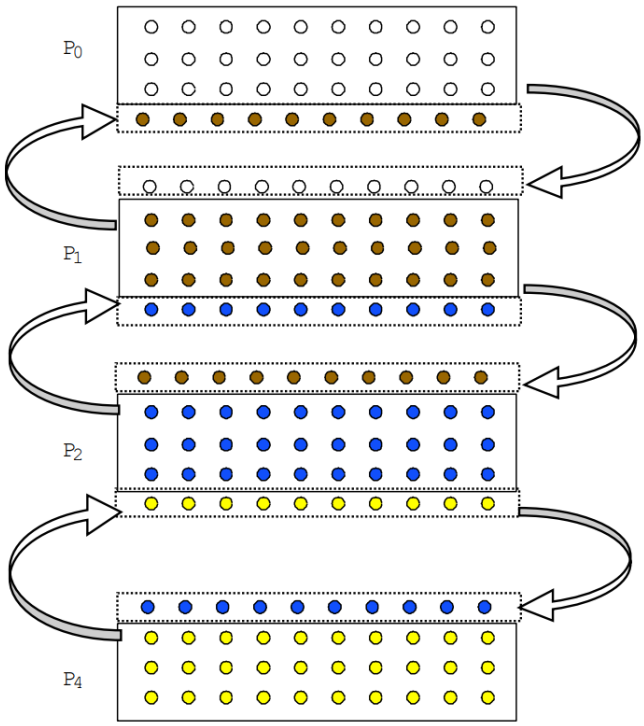


Figure 2.33: Ghost Rows

Here, we have divided the 2D grid into 4 processes as done in the previous case of shared memory model. Each process is responsible for updating the values of the elements in the rows assigned to it. Here Each process will have its local array because there is no concept of shared array in orrder to maintain these elements. In addition each process has an additional ghost row as shown in the figure 2.33. These rows are to accommodate the boundary rows coming from other processes. For example, as shown in the figure 2.33, the process 0 has a ghost row at the bottom which will accommodate the boundary row from process 1. Similarly, the process 1 has a ghost row at the top to accommodate boundary row from process 0 and at the bottom which will accommodate the boundary row from process 2 and so on. The ghost row values will be used for updating the local values and they will not be updated by the processes themselves. THe ghost row values will be used just to update the local values of the processes and they will be updated by the processes which own them. Thus, the name ghost rows. They are also called as halo rows or halo regions or ghost zone regions. Then each process can imply do sequential computation.

Now consider the following Generating processes code for the message passing version as shown in listing 2.5.

```
int n, nprocs; /* matrix: (n+2 x n+2) elements */
float **myA;
main()
begin
    read(n); /*read input parameter: matrix size*/
    realead(nprocs);
    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 2.5: Message Passing Version - Generating Processes

Now consider the following Solve function for the message passing version as shown in listing 2.6 which contains Array allocation and ghost row copying.

```
procedure Solve(A) /* solve the equation system*/
    float **A;
begin
    int i, j, pid, done =0;
    float mydiff, temp;
    myend=(n/nprocs);
    myA=malloc (array of (n/nprocs) x (n) floats);
    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),rpw);
        if(pid!=nprocs-1) then
            RECIEVE(&myA[myend+1,0],n*sizepf(foat),(pid-1),row);
        for (i=1 to myend) do /*sweep for all points of grid*/
            for (j=1 to n) do
                temp=A[i,j]; /*save old value of element*/
                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
        if(pid!=0) then
            SEND(mydiff,sizeof(float),0,DIFF);
            RECIEVE(done,sizeof(int),0,DONE);
        else
            for k=1 to nproces-1 do
                RECEIVE(tempdiff,sizeo(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
    end while
end procedure
```

Listing 2.6: MPI - Solve Function

In the code 2.6, the myA is the local array for each of the processes which is used to store the local values of the elements in the grid for the domain allocated to it (some

subset or rows and all columns). These can be done in many ways: one is to make each of the processes read from a file and the other is to make one of the process can initialize the entire matrix and then distribute it to the various processes. For the sake of simplicity, let us assume that the processes are somehow able to initialize the matrix. Inside the while loop, each of the process maintains a local diff value (no concept of shared/global memory since it is a message passing model nothing is shared). Then, at the beginning of the iteration each processes send their boundary data to the neighbouring processes and the neighbouring processes receive this boundary data and store into the ghost rows of their local array. This is done using the SEND and RECEIVE functions as can be seen in the code. The conditions written here are because the process 0 does not have a process before it and the last process does not have a process after it so they do not need to send or receive the boundary data to any processes before and after and need to send and receive data only to the processes that they share a boundary with. Here pid is the processid which does from 0 to nprocs-1. Thus, based on the process id the send and recieve blocks will change and based on these decisions the processes will change the corresponding send and receive statements. For example from the code we can see that a process with pid not 0 sends its top row to a lower rank process and a process with pid not equal to nprocs-1 sends its bottom row to a higher rank process. Then we have the corresponding receive statement for the same. This way the boundary data is exchanged between the processes and the ghost rows are updated with the boundary data. Then the processes update the values of the elements in the grid and then calculate the local diff value as done in the sequential code.

This time the diff value is not a global variable but a local variable for each of the processes. Then, the processes send their local diff values to the process 0 which is the master process. The process 0 receives the local diff values from all the processes and then adds them to get the global diff value. Then, the process 0 checks the terminating condition and if the terminating condition is met then it sends the done signal to all the processes and the processes terminate. This is the message passing version explained in a nutshell using a Psuedocode.

2.7.10 Notes on Message Passing Version

- In a shared memory model although there is no explicit communication, when a process writes the variable and the other process reads it you can take it as some kind of communication. But the communication is receiving process initiated i.e. when the receiving process receives the data is the communication completed whereas in message passing version a process explicitly sends to another process thus it is sender initiated.

- The synchronization in case of Message passing version is implicit. It is not explicitly done using barriers as in the case of Shared memory model but the synchronization happens using these communications (denoted by SEND and RECEIVE calls in the program). This arises the question whether a deadlock situation can arise due to communications in case for MPI? It may happen in the case when all the processes are waiting for the other process to send something and none of the processes are sending anything. This is called a deadlock situation. Say all the processes execute a RECEIVE statement and until this operation is completed they cannot move on to the next statement to SEND. This will lead to a deadlock situation since all the processes are stuck at RECEIVE statement to receive data but none of the processes are sending anything. This can also be though in other way that consider that all

the processes execute SEND statement and it is such that the processes will move on to the next statement only after it has been received by some other processes. Then in these cases a deadlock situation will happen since all the SEND operations of all the processes will be waiting for each other to execute their RECEIVE statement which will not happen because those processes are also waiting for other processes to consume the message. Thus, all the processes will be at the SEND stage waiting for each other. This can be avoided by using non-blocking communication calls. Such a communication is called synchronized communication where we want that a process return from SEND only if the other process begins to receives the message, thus called synchronized communications since the processes are synchronizing. We can try and break the deadlock by one process sending the data and the other process receiving the data.

- The communication is done at one in whole rows at the beginning of the iteration, not grid-point by grid-point. Instead of sending rows in each grid point the entire row is sent in one shot.

- Note the communication pattern when the mydiff variable is sent from all the processes to the master process and the the master process forms the global value using all this local values. This kind of communication pattern is called All to One communication and is called reduction. This is very common operation in parallel programming where all the processes send their local values to the master process and the master process combines all these local values to form the global value. Thus, many parallel programming libraries support automatic functions for reduction. Finally, the master process sends the done signal to all the processes which is called One to All communication which is also called a Broadcast operation where one of the process broadcasts a message to all the other processes. Thus, we can use the following code by replacing the SEND and RECEIVE operations to broadcast and reduction operations.

```
/*communcate local diff values and determine if done, using red
REDUCE(0,mydiff,sizeof(float),ADD);
if(pid==0) then
    if (mydiff/(n*n)<TOL)then
        done =1;
    endif
        BROADCAST(0,done,sizeof(int),DONE);
```

Note that the reduce operation is called to combine the diff at the process 0 using ADD operation to reduce. The ADD operation is used to add the diff values of all the processes to get the global diff value. There are even other various operators which can be applied to the reduce operation like MAX, MIN, etc. Then the master process checks the terminating condition. The BROADCAST operation is used to broadcast the done signal to all the processes. Thus, it can be simplified using the reduction and broadcast operations.

2.7.11 Send and Receive Alternatives

Recall that in the message passing version we used the SEND and RECEIVE operations to send and receive the data between the processes. Now it may happen that the send opera-

tion i.e. the process will wait until the data is sent (the receiver starts executing its receive and starts getting the message from the sender process) and the receive operation is also i.e. the process will wait until the data is received (the sender starts executing its send and starts sending the message to the receiver process). This can be avoided by using asynchronous operations. Many MPI libraries support Asynchronous communications sends and receives where a process does not have to wait for the other process to start executing i.e. in asynchronous send a process that sends the data does not have to wait until another process starts receiving the data and similarly, in case of asynchronous receive, a process does not have to wait for the other process to start sending the data. Asynchronous operations can be categorised into blocking asynchronous and nonblocking asynchronous operations for receive/send. This will be seen in detail in MPI.

2.7.12 Summary

- Shared Address Space: Shared and private data are explicitly separate. Communication is implicit in access patters and synchronization takes place via atomic operations like locks and barriers on shared data. Synchronization is thus explicit and distinct from data communication.

- Message Passing: There is no concept of shared memory and all the data structures are local data. Data is distributed among local address spaces. Here, the communication is explicit and synchronization (at least for the above example) is implicit i.e. no explicit barrier but there are message passing versions where you can explicit synchronization as well.

- Note that the decomposition and Assignment were similar in both Shared Address Space model and Message passing model, where the difference was in orchestration and mapping. Consider the following table for summary table 2.4.

	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 2.4: Summary

2.8 Shared Memory Parallelism - OpenMP

2.8.1 Introduction

Recall the word, processes, used in case of shared memory address space models. Here, the processes are referred to as threads. Threads are lightweight processes which share the same address space. Individual threads need to use barriers and lock to protect the variables. The program also needs to mention certain variables as shared and certain variables as local. Thus, threads can access the same data structures and variables. The programming library that is used for shared memory parallelism is OpenMP (a portable programming model). OpenMP is a set of compiler directives, library routines, and environment

variables that can be used to specify high-level parallelism in Fortran and C/C++ programs. What are compiler directives? Within your code you can add certain statements which are guidelines to the compiler to do certain special things. For example, there are certain levels of optimization O1, O2,.. one can direct the compiler to optimize certain section of code to a certain level of code using these compiler directives. In case of OpenMP compiler directives are given related to shared memory parallelism. Compiler directives are also called pragmas in OpenMP.

It is very easy to program using OpenMP to convert the sequential program by writing the sequential code and then using the OpenMP directives and execute using OpenMP flags to achieve parallelism. First version of OpenMP came in 1997 with the latest version 4.5 in 2015.

2.8.2 Fork-Join Model

OpenMP uses a fork-join model. The main program is called the master thread and the master thread forks a team of threads whenever it encounters parallel construct. The team of threads then execute the parallel region and then the threads join back to the master thread. This is as shown in the figure 2.34.

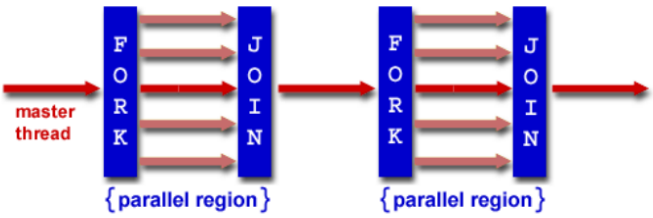


Figure 2.34: Fork-Join Model

The program begins as a single thread called master thread. Then when it encounters parallel construct, it forks/spawns a team of threads. The statement are then executed in parallel for the parallel region. At the end of the parallel region the team of threads synchronize and terminate and join back to the master thread.

This is one of the good things about OpenMP has selective parallelism i.e. only certain sections of the code are executed in parallel based on the requirement. Thereby we are adhering to Amhadal's Law which says that not all parts of the code can be parallelized and only certain parts of the code can be parallelized. Thus we are not spawning all of the program in parallel and only those where we require parallelism.

OpenMP consists of:

- Work-sharing constructs: specifies how region should be parallelised and distributed among threads.
- Synchronization constructs: for providing synchronization between threads.
- Data environment constructs: for specifying which data is shared and which is private.
- Library calls, environment variables: special function calls and set environment variables. (set env command before executing your code).

A construct is a directive combined with the region where the directive applies. Above shown are the three constructs OpenMP supports. OpenMP support loop-level parallelism to parallelize mainly for/do loops. If there are no dependencies between the iterations of the for loops then we can distribute this iterations to the processes where some process executes some set of iterations and the other process executes the other set of iterations. This is called loop-level parallelism. It gives the user a control over the fine-grained parallelism i.e.what regions to be parallelised. Hence, we call this as fine-level parallelism. It follows Amdahl's law which says that only certain parts of the sequential code can be parallelized which is what is seen in this case. Note that in this we can also change the phases of parallelism meaning that based on the requirement we can change the number of threads that are executing the parallel region. Regions which can be more paralleized, one can have more number of threads. This is called dynamic parallelism. In addition OpenMP also supports coarse-level parallelism i.e. parallelism where executing completely two different sections of code in parallel in addition to fine-level parallelism. In the recent versions of OpenMP it also support executions on accelerators (GPUs), SIMD vectorizations (automatically updates a vector in parallel units) and task-core affinity. The task-core affinity refers to the fact that the tasks can be explicitly mapped to a particular core. This can be helpful when we want to keep data which is highly related closer in order to save time.

2.8.3 Parallel Construct

omp parallel is the parallel construct in OpenMP. It is used to create a team of threads. The syntax is as follows:

```
#pragma omp parallel
{
    /*parallel region*/
}
```

The parallel construct is followed by a block of code which is executed by all the threads in the team. The block of code is called the parallel region. So when a compiler encounters the pragma omp parallel construct it creates a team of threads and the subsequent region inside the block is then executed in parallel. Before the encounter with the omp parallel the compiler will execute the code in the sequential manner on a master thread. This is thus achieved by the parallel directive along with the block of code to be executed in parallel enclosed in curly braces which will be executed in parallel by a number of threads is called parallel construct.

How many number of threads? There are various ways to specify it, one can set an environment variable from command line set env omp numthreads=number of threads. Another way is to use the omp set numthreads(number of threads) function in the code. We can also specify the number of threads in clauses. The following clauses can be used:

- if (condition): will execute in parallel if the condition is true.
- num_threads (n): will execute in parallel using n threads.
- default (shared — none): specifies the default data scope for variables within a parallel region. Should the variables be shared or private among the threads.
- private (list): specifies a list of variables that are private to each thread.

- firstprivate(list): specifies a list of variables that are private to each thread and are initialized with the value of the variable outside the parallel region.
- shared(list): specifies a list of variables that are shared among all the threads.
- copyin(list): specifies a list of variables that are shared among all the threads and are initialized with the value of the variable outside the parallel region.
- reduction(operator:list): specifies a list of variables that are private to each threads and which undergo changes by different threads. Then at the end of this parallel region all of these private variables are combined using the operator specified to give a value.
- proc bind (master — close — spread): specifies the binding of the threads to the cores. How you want the task to be distributed to the cores. To the same core as the master thread or close to the master thread or spread.

Consider the code as shown in the listing below which return the number of threads used in the parallel region. So say n threads were used then it would return n, n times each time when executed by one of the thread.

```
#include <omp.h>
int main()
{
    int nthreads;
    #pragma omp parallel
    {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }
    return 0;
}
```

I encourage you to run these codes by simply copying and pasting it into your IDE and then execute them to see the results.

Consider another Hello World code as shown in the listing below.

```
#include <omp.h>
int main()
{
    int nthreads,tid;
    #pragma omp parallel private(nthreads,tid){
        printf("Hello World\n");
    }
    return 0;
}
```

So some number of threads will be spawned and you will see the Hello World printed by each of the threads.

2.8.4 Work sharing construct

The work sharing construct is used to distribute the work among the threads.using the parallel construct we have spawned multiple threads. Now to specify how to distribute the work among the threads we use work sharing construct. There are three types of work sharing constructs - loops, sections, single.

for-loop

It is used for a for-loop or a do loop using pragma omp for (clause) as in the case of parallel construct. The following clauses can be used :

- private (list)
- firstprivate(list)
- lastprivate(list): here the value of the variable in the list is equal to the value that is set by the last thread.
- linear(list):
- reduction(operator:list)
- schedule([modifier[,modifier]:]kind[,chunksize]): To distribute the iterations among the threads, schedule clause specifies how we schedule the iterations of the loop among the different threads.
- collapse(n): Useful for nested level parallelism. Say two for loops are nested and we want to parallelize both of them then we can use the collapse clause. In general only one of the for loop collapses but using collapse clause we can parallelise both of them by collapsing both the for loops. Say first for loop executes for 3 iterations and the second for loop executes for two iterations, Then using collapse we can consider them as a total of six iterations in parallel across the threads. If we don't use the collapse clause only the outermost three iterations will be distributed among the threads.
- ordered(n): In some cases we want threads to execute in order i.e. we want the threads managed by the first thread to be executed first and then the second thread and so on.
- nowait: To tell the threads not to wait for each other at the end of for loop. By default all the threads will be waiting for each other for all the threads to finish executing the for loop.

For construct is the most commonly used and th most important construct in OpenMP. Note that we are assuming that the iterations can be independently executed and simultaneously in parallel by the threads. This works as long as the iterations are not dependent on each other whereas in the case when the iterations are dependent on each other say we are using a value in the loop which is dependent on the value resulted in previous loop, in those cases the programmer should not use for construct.

Schedule clause: The assignment of iterations to threads depend on the schedule clause. There are five schedule clauses.

1. schedule(static,chink size): In this the total number of iterations of the for loop will divided by this chunk size to obtain chunks and these chunks are distributed to the threads in a round-robin manner. For example, say 20 iterations of for loop and say chink size is 2 then 20/2 = 10 chunks are formed which each consisting of two iterations (first chunk consists of first two iterations and so on) and say we have only four threads then these chunks are distributed among these four threads in a round-robin manner. First chunk to first thread, second chunk to second thread, third chunk to third thread, fourth chunk to fourth thread and back from fifth chunk to first thread and so on.

2. schedule(dynamic, chunksize): It is the same as above but here the chunks can be distributed dynamically, based on OpenMP runtime system libraries own policy to distribute these chunks to the threads that are ready to execute in a dynamic fashion.

3. schedule(runtime): Here we give complete freedom to the OpenMP library to distribute the iterations based on the availability of thread.

Consider the following for-loop example code 2.7.

```
#include<omp.h>
#define CHUNKSIZE 100
#define N 1000
main(){
    int i,chunk;
    float a[N],b[N],c[N];
    /*some initializations*/
    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];
            }/*end of parallel section*/
        }
}
```

Listing 2.7: for-Example: Addition of two Arrays

Here we ared adding two arrays a and b to form a third array c. Note that we can perform this additions completely independent of each other and hence a good opportunity for using parallelism. First for parallelization we need to spawn the threads for which we used pragma omp parallel shared(a,b,c,chunk) private(i). Here a,b,c are shared among all the threads (since they could be very large arrays) and chunk is also a shared variable and iterator i is private variable to each thread because each thread is going to have its own set of iterations, and those set of iterations should be managed by a private iterator i to avoid any conflicts between which iterations to perform among the threads. Then we use the for construct to parallelize the for loop with a dynamic schedule clause and nowait for threads to not wait after finishing their iterations. Note we can also combine the two

pragma into one line in the code as pragma omp parallel for shared(a,b,c,chunk) private(i) schedule(dynamic,chunk) nowait.

Coarse Level parallelism - Sections and Tasks

In addition to fine-level parallelism as shown with the for loop example, OpenMP also supports coarse-level parallelism. When we talk about given a set of iterations to a thread that is a fine-level parallelism regarding the iterations of the for loop. but now suppose that we wish to execute two different functions containing many such for loops in parallel then it requires a coarse-level parallelism, we can use the sections construct. Consider the following code showing use of sections construct as shown in the listing 2.8.

```
#pragma omp parllel sections <\clause-list\>
{
    #pragma omp section
        structure-block i
    #pragma omp section
        structure-block j
    ...
}
```

Listing 2.8: Sections Construct

NOw suppose the structure blocks consists of functions and we wish to execute them in parallel, then as shown, we first enclose the entire in pragma omp sections and then the small parts which we wish to execute in parallel to be in pragma omp parallel section Then the first section will be executed by one thread and the second section will be executed by another thread and so on.

OpenMP also supports dynamically spawning tasks. Suppose you have a parallel region and you wish to execute a few set of statements (a task) i.e. we are dynamically creating a task. This generally occurs in large simulations where depending on the requirement one might need to create a task dynamically. This is done using the task construct. This task will be automatically assigned to one of the threads. This is another coarse-level parallelism that OpenMP supports. Tasks is more dynamic construct where your code might be creating tasks dynamically and you want to assign them to the threads dynamically. Task will be assigned to one of the threads you have spawned by OpenMP. You can also use a depend clause which allows us to create dependencies between the tasks by specifying certain variable_list. Suppose a tasks uses some variable x and executes its program and now we wish to use that variable for another task such that the second task depends on the first task. Then we can use the depend clause to specify that variable and thus to have the second task depends on the first task. This is done using the depend clause. Thus, it creates dependencies between variables and thus, second task begins execution only after the first task has finished execution. Thus, it creates the dependencies between the tasks. Now you can dynamically create task graphs where nodes in the graphs represent the tasks and the edges between the nodes represent the dependencies between the tasks. This is a very powerful construct in OpenMP. Thus, with the tasks construct OpenMP can follow its own scheduling mechanism to assign the tasks to the threads such that the dependencies are maintained. This is as shown in the listing 2.9.

```
#pragma omp parallel \<clause-list\>
{
```

```
    ...
    #pragma omp task \<clause-list\>
    ...
    #pragma omp task depend(dependence type: variable\_list)
}
```

Listing 2.9: Tasks Construct

2.8.5 Synchronization Directives

Having distribute work among the threads there is always a chance of threads accessing some common variables. Hence, you need to synchronize the threads and protect certain variables using locking mechanisms. OpenMP supports this using the following synchronization directives: Note that all of this are already enclosed in a parallel region/parallel construct.

- pragma omp master: This is used to specify that the block of code following this directive is executed by the master thread only. This is generally useful for print statements where you want only one thread to print the output. Say to print the value of a variable.
- pragma omp critical: This is used to specify that the block of code following this directive is executed by only one thread at a time. Say you are updating a variable and we do not want multiple threads to update the variable at the same time else it might result in garbage values. Then we can use the critical directive to protect the variable by enclosing that statement in this directive and only one thread can update the variable at a time.
- pragma omp barrier: This is used to specify that the threads wait at this point until all the threads have reached this point.
- pragma omp atomic: This is used to specify that the operation statement (expression) following this directive is executed atomically i.e. only one thread can execute this operation at a time.
- pragma omp ordered: This is used to specify that the block of code following this directive is executed in the order of the thread number. Say we want the first thread to execute those statements first and then the second thread to execute those statements and so on. This is useful when we want to maintain the order of the threads executing the statements.Say for example an array is distributed among threads and then we wish to print the entire array. Then we would like to print the array in the order of the threads i.e. the first set of elements residing in the first thread to print and then the second and so on.
- pragma omp flush: This is used to specify that the threads flush their private data to the shared memory. This is used to ensure that the shared memory is updated with the latest values of the private data. For example, say a thread has updated a variable and we want to ensure that the shared memory is updated with the latest value of the variable. Now there is no guarantee that the variable can be updated immediately to the shared array and be visible to the other threads. There may be a time lapse before

it gets written to the shared array and before its visible to the other threads. Because the operating system maintains its own buffers where it may store the variables. Instead, if we wish to immediately flushed out so that it is visible to the other threads, then we can use the flush directive. Flush directive is used when we wish to see consistent vie of memory among the threads. Using flush directive, Thread-visible variables (global variables, shared variables etc..) are written to memory. If a variable list is used, only variables in the list are flushed. Consider the following code as shown in the listing 2.10.

```
int sync(NUMBER_OF_THREDS);
float work[NUMBER_OF_THREADS];
#pragma omp parallel pritvate(iam,neighbor) shared (work,sync)
{
iam = omp_get_thread_num();
sync[iam]=0;
#pragma omp barrier

/*Do computation into my portion of work array*/
work[iam]=compute(iam);

/* Announce that I am done with my work
    * The first flush ensures that my work is
    * made visible before sync.
    * The second flush ensures that sync is made visible.
*/
#pragma omp flush(work)
sync[iam]=1;
#pragma omp flush(sync)

/* Wait for neighbor*/
neighbor = (iam>0?iam : omp_get_num+threads())-1;
while(sync[neighbor]==0){
    #pragma omp flush(sync)
}
/* Read neighbor's values of work array*/
...=work[neighbor];
}
```

Listing 2.10: Flush Directive

Here in the example, a thread writes its respective value to its respective array elements and at the same time neighboring thread tries to read the values that are written by this particular thread. Since, the neighboring threads are trying to read the values that are written by the threads, we need to ensure that the values are written to the shared memory before they are read by the neighboring threads. This is done using the flush directive. The first flush directive ensures that the values are written to the shared memory before they are read by the neighboring threads. The second flush directive ensures that the sync array is made visible to the neighboring threads. The above shown SPMD code is executed by all the threads. From the perspective of a thread the iam variables denotes its thread id, the neighbor repre-

sents the neighboring thread id. The work and sync are the variables that are shared among the threads. The work array is the main array where the threads write their respective values (and also being read by the other threads) and the sync array is used to signal the other threads to say that this thread has finished writing to the work array so that the other thread can read those values. In order to ensure that the values are written to the shared memory before they are read by the neighboring threads, the sync variable has to be flushed out. Note that we can get the thread id as shown in the code using omp_get_thread_num. So at the beginning of the code each thread will get its thread id and then set the sync[id]=0 and then barrier is used to ensure that all the threads have reached this point. Each thread then does some computation and at the end of the computation stores its result to its corresponding work element. Now it requires to announce that it is done with its work so that the neighboring thread can make use of that value. This is done using flush statement so that the changes in work array is visible and then it changes the value of sync array to 1 to signal the other threads that is is done. Then another flush statement so that the changes in sync are visible. Then the neighboring threads will find out its thread id based on their own thread id and will actively look whether the neighboring variables are updated or not. If not then it will keep on checking until the neighboring thread has updated the value (because of the while loop), it keeps flushing the sync variable. Once the sync variable is updated then it reads the value of the work array of the neighboring thread.

2.8.6 Data Scope Attribute Clauses

OpenMP supports many clauses supporting data scope. Data scopes is explicitly specified by data scope attribute clauses.

1. private: A separate variable will be created fore each of the thread and it is initialized to the default value to which any programming language initializes it to(generally garbage values by default in C/C++).

2. firstprivate: If you want a private variable for a thread to be initialized with value of the variable that existed before entering the parallel clause i.e. value in the master thread, it is to be declared as firstprivate. In this, separate private copies will be created for each thread and the value will be initialized to that of a variable in the master thread before entering the parallel region.

3. lastprivate: When you exit a parallel region the value that will be assigned to the variable will be the one that is assigned by the thread that executed at last. In a parallel region, multiple threads will be executing, the value of this private variable will be the one assigned by the thread that executes at last to the variable in the master thread.

4. shared: Declaring the list of variables to be shared among the threads.

5. default: To set the default behavior of the variables in the parallel region. Most variables are shared by default.

6. reduction: If you wish to combine all the private variables in all the threads and reduce it using some operator (say like addition) to get the final value of the reduction to be assigned to a variable in the master thread after exiting the parallel region.

7. copyin: Copyin acts similar to private.

8. copyprivate: It is used to assign thread private variables and if you wish to assign some variables as thread private and initialize them, then we use copyprivate. More will be seen later.

Sometimes while declaring a large multidimensional arrays as private leads to overhead because now each thread would have to create this large multidimensional arrays. Moreover, declaring them as firstprivate will incur even further overhead as each thread now also requires it to initialize the threads which results in overheads.

threadprivate

They are global variable-list where private copies are maintained for each thread (just like private variables) but the variable values will exist across different parallel regions. Say for example say a variable x is declared as threadprivate, then separate copy of x will be maintained for each of the threads. Say now at the end of the first parallel region the value of the variable x in the first thread is say 5. Now in the next parallel region if we declare the variable x as private then the value of 5 will be lost and x will be initialized to some other initial value in the first thread. This would happen if we had not declared the variable x as threadprivate. But since we have declared the variable as threadprivate, the value 5 will exist across multiple parallel regions in the first thread. So for doing this, we declare a variable using a special directive threadprivate instead of private. Consider the following code 2.11.

```
#include<omp.h>
int alpha[10], beta[10], i;
#pragma omp threadprivate(alpha)
main(){
    /* explicitly turn off dynamic threads*/
    omp_set_dynamic(0);
    /*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=%d\n",alpha[3],beta[3]);
}
```

Listing 2.11: threadprivate Directive

Note that we have declared alpha as threadprivate variable. In the first parallel region we have declared i and beta as private variables to each thread. Then in the first parallel region we are assigning value to alpha and beta arrays equal to their array index i (alpha is threadprivate and beta is private). Then in the second region when we print the value of alpha[3] we will get a value of 3 since alpha has been declared as a threadprivate variable and hence will maintain its value across different parallel regions. But when we print beta[3] we will get some garbage value (or may be some initial value if initialized in the second parallel region). This is because we have declared beta as private and not threadprivate. So for beta its scope exists only inside that parallel region whereas for alpha its scope exists across different parallel regions. Hence, if we want the scope of a variable to span across different

parallel regions we have to declare them as threadprivate. Note that here the parallelism should be the same for using threadprivate i.e. say if you have spawned 4 threads in the first parallel region then you must spawn only 4 threads in the next parallel region. Thus, we have used set_dynamic(0) to turn off the dynamic parallelism. If we leave it to OpenMP then it may spawn different number of threads in different parallel regions.

default-example

Consider the following example code for better understanding of concepts of private, shared and default 2.12.

```
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();
            //O.K. - j is declared within parallel region
            a = z[j];
            x=c;
            z[1]=y;
    }
}
```

Listing 2.12: default-Example

Here we have declared variables x, y and array z as global variables. In this code we have declared x as threadprivate variable. We have a function fun in which we have declared a const it c and int i. Then in the clauses while entering the parallel region we have declared default as none which means that we do not want any of the variables to be shared by default. Then we have declared variable a as private to each of the thread and array z as shared among the threads. Inside the parallel region we have declared a variable j, this works since we have declared j locally to each of the threads. Then we do a=z[j], this works since a is listed as private and z is listed as shared. then we do x=c, this should also work since we have declared x as threadprivate and c is const-qualified type. Then we do z[i]=y, this will not work since although z is a shared variable, but variables i and y are not known inside the parallel region since by default we have set none thus, variables won't be shared inside the parallel region. Also, see that we have not declared anything about the variables y and i so they don't exist inside the parallel region (as default is set to none, if the default had been set to shared then the variables would have been shared and thus statement would have worked), thus compiler will flag this as error as cannot reference i or y here.

2.8.7 Library Routines (API)

In addition to the above seen Directive and clauses, OpenMP also supports certain functions like:

- Querying function (number of threads etc)
- General purpose locking routines
- Setting Execution environment (dynamic threads, nested parallelism etc.)

Example of some of the functions are as follows:

- OMP_SET_NUM_THREADS(num_threads): To set the number of threads you wish to spawn in the corresponding parallel region.
- OMP_GET_NUM_THREADS(): To find the number of threads used in the parallel region.
- OMP_GET_MAX_THREADS(): To query the maximum number of available threads you can get for a parallel region.
- OMP_GET_THREAD_NUM(): To get the thread id of a thread inside the parallel region.
- OMP_GET_NUM_PROCS(): To get the number of processors on which OpenMP is executing on.
- OMP_IN_PARALLEL(): To query whether you are in a parallel region.
- OMP_SET_DYNAMIC(dynamic_threads): Set the number of threads to be spawned dynamically (chosen by OpenMP) or a certain value (say maximum) inside the parallel region.
- OMP_GET_DYNAMIC:
- OMP_SET_NESTED(nested): To run on or off nested parallelism in OpenMP
- OMP_SET_NESTED()
- omp_init_lock(omp_lock_t *lock): To initialize a lock
- omp_init_nest_lock(omp_nest_lock_t *lock)
- omp_destroy_lock(omp_lock_t *lock): To free the lock
- omp_destroy_nest_lock(omp_nest_lock_t *lock)
- omp_set_lock(omp_lock_t *lock): For acquiring a lock
- omp_set_nest_lock(omp_nest_lock_t *lock)
- omp_unset_lock(omp_lock_t *lock): For releasing the lock
- omp_unset_nest_lock(omp_nest_lock_t *lock)
- omp_test_lock(omp_lock_t *lock): To see if the lock is available or not
- omp_test_nest_lock(omp_nest_lock_t *lock)

- omp_get_wtime(): To get the time taken in the parallel region by subtracting the time at the start and at the end of the parallel region.
- omp_get_wtick()
- omp_get_thread_num(): Number of threads being used
- omp_get_num_proc(): Number of processors you are currently using

2.8.8 Locks

OpenMP supports both simple locks and nestable locks. For a simple lock, if the thread acquires the lock once then it cannot acquire that lock once again i.e. you cannot use the set_lock function twice for locking on the same variable by the same thread Thus, we cannot use a nest statement for locking simple locks. This will be a deadlock situation since the thread that once who has acquired lock on it and now putting a lock again means it will be waiting for it to be released but since it was the one to acquire it it will be waiting forever to acquire it once again since the lock will never be released. A lock can be released only by the thread that acquires it. So till the thread does not release the lock it cannot acquire it once again. So acquiring the lock twice in a row will lead to a deadlock situation.

In some cases, it may be desired to use a nested lock. For example in case of some Linear Algebra libraries using OpenMP functions and there are say multiple routes from the main function to enter into those functions. Say there are multiple ways of accessing those functions. then it is desired to protect each of the functions and thus use nested locks.

Simple locks are not locked if they are already in a locked state. nestable locks can be locked multiple times by the same thread. Simple locks are available if they are locked. Nestable locks are available if they are unlocked or owned by a calling thread i.e. it uses a counter to count the number of locks on it by different threads and they will be released based on the same manner they were locked by different threads in their calling manner, thus they need to be released by all other calling threads for a thread that wishes to use it.

Consider the following code to show the importance of nested locks 2.13.

```
#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,
    //so need a nestable lock.

    omp_set_nest_lock(&p->lck);
    p->b+=b;
    omp_unset_nest_lock(&p->lck);
}
```

Listing 2.13: Nested lock-Example

In the above code we have declared a structure which has two integer variables a and b and a lock. pair is a variable of struct type. There are two function declared as incr_a and incr_b. Consider that incr_a is called only from incr_pair so there is no need for lock. Now consider the function incr_b which is say called from both incr_pair and elsewhere, thus we need a nestable lock in order to avoid simultaneous updation (or else may lead to garbage values). Thus we have used set and unset lock around it.

Now consider the following continuation of the code as shown in 2.14.

```
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(0;
#pragma omp parallel sections
{
    #pragma omp section
        incr_pair(p,work1(),work2());
    #pragma omp section
        incr_b(p,work3());
}
}
```

Listing 2.14: Nest lock-Example2

In this example note that we are using parallel sections for calling the funtions incr_pair and incr_b, and each of those functions act on p. Now note that incr_a is called only from incr_pair and thus there is no need for lock, but note that incr_b is called from incr_pair as well as directly from incr_b. Thus, it requires a nested lock for locking the value of p while performing the operation and then releasing the lock, which is done by incr_pair and incr_b using set and unset lock functions.

2.8.9 Example 1: Jacobi Solver

This is an example of previously explained grid solver, excpet for the fact that here we are parallelizing across the iterations and not across the rows. Consider the following code as shown in 2.15

```
#include<omp.h>
int main(int argc,char**argv){
...
int rows,cols;
int* grid;
int chunk_size, threads=16;
...
    /*Allocate and initialize the grid */
    grid=malloc (sizeof(int*)*N*N);
```

```
    for(i=0;i<N;i++){
        for(j=0;j<N;j++){
        grid[i*cols+j]=...;
        }
    }

    chunk_size=N/threads;
    #pragma omp parallel for num_threads(16) private(i,j) shared(rows,c
    for(i=1;i<rows-1;i++){
        for(j=1;j<cols-1;j++){
            grid[i*N+j]=1/4*(grid[i*N+j-1\+grid[i*N+j+1]+grid[(i-1)*N+j
        }
    }
}
```

Listing 2.15: jacobi-solver

So as can be seen in the code we have first initialized and then we have used a schedule in parallelizing the code. For that we have declared the chunk_size=N/num_threads which is the total number of rows/columns divided by the number of threads=16. So in the next statement where we start parallel region using pragma omp parallel for indicates that we are using 16 threads, the iterators i and j have been declared as private. The number of rows, columns and grids are shared variables i.e. shared among threads. Then we are using static in the schedule clause so the total number of iterations will be given based on chunk_size to each of the threads in a round robin manner. Note that we are also using collapse(2) which means that it will collapse the two for loops i.e. the total number of iterations will be considered as rows*cols and these many iterations will be distributed among all the threads based on the chunk_size. In this if we had not used collapse then only the first for loop i.e. the iterations for the rows would have been distributed among the threads and now each thread would have to take care of all the iterations for the columns.

2.8.10 Breadth First Search (BFS) Algorithm

The idea behind BFS is that we start from some source vertex at level 0 and then in the first level we explore all the neighbours of the source vertex, then in the second level we pick one of the neighbours of the source vertex and explore its neighbours and so on till all the neighbours have been explored. Below shown is the parallel version of BFS.

Version 1 (Nested Parallelism)

Consider the following example code 2.16

```
...
    level[0] = s;
    curLevel = 0;
    dist[s]=0; dist[v!=s]=-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 2.16: BFS-Example

Here we are at first assigning only the source so the 0th level has only the source vertex s. The current level =0, and distance of the source vertex from the source vertex is set 0 and the distance of other vertex from the source vertex is set to -1. We are suppose to find the distance of the other vertices from the source vertex. Inside the while loop we keep on going until we don't see any nodes in the current level. Using the for loop we explore the vertices at the current level and check all its neighbours and store the neighbours in the variable neigh. Then in the next for loop we start at one of the neighbours and check if the distance is -1 i.e. whether it ha been already explored or not. If it has not been explored then we add that vertex into the next level and add the distance of that node as the distance till previous node +1. Thus, using two pragma omp for we are parallelizing both the for loops nesting the inner and outer for loops, thus a nested for parallelism.

In order to explain in further depth, what we have done is that we spawn a thread corresponding to each of the vertex in the current level which is done by the pragma omp for on the outer loop. Then for each of the vertex we further spawn threads for exploring its neighbours and adding the corresponding distance which can be done independently. Thus we are spawning threads inside a thread i.e. nested level parallelism.

Version 2 (Task constructs)

Here we are going to use task construct to implement the BFS algorithm. Consider the following code 2.17.

```
...
    level[0] = s;
    curLevel = 0;
    dist[s]=0; dist[v!=s]=-1;

    while(level[curLevel]!=NULL){
    #pragma omp parallel ...
        for(v in level[curLevel]){
            for (w in neighbours(v)){
                # pragma omp task...
                {
                    if (dist[w]=-1){
```

```
                                level[curLevel+1]=union(level[curLevel+1],w)
                                ..
                                dist[w]=dist[v]+1;
                            }
                        }
                    }
                }
            }
```

Listing 2.17: BFS-Example

Here we are not using for construct instead we are just opening a parallel region and anytime we are required to process a neighbour then for that neighbour we are dynamically creating a task, the task will be dynamically assigned to OpenMP threads by the OpenMP library.

2.9 Message Passing Interface - MPI

Recall that MPI uses distributed memory parallelism. Here we have explicit communication and synchronization. Thus, unlike in the case of shared memory paralellism where we just used parallel constructs in order to make the program work in parallel, here, we will be required to write an explicit program to be run by each of the processors. Hence, MPI is said to have high programming complexity. MPI is highly popular because even though you may have high multiple cores within a node, for very large problem you need to have many such nodes and when you have many such nodes the only way to share information between parallel processes executing on these two nodes is through message passing and hence we have to inevitably used MPI for it.

2.9.1 MPI Intrdocution

It is a stanard for explicit message passing in MIMD machines. Thus, a MPI program on one machine will also work on any other machines. Thus, it supports portability, and is a standard for hardware vendors. Currently the latest version is MPI-3 in 2012. MPI containes the following standards:

- Point-Point: point to point communications. When we are required to send information from one process to another process using a send and receive function.
- Collectives: When we wish to communicate across multiple processes, they are done using this collectives.
- Communication contests: For each communication it defines a scope.
- Process topologies: MPI supports process topologies. Recall that we have already seen network topologies. Network topologies refer to how the processes are connected as a mesh, bus, star, etc.. All these happen at the hardware level. Likewise at the programming level the processes may have certain communication pattern. Example, may be in a linear manner or in a mesh like manner. This is called Process topologies.
- Profiling interface: MPI allows profiling interface that is to implement MPI functions so that you can create a library of your own.
- I/O: This is defined in MPI standard 2. How can we make multiples processes read and write data simultaneously.
- Dynamic Process groups: MPI -2 allows you to dynamically spawn multiple processes during the execution.
- One-sided communication: Like in Shared memory parallelism where only one process is required for communication. Here, in MPI 2 using one-sided communication can be used where one of the process just adds data and there is no other process required to receive that data. Thus, instead of requiring two processes to share data we can have only one process which can put data into a remote process memory without requiring the other process to receive it.
- Extended collectives:

There are many such MPI functions which can be used to write a program.

2.9.2 Communication Primitives

Point-Point Communications

```
MPI_SEND(buf, count, datatype, dest, tag, comm)
MPI_RECV(buf, count, datatype, source, tag, comm, status)
MPI_GET_COUNT(status, datatype, count)
```

Here, sending data from one process to another is called MPI_SEND. The sending process will send an array, along with the count of number of elements in that array, and the data type of the array elements. The buf, count, datatype corresponds to MPI message i.e. the sender process that wants to send message is defined by these three parameters to another process called the reciever process.

- dest: To identify the receiver process the sender process should give the id of the receiver process. In MPI terminology. we call these ids as Rank of a process. When we spawn a set of process, each process will get a rank starting from 0 to p-1, where p is the number of processes. For example, if process 0 wishes to send a message to process 4, the it will fill 3 in the dest.
- tag: Just like the identifies for the process, it also gives identifier for the message. This is especially useful when a sender process wants to send multiple messages to the same reciever process and we want the receiver process to demarcate the messages clearly, then we need to use message id.
- comm: It shares the communication context. It is a communicator pointer available to the process that calls this function the pointer points to a communication group

Similarly, for MPI_RECV:

- buf: When the message is send it will be received in this as the receiving buffer.
- count: This will count the number of elements in the array that is supposed to be receive.
- datatype: The datatype of the received data.
- source: Just like sender has to say for which process it is sending to, the receiver has to mention from which process it is receiving from which is by giving the rank of the process it is receiving from.
- tag: It also needs the same tag as that of the sender process so that the message is matched with the receiver process
- comm: Then a communicator.
- status: It is an extra parameter to find out the status of the message. For this we can use MPI_GET_COUNT(status, datatype,count) to know how much of the data you ended up receiving.

When you initially start a process all the processes will belong to a certain group and all the processes will be given a rank with respect o processes in the group. In MPI you can create subgroups (may or may not be overlapping). Inside a subgroup the process can have

different ranks with respect to other processes in the sub group than the ones originally assigned. For example, withing a group say a process is ranked 4, then in a subgroup consisting of that processor it can be ranked 0. Thus, the process had id 4 with respect to overall group but now has rank 0 with respect to its subgroup. Thus a particular process can have multiple ranks depending upon the subgroup you are considering it in. Thus, say that sending to rank r is now confusing since there could be multiple processes with rank r within different subgroups, thus, we need to also mention the subgroup, only then the process you are sending to it will be clear. Thus, we not only need to tell the rank but also the communication scope/communication group with which you are referring . That communication group is pointed to by a variable called the communicator.

Consider the following simple example code

```
comm=MPI_COMM_WORLD
rank=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 array a*/

/*do reverse communication*/
```

Let us assume only two processes in the overall world with no subgroups. The overall group in MPI is called by a default value MPI_COMM_WORLD. MPI_Comm_rank is a special utility function to find out the rank with respect to the particular group that you are belonging to. The above shown is a SPMD program. Thus when the program is executed by the process it will get its own id/rank in the overall group stored in the variable rank. MPI_INT are called wild cards which is used to define integers, similarly, MPI_DOUBLE stands for double and so on. Now in thcode we initialise an array and then the process whose rank=0 send the second half of the array to the process with rank=1. See that we we are sending A=n/2 in the buff and then n/2 elements of the array whose datatype is integer. We can use some integer value (say 5) as tag/id of the message, then the comm refers to the process 1 in the overall communicator defined in the MPI_COMM_WORLD. In the else loop, the process whose rank is not zero i.e. rank 1 will do MPI_RECV and will receive the message into its array and will receive n/2 elements of the array of data type int (MPI_INT) from process 0, with the same tag and comm, and the status stored in the variable by &status. After that the processes work on their individual arrays in parallel.

The default communicator is MPI_COMM_WORLD which includes all processes We can also use Wild cards in the MPI_RECV function in the receiver source and tag fields as MPI_ANY_SOURCE and MPI_ANY_TAG respectively. This is extremely useful when a particular process is receiving messages, using this, it does not need to worry about any particular order in which it should receive and it could receive in any order if we use MPI_ANY_SOURCE. Likewise MPI_ANY_TAG can be used when a particular process is receiving multiple message from multiple sources with multiple message ids and the if the order of messages does not matter, then, in that case we can specify wild card MPI_ANY_TAG. Recall that we can also use MPI_COMM_RANK to get the process rank/id.

Following are a few Utility functions in MPI:

- MPI_INIT: This is used to initialize the MPI environment at the beginning of the program.
- MPI_FINALIZE: This is used to finalize the MPI environment at the end of the program.
- MPI_Comm_size(comm,&size): With the function a process can find out how many processes are involved in the MPI communicator world that you enter (here comm).
- MPI_Comm_rank: A process can query its own id.
- MPI_Wtime(): It is used to get the time spent in the parallel region of the code by inserting before and after the section and then subtracting to get the time taken in that section of the code.

EXAMPLE 1: Finding maximum using 2 processes

Consider the following C program.

```
#include "mpi.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 prodcesses.\n");
        exit(0);
    }
    if(rank==0){
        /*Read N from console*/
        MPI_Send(&N,1,MPI_INT,1,5,comm);
        /*Do dynamic allocation of A array with N elements*/
        /*Initialize an array A of N elements*/
        /*Do dynamic allocation of local_array with N/2 elements */
        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);
        /*Do dynamic allocation of local_array with N/w elements */
        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);
        if(rank1_max>max){
            max=rank1_max;
        }
    }
    printf("Maximum number is %d\n",max);

    MPI_FINALIZE();
}
```

This program tries to find out maximum element in an array using two processes. Note that we need to include mpi.h header file to use MPI. Then we declared a communicator comm of type MPI_Comm. Then another variable status of type MPI_Status which will be used in receive function. We first initialize the MPI environment using MPI_Init and pass the arguments &argc,&argv. Then we use the default communicator MPI_COMM_WORLD and store it in comm variable. Then using MPI_Comm_size we can find the total number of processes with which this program is started and then find its own ran using MPI_Comm_rank. Since the program is required to use only two processes we put a check on the number of processes with which user starts using if loop. If the size is not equal to it prints the statement or else we continue with the algorithm. If rank =0 for the process, it will read N from the console and then sends that N to the Processor with rank 1. This is done using the MPI_Send command with sending the address using &N and since its only 1 value we write 1 in place of count, since the datatype of N is an integer we write MPI_INT. If the process is not rank=0 i.e. rank=1, it will receive similarly using MPI_Recv as shown in the code above. Then the process 0 dynamically allocates the array A with N elements and initializes it. The process 0 then reads the first half of the array and stores it in its local array. It then sends the second half using MPI_Send to the process with rank=1. Note that in this case we used message id as 10. Then the process with rank=1 receives it into its local array. Then after storing the value of large negative number in local_max, we use a for loop which is executed by both the processes to find the maximum in their own part of array. This is a case of SPMD program where the same program is executed by multiple processes on different data. Both then find the local maximum and then the process with

rank=1 sends its maximum to process with rank 0 and after process with rank 0 compares both the maximum and chooses the larger one and prints the maximum number. Then we use MPI_Finalize to end the MPI environment.

2.9.3 Buffering and Safety

Blocking Communications

One of the important aspects related to send and receive is send and receive buffers. In order to understand this better consider an example. Say you have a written a program and in it you call MPI_Send(user_buf,...) where user_buf refers to the buffer (data address) to be send. This user program is linked with a MPI library. So when user program uses MPI_Send it calls the MPI library which implements the MPI functions and interacts with the underlying hardware and network using TCPIP protocols (Tcp_send()) in the networks functionality, which will then be received by some receiver. At the receiver side there is a MPI library which has implemented MPI_Recv using the underlying network receive. This is then called by the receiving process in the user program by invoking the function MPI_Recv. The MPI libraries will have its own global send and receive buffer, and at some point in the MPI library implementation of MPI_Send the user_buf gets copied into the send_buf and likewise after the message is received in the MPI library inside the MPI_Recv function it gets copied from recv_buf to user_buf. MPI_Send also spawns threads and then return. This threads take care of any additional overhead like inserting message headers, dealing with acknowledgments and invoking the network send function TCP_send(). The user program does not have to wait for all of these since inside the MPI library function it spawns threads to take care of the sending and then immediately returns because the main thread which called this function can immediately return and make progress with the subsequent statements. Behind the scenes, MPI library threads is still taking care of sending the communication. Likewise is the case with MPI recv_buffer. This is called blocking send and blocking receive because the MPI_Send has to wait till the user_buf gets copied into send_buf. Likewise MPI_Recv has to block or wait until the message from recv_buf is safely copied to user_buf passed by the receiving function.

Consider the following three scenarios. Case 1: Process 0 does MPI_Send and then MPI_Recv and the process 1 does the matching receive MPI_Recv and then a matching send MPI_Send for the receive in process 0. This kind of program always works since it has simply the send corresponding to a receive and a receive corresponding to a send.

Case 2: Now consider a scenario where process 0 does MPI_Recv and MPI_Send and process 1 does MPI_Recv and MPI_Send. Now the process 0 can make progress only after it has received from process 1 and similarly process 1 can proceed only after it receives from process 0. Thus, unless both the processes do not receive they cannot proceed to their next statement of MPI_Send which is the corresponding statement for the other process. So both of them cannot proceed and will be waiting for each other. Hence, this situation will result in a deadlock.

Case 3: Now consider a situation where Process 0 first calls MPI_Send and then MPI_Recv and the process 1 does MPI_Send and then MPI_Recv. This will work because the process 0 user buffer that is passed in the MPI_Send will be safely copied to MPI implementations send_buf and will return to the user program. While the communication is happening in the background threads, the process 0 will call MPI_Recv function. At the same time process 1 calls MPI_Send and the corresponding MPI_Recv function in process 0 will be able

to receive it. Hence, there is a progress in the program as in the process 0 we will safely return from MPI_Send operation since the user_buf was safely copied to send_buf and the function returns and continue on to the next statement in the user program to receive from process 1. Likewise process 1 will be able to execute MPI_Recv statement. This may or may not work provided that the MPI libraries send and receive buffers have got sufficient space to accommodate the message that is being passed in the userbuf. This is because send_buf is an internal buffer that is maintained by the MPI library and it cannot allocate really large size buffers because if it allocates large size buffers a significant amount of memory will be consumed by the MPI library and there will be very little memory left for the user program for its own variables. So MPI implementation have typically fixed size buffers. If the fixed size buffer is of adequate capacity greater than message size being sent using MPI_Send then it will work as described and user_buf will be safely copied to send_buf and it will then safely proceed. On the other hand, if the implementation buffer send_buf is smaller in size than the user_buf passed in MPI send then all the part of the user_buf will not copied and only some part of it will be copied and thus in order to send all of the message, the copied part needs to be send first and received by the receiving process and then when the buffer is empty and the entire message is again copied and this process goes on. But note that in order for the buffer to copy the entire message it needs to first send the already copied part to the required processor which then needs to be received by the receiving end processor. If the receiving end processor is also trying to send as in this case then the situation will lead to a deadlock situation, as both the processes are not waiting for the other process to start receiving so that the rest of the user_buf can be copied to the send_buf. Overall, if the implementation buffer is of adequate capacity then this work or else if it is of a lesser capacity than the user_buf then it will result in a deadlock situation. Note that this is not the case with case 2 because here the implementation will require the MPI_Recv to first receive all of the message before moving on to the next statement. Thus case3 may or may not work and hence is an unsafe way of writing program so avoid writing those kinds of programs.

These are called Blocking functions.

Non-Blocking Communications

MPI also supports non-blocking functions. Here the process does not even wait for the message to be copied to the send buffer or receive buffer. Within these functions immediately the thread is spawned and the control returns to the user program even before the message is safely copied to MPI buffers (send or receive). This means that the communication may not have reached a safe point in both send and receive when you use a non-blocking function. Thus, the sender or receiver after sending/receiving should not start modifying/using this buffer in the program as there is no guarantee that the communication is complete. As the threads spawned are taking care of the communication. Thus, after calling send/receive you cannot use the send/receive buffers. Recall that on the contrary in case of blocking functions, a thread is spawned only after the message is safely copied to the send buffer and then the control returns to the user program and in case of blocking receive whether you spawn a thread or not only when a message is actually received to the user buffer the control returns to the user program and the MPI receive returns until then the MPI_RECV will block. In case of non-blocking functions MPI_ISEND and MPI_IRECV as soon as they are called immediately the threads are spawned in both the cases and this thread will take care of all the things that the blocking send and receive

had implemented in the background and the control will immediately returns the control the user program. Here, the return would be very quick. But, because of this quick return the communication might still be happening in the background by means of those threads. Thus the user_buf may still not have been copied to send_buf. Likewise the recv_buf may still not have been copied to user_buf on the receiver side. Hence, even though the function returns the programmer cannot assume that this buffers are safe to use and hence should not be using those buffers. The following are the non-blocking functions:

- MPI_ISEND(buf,count,datatype,dest,tag,comm,request)
- MPI_IRECV(buf,count,datatype,dest,tag,comm,request)
- MPI_WAIT(request,status)
- MPI_TEST(request,flag,status)
- MPI_REQUEST_FREE(request)

We get a token called request when we use MPI_ISEND and MPI_IRECV and with the help of this token the programmer can complete the process using MPI_WAIT function. This is helpful because between the MPI_ISEND/MPI_IRECV and MPI_WAIT function the user can do some other processes which are not dependent on this communication. MPI non-blocking communications allow you to use the principle of orchestration that is overlapping computation and communication. The MPI_ISEND and MPI_IRECV is also called posting send and posting receive respectively and MPI_WAIT is called completion function.So between positing and completion the user can have some other statements which do not depend on these communication and thereby overlap the communication statements with computations. MPI_TEST function is used to check whether the communication is completed or not and if it is completed then we use MPI_WAIT to complete.

A post-send returns before the message is copied out of the send buffer. A post-recv returns before data is copied into the recv buffer. Efficiency depends on implementation. Other non-blocking communications:

- 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)

A particular process can post may non-blocking functions and thus we can have an array of requests as shown by the above non-blocking functions and the above functions as the name suggest can be used to check or test accordingly. For example to check whether any one of them is completed we use test for any. To check if all has complete we use testforall and so on as the names suggest. Thus, one can use these variants of test and variants of wait when many non-blocking functions are used.

Now consider two cases:

Case 1: When process 0 does MPI_Send(1) and then does MPI_Send(2) and process 1 does MPI_Irecv(2) and then MPI_IRecv(1). Then in this case this would not be a problem since the process 1 calls non-blocking functions and hence MPI_IRecv(1) means that it will not wait for the message to be copied and will move on to the next statement MPI_IRecv(2) and thus the send and receive will match and the program will continue executing. Even though the process 0 uses blocking send, irrespective of the size of the buffer the program will continue working since the corresponding receive and send are matched.

Case 2: Process 0 does MPI_ISEND and then MPI_Recv and process 1 does MPI_ISEND and then MPI_Recv. Then in this case, the program works and is safe to write because the send functions are non-blocking functions and hence both the processes will post receiving functions. Thus, we can convert any communication to safe using non-blocking functions.

2.9.4 Example: Finding a Particular element in an Array

Consider the following example, given an array of elements within a program and the user wants to search for a particular element in this array like a database search. Assuming that the array elements are unique and only two processes are involved. So the idea is we divide the array into two parts and send the second half to the process 1 and we also send the key element to search for to the process 1. Both the processes then start searching for the element and if found immediately terminate the program. The following is SPMD program.

```
#include "mpi.h"
int main(int argc,char** argv){
...
...
int other_i,other_rank,flag;
MPI_Request request;
...
...
    ...
    ...
    other_rank=(rank==0)?1:0
    MPI_Irecv(&other_i,1,MPI_INT,other_rank,10,comm,&request);
    for(i=0;i<N/2;i++)
    {
        if(A[i]==elem){
        /*Found it!Inform the other process of the index position*/
            MPI_Send(&i,1,MPI_INT,other_rank,10,comm);
            if(rank==0){
                global_pos=i;
            }
            /*Cancel the posted Irecv by this process*/
            MPI_Cancel(&request);
            break;
        }
        else{
            MPI_Test(&request,&flag,&status);
            if(flag==1){
```

```
                    /* The other process has found it*/
                    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();
}
```

Here we first find the rank of the individual process and store the rank of the other process in other_rank variable. Then we post a non-blocking MPI_Irecv. Now each of the process checks in its part of the array for the element. If it's found then we inform the other process of the index position using blocking MPI_Send instead of searching in the rest of the array and terminate the program. Since there is no guarantee that the when and by what process will that element will be found and by whom we use non-blocking receive and hence it's very beneficial in this scenario. For example, here if a process finds the element in the array then it immediately sends the index position to the other process, but if it is not found then it may never send the position. Thus, a particular process may or may not send the message depending on whether it was able to find the element or not. Hence, in this kinds of non-deterministic cases it will be highly useful to make use of non-blocking communication than blocking communications. In case, we had used blocking communications instead of non-blocking communication we would be required to communicate with the other process whether found or not after searching through each element which will be a waste of communications. This is because the other process will be positing blocking receive and hence it needs to be communicated in each and every step whether it was found or not in every iteration. We want a process to send only when the element was found and not every time even when element was not found.

For the process that found the element will cancel its request of Irecv using MPI_Cancel. For the other process that did not find it will Test whether the communication has been made using MPI_Test and if the communication is made, it will wait using MPI_WAIT for the communication to be completed and then after setting the correct global position it will break. Note that in the case we wanted to use blocking MPI_Recv then we would be required to move the call to the end of the iteration since it will b anticipating a communication every time and hence need to communication after each iteration whether the element was found or not after each iteration which is a waste of communication resources. Note that here we are not doing busy wait as the processes continue searching and not wait for the other process to complete.

2.9.5 Communication Modes

There are different communication modes in MPI. The send and receive that we have seen till now correspond to standard communication i.e. send can be posted before or after the receive is executed at the receiver side. The communication will be completed from the

Mode	Start	Completion
Standard(MPI_Send)	Before or after recv	Before recv (buffer) or after (no buffer)
Buffered (MPI_Bsend)(Uses MPI_Buffer_Attach)	Before or after recv	Before recv
Synchronous(MPI_Ssend)	Before or after recv	Particular point in recv
Ready(MPI_Rsend)	After recv	After recv

Table 2.5: Communication Modes

sender side before the receiver is executed at the receiver side if there is adequate buffer capacity. In case the there is not adequate buffer then the send will return only after the receiver side has started receiving side and has started consuming the message.

MPI also supports buffered send, in this you can create an array of adequate capacity and pass it to MPI library to use it as a buffer. In this case there will be no problem of buffering and safety as you are creating adequate size of array to pass it to the MPI library to use it as buffer. Here, we create the large enough array and pass it to the MPI using buffer attach function and MPI will then use this array for subsequent send and receive as implementation buffer. In this case we call Buffered_Send. Thus, in this case send can start before or after receive and be completed even before the receive is executed.

Synchronous send, can be executed before or after the receive is executed on the receiver side but it will complete only when the receiver has started consuming some of the message. This is particularly useful when you want to make sure that the receiver has reached that particular point in the program. Thus, when the synchronous send returns, you will know that the receiver process has started consuming the message on the receiver side.

Read send or Rsend: Here, send can start executing only after receive is executed. If your program reaches send first and the receive has not been executed then the MPI will throw an error. This is especially useful for large message sizes where we want to send a huge message and we know that the implementation buffer will not have that capacity then you want this buffer to directly be communicated with the receiver side and you want the receiver to start receiving the message. It is in this case we use ready send.

The summary of all the Communication Modes is as shown in the table 2.5.

Bibliography

Randal E Bryant and David R O'Hallaron. Computer systems, 2016.

V Rajaraman. Ieee standard for floating point numbers. *Resonance*, 21:11–30, 2016.