

Course: Parallel Computing

Course Code: BCS702

Credits: 03& 2022 Scheme

Module 1: Introduction to Parallel Computing

Module – 01

Chapter – 01

Introduction to parallel programming

Why do we care? Aren't single-processor systems fast enough?

Slowing Performance Improvement:

We care because the rate of performance improvement for single processors has slowed dramatically. From 1986 to 2003, performance increased by over 50% per year, but from 2015 to 2017, it increased by less than 4% per year.

Stark Difference in Growth:

This change is significant. At 50% per year, performance grows by a factor of 60 in 10 years, while at 4%, it only grows by a factor of 1.5. This means waiting for the next generation of microprocessors is no longer an effective strategy for performance gains.

Need for Increased Performance:

While some applications may be fast enough, many others (implied by the move to parallel systems) require greater performance that single processors can no longer provide at the historical rate of improvement.

Why can't microprocessor manufacturers continue to develop much faster single-processor systems?

Why build parallel systems?

Physical Limits:

The text implies that manufacturers faced limitations in their ability to continue developing faster, single-processor systems. By 2005, they determined that the path to rapid performance increases lay in a different direction.

Shift to Parallelism:

The solution was to abandon the effort to create "ever-faster monolithic processors" and instead focus on building parallel systems.

Multiple Processors:

The new design approach was to "put multiple complete processors on a single integrated circuit" to achieve the desired performance gains.

Why can't we write programs that will automatically convert serial programs into parallel programs?

Lack of Awareness:

The fundamental issue is that programs written for a single processor are "unaware of the existence of multiple processors."

No Automatic Benefit:

Simply running a serial program on a system with multiple processors does not automatically improve its performance. The performance will be "effectively the same as its performance on a single processor."

Incompatible Design:

The text highlights that parallel execution is not a "magic" addition. The program's design itself must be written to take advantage of the multiple processors for any performance benefit to be realized.

Why we need ever-increasing performance

Driving Major Advances: Increased computational power has been the core reason behind significant advances in various fields, including science (e.g., decoding the human genome, accurate medical imaging), the internet (e.g., fast web searches), and entertainment (e.g., realistic computer games).

Expanding the Scope of Solvable Problems: As our computational power grows, it allows us to tackle a greater number of complex problems that were previously out of reach.

The text provides several examples of these problems:

 **Climate Modeling:** To better understand climate change, we need highly accurate computer models that can simulate the complex interactions between the atmosphere, oceans, solid land, and ice caps.

 **Protein Folding:** Misfolded proteins are believed to be linked to diseases like Huntington's, Parkinson's, and Alzheimer's, but current computational power limits our ability to study their configurations.

 **Drug Discovery:** Increased computational power is needed to analyze genomes extensively to devise personalized treatments for individuals who do not respond to existing drugs.

🚀 **Energy Research:** Greater computational power will enable the creation of more detailed models of clean energy technologies like wind turbines, solar cells, and batteries, which could lead to far more efficient designs.

🚀 **Data Analysis:** A tremendous amount of data is being generated (doubling every two years), and it is largely useless without extensive analysis. This includes data from genomics, particle colliders, medical imaging, and astronomical research.

Why we are building parallel systems

Heat and Power Limitations:

The historical increase in single-processor speed came from making transistors smaller and faster. However, faster transistors consume more power, which is dissipated as heat. By the early 21st century, air-cooled integrated circuits reached the limit of their ability to dissipate this heat, making it impossible to continue increasing their speed.

Slowdown in Transistor Density:

The increase in transistor density, a key driver of performance, has also slowed dramatically in recent years, further limiting the potential for single-processor improvements.

A "Moral Imperative":

The text argues that there is a need to continue increasing computational power because of its potential to drive significant advances and improve our existence.

The Solution is Parallelism:

Since building faster single processors is no longer a viable path, the industry has turned to parallelism as the solution. Instead of building one faster, complex processor, manufacturers now put multiple, simpler, complete processors (known as cores) on a single chip.

Why we need to write parallel programs

Based on the text, we need to write parallel programs because traditional serial programs cannot effectively utilize modern multi-core systems, and automatic conversion from serial to parallel is not a viable solution.

The core reason is that the most efficient parallel algorithms are often fundamentally different from their serial counterparts and are unlikely to be "discovered" by a simple translation program. The text illustrates this with an example of summing a large number of values.

The Problem: Summing Values

1. Serial Program A serial program to sum n values would run on a single core and would look like this:

```
sum = 0;
for (i = 0; i < n; i++) {
    x = Compute_next_value(. . .);
    sum += x;
}
```

2. A Simple Parallel Approach (Master/Slave) A basic parallel version would divide the work among p cores, with each core calculating a "partial sum" of its assigned values.

```
my_sum = 0;
my_first_i = . . . ;
my_last_i =      ;
for (my_i = my_first_i; my_i < my_last_i; my_i++) {
    my_x = Compute_next_value(. . .);
    my_sum += my_x;
}
```

After computing their my_sum, all cores send their results to a designated "master" core (e.g., Core 0) which then adds them all together:

```
if (I'm the master core) {
    sum = my_sum;
    for each core other than myself {
        receive value from core;
        sum += value;
    }
} else {
    send my_sum to the master;
}
```

This method is a simple generalization of the serial process, but it creates a **bottleneck** because the master core does all the final work. For 1000 cores, the master would have to perform 999 receives and adds.

A Better Parallel Approach (Tree-based Summation)

A far more efficient parallel algorithm for this problem is one that does not resemble the serial approach.

Instead of a single master, cores are paired up to sum their values. This process is repeated, with the number of active cores being halved at each step, until only one core holds the final sum.

The text describes this as follows: "Instead of making the master core do all the work...we can pair the cores so that while core 0 adds in the result of core 1, core 2 can add in the result of core 3, core 4 can add in the result of core 5, and so on."

This method is illustrated by the provided diagram:

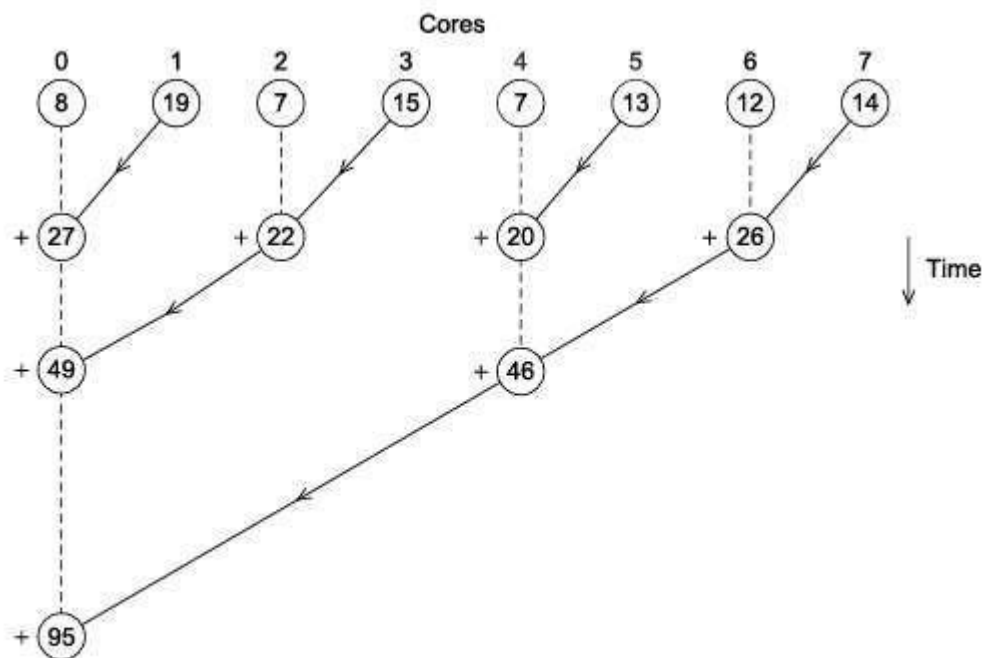


FIGURE 1.1

Multiple cores forming a global sum.

FIGURE 1.1 Multiple cores forming a global sum. (Image description: A diagram showing 8 cores (0-7). In the first step, core 0 receives from core 1, core 2 from core 3, etc. In the next step, core 0 receives from core 2, and core 4 from core 6.

In the final step, core 0 receives from core 4. The final result is in core 0. This shows a tree-like structure of communication and summation.)

This second method is dramatically more efficient. With 8 cores, the master core only performs 3 receives and adds (compared to 7).

With 1000 cores, the difference is even more stark: the master core would only need to perform 10 receives and adds, an improvement of almost a factor of 100 over the first method.

How do we write parallel programs?

Based on the text provided, we write parallel programs by following the basic idea of **partitioning the work** among the available cores. There are two primary approaches to this partitioning:

1. Task-Parallelism

In this approach, you divide the problem into different tasks, and each core executes a different task. The cores may be performing different operations.

- **Example:** A professor and her TAs are grading an exam with five questions. To use task-parallelism, they could each be assigned to grade all 100 student responses for a single question (e.g., the professor grades all responses for Question 1, a TA grades all responses for Question 2, and so on). Each person is performing a different grading task.

2. Data-Parallelism

In this approach, you partition the data used in the problem among the cores, and each core performs more or less the same operation on its assigned piece of the data.

- **Example:** For the same exam grading scenario, data-parallelism would involve dividing the 100 exams into five piles of 20 exams each. The professor and each of her TAs would then grade all questions on every exam in their respective pile. Each person is performing the same task (grading a full exam) on a different set of data.

Coordination of Cores

Writing a parallel program becomes much more complex when the cores cannot work independently and need to coordinate their efforts. This coordination involves three key issues:

- **Communication:** Cores must be able to send data to each other. For example, in the global sum problem, cores send their partial sums to a master core to be combined.

- **Load Balancing:** The work must be divided evenly among the cores. If one core is assigned significantly more work than others, its computational power will be wasted as it sits idle, waiting for the heavily loaded core to finish.
- **Synchronization:** Cores must be able to wait for each other at specific points in the program. This prevents some cores from starting work on data that has not yet been prepared by another core (e.g., waiting for a master core to read in all the data from a file before starting calculations).

In practice, the most powerful parallel programs are written using **explicit parallel constructs** (e.g., extensions to C, C++, and Java).

These programs contain direct instructions that tell each core what task to execute and when to coordinate with other cores. However, this level of explicit control makes the programs extremely complex to write.

What we'll be doing

Based on the text provided, this section outlines the approach to learning explicit parallel programming by focusing on a few key APIs that correspond to different types of parallel systems.

The Tools: Four APIs

The purpose is to learn how to write parallel programs using the C language with four specific APIs:

- **Message-Passing Interface (MPI):** A library of functions for message-passing between cores.
- **POSIX Threads (Pthreads):** A library for managing threads (cores) in shared-memory systems.
- **OpenMP:** A library and compiler modifications for parallelism.
- **CUDA:** A library and compiler modifications for programming Nvidia GPUs.

Why Four APIs?

The need for multiple APIs stems from the different ways parallel systems are designed. These systems are typically classified in two ways: by how they handle memory and by how they handle instructions.

1. Memory Classification

- **Shared-Memory Systems:** All cores can share access to the computer's memory. Cores coordinate by examining and updating these shared memory locations.
- **Distributed-Memory Systems:** Each core has its own private memory, and cores communicate by explicitly sending and receiving messages to each other.

This classification is illustrated in the provided diagram:

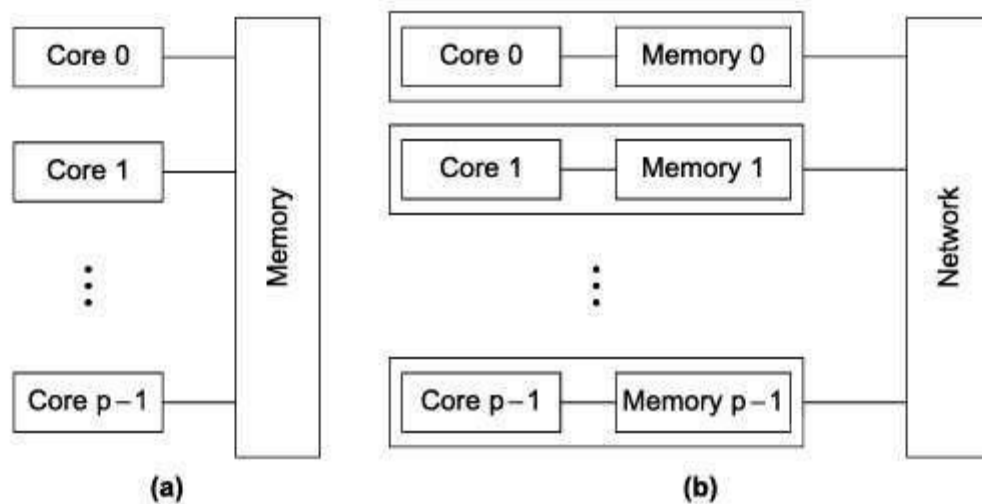


FIGURE 1.2

(a) A shared memory system and (b) a distributed memory system.

FIGURE 1.2 (a) A shared memory system and (b) a distributed memory system. (Image Description: (a) A schematic showing multiple cores connected to a single, shared memory block. (b) A schematic showing multiple cores, each with its own memory block, connected by a network.)

2. Instruction/Data Stream Classification

- **Multiple-Instruction, Multiple-Data (MIMD) Systems:** Each core has its own control unit and can execute its own stream of instructions on its own data, independently of other cores. This is how conventional processors work.
- **Single-Instruction, Multiple-Data (SIMD) Systems:** Cores share a single control unit, so they all execute the exact same instruction at the same time, but each on its own separate data. This is the fundamental principle behind modern Graphics Processing Units (GPUs).

APIs and Their Corresponding System Types

Each of the APIs is designed to program a specific type of parallel system:

- **MPI:** Used for **distributed-memory MIMD** systems.
- **Pthreads:** Used for **shared-memory MIMD** systems.
- **OpenMP:** Used for **shared-memory MIMD** and **shared-memory SIMD** systems.
- **CUDA:** Used for programming Nvidia GPUs, which exhibit aspects of all four classifications (shared memory, distributed memory, SIMD, and MIMD). The focus, however, will be on the shared memory SIMD and MIMD aspects.

Concurrent, parallel, distributed

Concurrent Computing:

A program where multiple tasks can be in progress at the same time. This is a broad term. For example, a multitasking operating system running on a single-core machine is concurrent, as it quickly switches between tasks, giving the appearance that they are all in progress simultaneously. Both parallel and distributed programs are considered concurrent.

Parallel Computing:

A program in which multiple tasks cooperate **closely** to solve a single problem. These tasks usually run **simultaneously** on cores that are physically close to each other and are connected by a very high-speed network or share the same memory. The text's previous examples of the two concurrent addition programs are considered parallel.

Distributed Computing:

A program that may need to cooperate with other programs to solve a problem. These programs are often "loosely coupled" and can run on computers that are physically separated by large distances. The programs themselves may have been created independently. A Web search program is provided as an example of distributed computing.

A word of warning

Increased Complexity: Writing parallel programs is almost always more complex than writing serial programs because it requires coordinating the actions of multiple cores.

Avoid "Seat-of-Your-Pants" Programming: It is a mistake to write parallel programs without careful design.

Design is Crucial: All the rules for careful program design and development are even more important when writing parallel programs than they are for serial programs.

Typographical conventions

Program Text: Code, whether displayed in a block or within a sentence, will use a specific typeface, like the following example: `printf("hello, world\n");`

Definitions: When a term is being defined, it will be displayed in **boldface** type. For instance, "A parallel program can make use of multiple cores."

UNIX Shell Commands: Commands intended to be typed at a shell prompt will be preceded by a dollar sign (\$), for example: \$ gcc -g -Wall -o hello hello.c

Function Syntax: The syntax of a function call will be specified by providing a sample argument list, such as `int abs( int x);`.

General Syntax Notation:

- Angle brackets (<>) are used to enclose **required** content.
- Square brackets ([]) are used to enclose **optional** content.
- For example, the syntax `if ( <expression> ) <statement> [ else <statement> ]` indicates that the expression and the first statement are required, while the else clause and its statement are optional.

Parallel hardware and parallel software

Classifications of parallel computers

1. Flynn's Taxonomy

This classification is based on the number of instruction streams and data streams a computer can manage simultaneously.

- **SISD (Single Instruction, Single Data):** A classical von Neumann system. It executes one instruction at a time and computes a single data value at a time.
- **SIMD (Single Instruction, Multiple Data):** These systems manage multiple data streams but support only a single instruction stream. All cores execute the same instruction on their own data.
- **MIMD (Multiple Instruction, Multiple Data):** These systems support both multiple instruction streams and multiple data streams. Each core can execute its own set of instructions independently.

2. Memory Access

This classification is based on how the cores access the computer's memory.

- **Shared Memory Systems:** In these systems, all cores can share access to the same memory locations. They coordinate their work by modifying these shared memory locations.
- **Distributed Memory Systems:** In these systems, each core has its own private memory. Cores coordinate their work by communicating with each other across a network.

SIMD Systems

Single Instruction, Multiple Data (SIMD) systems are a type of parallel computer that operates on multiple data streams using a single instruction stream. In this model, a central control unit broadcasts an instruction to multiple datapaths. Each datapath then either executes the instruction on its own data or remains idle.

- **Key Principle:** SIMD systems are ideal for problems with "data-parallelism," where the same operation is applied to a large array of data. This allows for very efficient parallelization of simple loops.
- **Synchronous Operation:** In a classical SIMD system, all datapaths operate synchronously and must wait for the next broadcasted instruction. They have no internal instruction storage.
- **History:** SIMD systems had a checkered history, but their principles have re-emerged in modern vector processors and GPUs.

Vector Processors

Vector processors are a type of SIMD system designed to operate on arrays of data (vectors) rather than individual data elements (scalars).

- **Vector Registers:** These are specialized registers that can store and operate on a fixed-length array of operands simultaneously.
- **Vector Instructions:** A single instruction can perform an operation on an entire vector. For example, a single instruction could replace a for loop that adds two arrays, which is a significant efficiency gain.
- **Interleaved Memory:** The memory system is structured into multiple banks, allowing for faster, simultaneous access to elements of a vector, as they are distributed across different banks.
- **Specialized Hardware:** Vector processors include special hardware to accelerate **strided memory access** (accessing elements at fixed intervals) and **scatter/gather** operations (accessing elements at irregular intervals).
- **Strengths and Weaknesses:** They are very fast and easy to use for applications that can be "vectorized." However, they do not handle irregular data structures well and face scalability limits in their vector length.

Graphics Processing Units (GPUs)

GPUs are powerful processors that leverage SIMD principles for real-time graphics and general-purpose computing.

- **SIMD Foundation:** GPUs use a graphics pipeline where programmable "shader functions" are applied to thousands of elements (like vertices or pixels). Since the same operation is applied to many elements, GPUs optimize performance by using SIMD parallelism with a large number of datapaths on each core.
- **Hardware Multithreading:** Due to the massive amount of data they process, GPUs use extensive hardware multithreading to manage multiple tasks and avoid memory stalls.
- **Not Purely SIMD:** Although they use SIMD parallelism, modern GPUs are not purely SIMD systems. They can also execute multiple instruction streams on a single core and have dozens of cores running independent instruction streams, making them **both SIMD and MIMD**.
- **Memory:** GPUs can have both shared and distributed memory.
- **Growing Popularity:** They are becoming increasingly popular for general high-performance computing problems, as they are capable of very high computational power.

MIMD Systems

Multiple Instruction, Multiple Data (MIMD) systems support multiple independent instruction streams operating on multiple data streams simultaneously.

They typically consist of a collection of fully autonomous processors, each with its own control unit and datapath.

Unlike SIMD systems, MIMD systems are usually **asynchronous**, meaning the processors can operate at their own pace without a global clock.

There are two principal types of MIMD systems, classified by how their processors access memory:

Shared-Memory Systems:

A collection of autonomous processors is connected to a single memory system. All processors can access any memory location.

Communication and coordination are typically **implicit**, as processors communicate by accessing and updating shared data structures.

This is illustrated in **Figure 2.3**.

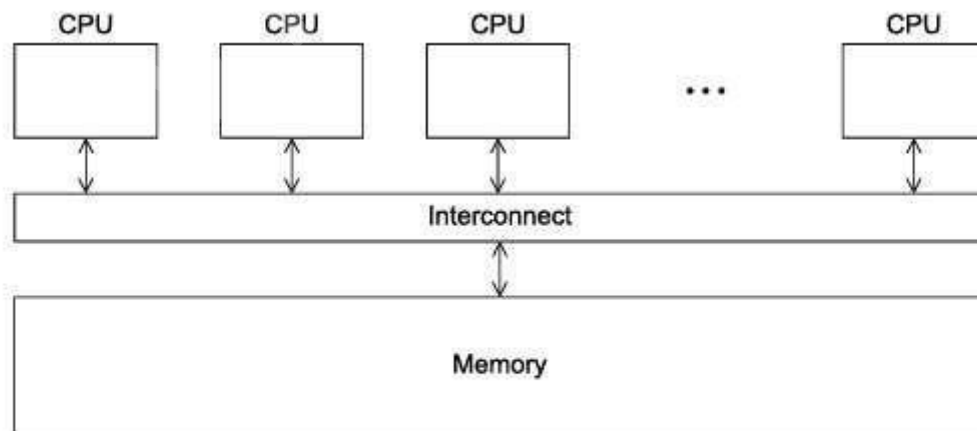


FIGURE 2.3

A shared-memory system.

Distributed-Memory Systems:

Each processor is paired with its own private memory, and these processor-memory pairs are connected by an interconnection network.

Communication and coordination are **explicit**, as processors must send messages or use special functions to access another processor's memory.

This is illustrated in **Figure 2.4**.

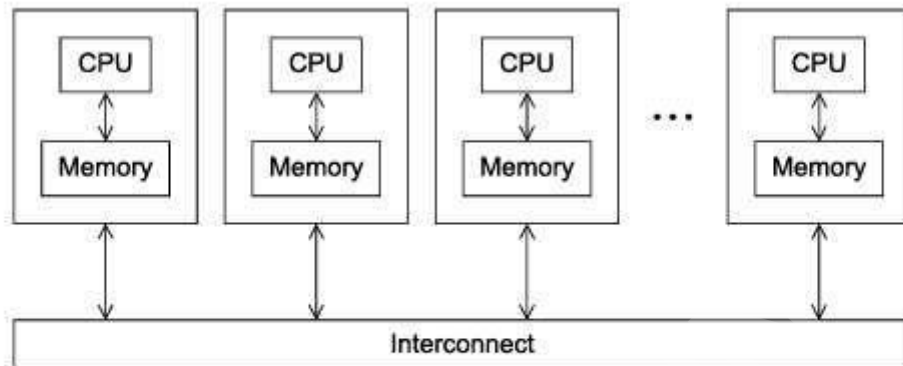


FIGURE 2.4

A distributed-memory system.

Shared-Memory Systems

The most common shared-memory systems today are multi-core processors. In systems with multiple multi-core processors, memory access can be further classified:

UMA (Uniform Memory Access):

In a UMA system, all processors are connected directly to a shared main memory. This means all cores have the **same access time** to all memory locations, which generally makes them easier to program.

This is shown in **Figure 2.5**.

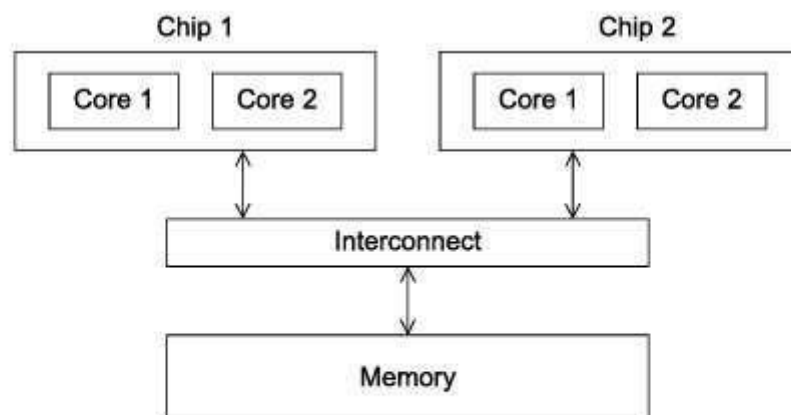


FIGURE 2.5

A UMA multicore system.

NUMA (Nonuniform Memory Access):

In a NUMA system, each processor has a direct connection to a block of main memory. A core can access memory in its directly connected block more quickly than memory that must be accessed through another chip. While this can complicate programming, it can also lead to faster access times to local memory and support larger amounts of total memory.

This is shown in **Figure 2.6**.

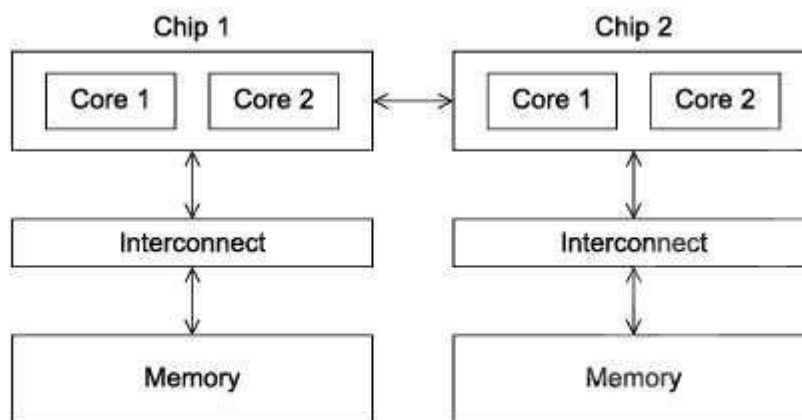


FIGURE 2.6

A NUMA multicore system.

Distributed-Memory Systems

The most widely available distributed-memory systems are **clusters**. A cluster is a collection of commodity systems (like PCs) connected by a commodity network (like Ethernet). The individual nodes within a cluster are often themselves shared-memory systems, making modern clusters a type of **hybrid system**.

A **grid** is a larger-scale, geographically distributed distributed-memory system that connects different types of hardware (making it heterogeneous) into a unified system.

Interconnection networks**Interconnects for Shared-Memory Systems****Bus:**

An older type of interconnect that uses shared communication wires. It is inexpensive and flexible, but performance degrades significantly as the number of connected devices increases due to contention for the shared wires. For this reason, buses are being replaced by switched interconnects in larger systems.

Switched Interconnects (e.g., Crossbar):

These use switches to control data routing. A crossbar is a fast and powerful example that allows multiple devices to communicate simultaneously without contention, provided they are not trying to access the same memory module. However, crossbars are much more expensive to build than bus-based systems.

Figure 2.7 illustrates a crossbar switch, showing how multiple processors can access different memory modules simultaneously.

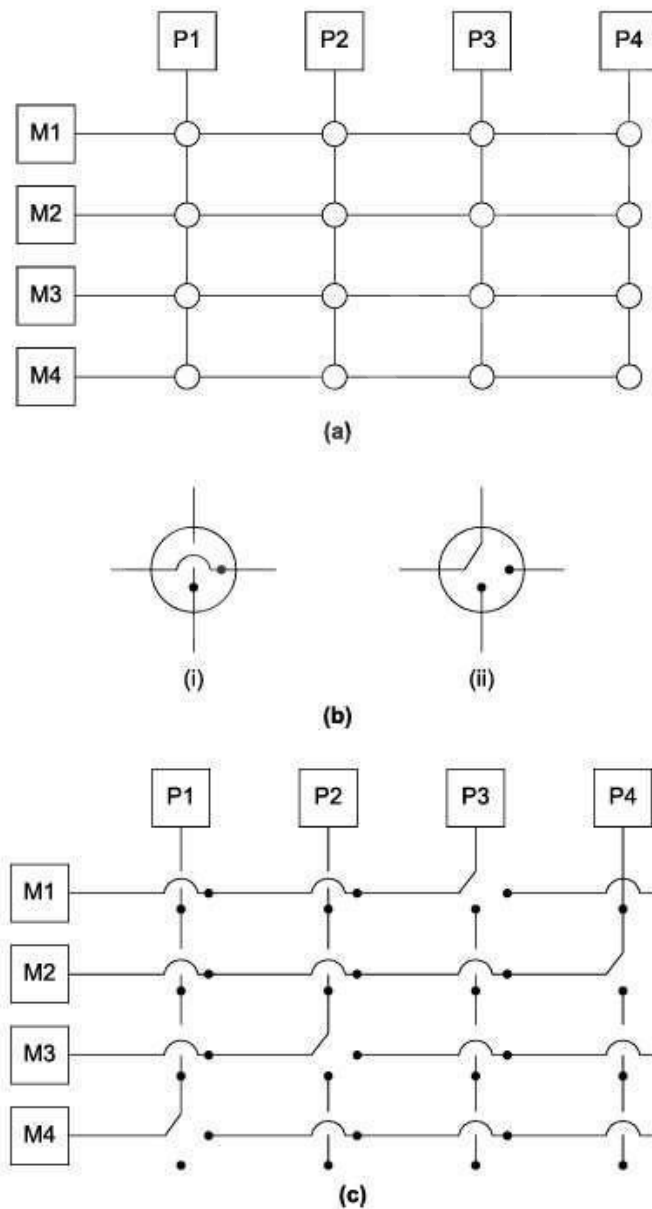


FIGURE 2.7

(a) A crossbar switch connecting four processors (P_i) and four memory modules (M_j);
 (b) configuration of internal switches in a crossbar; (c) simultaneous memory accesses by the processors.

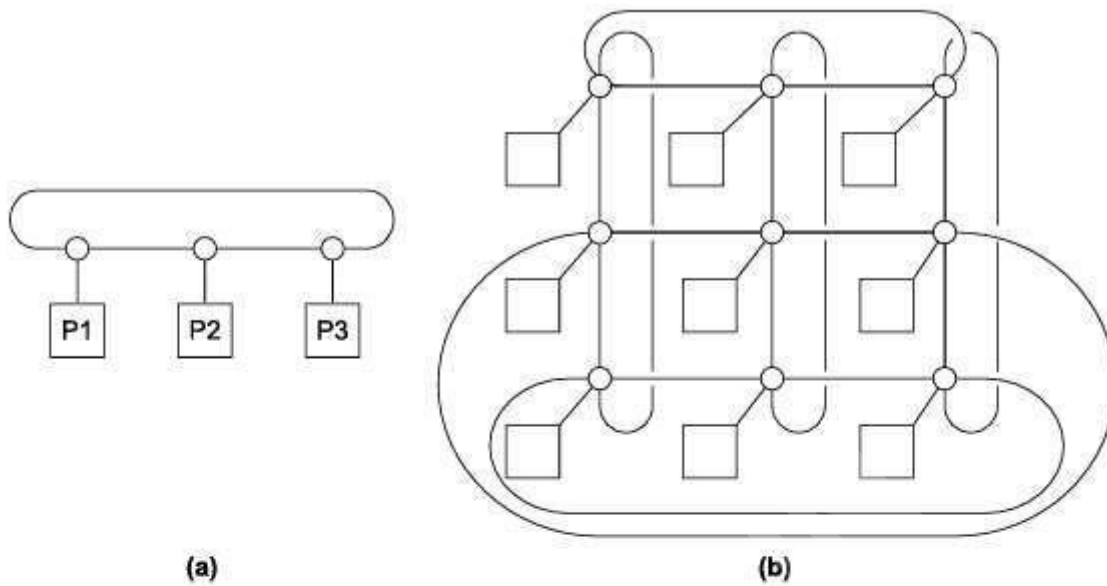


FIGURE 2.8

(a) A ring and (b) a toroidal mesh.

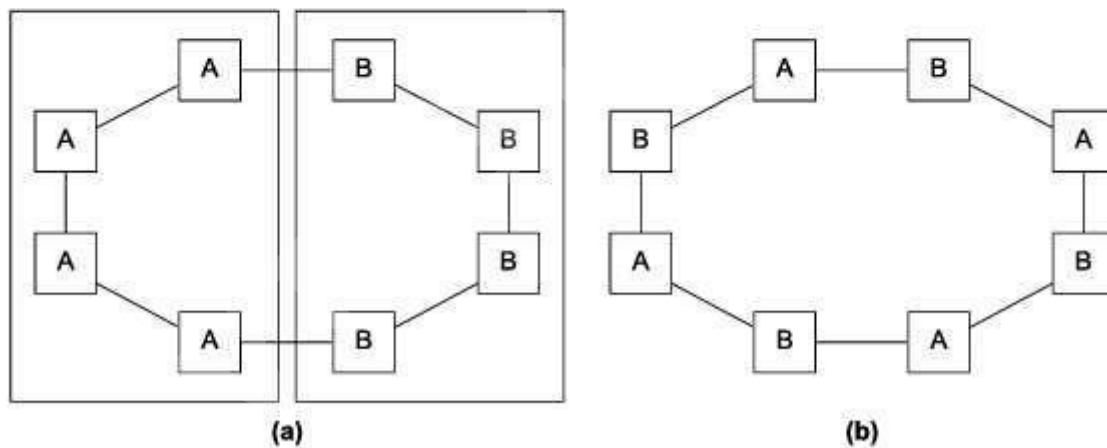


FIGURE 2.9

Two bisections of a ring: (a) only two communications can take place between the halves and (b) four simultaneous connections can take place.

Interconnects for Distributed-Memory Systems

These interconnects are often divided into two groups: direct and indirect.

Direct Interconnects

In a direct interconnect, each switch is directly connected to a processor-memory pair, and the switches are connected to each other.

Examples: Common examples include the **ring** and the **two-dimensional toroidal mesh** (Figure 2.8). A mesh is more expensive than a ring but offers greater connectivity and more opportunities for simultaneous communication.

Connectivity Measures:

- **Bisection Width:** This is a measure of the "worst-case" connectivity. It is defined as the minimum number of links that must be removed to divide the system's nodes into two equal halves. (Figures 2.9, 2.10)

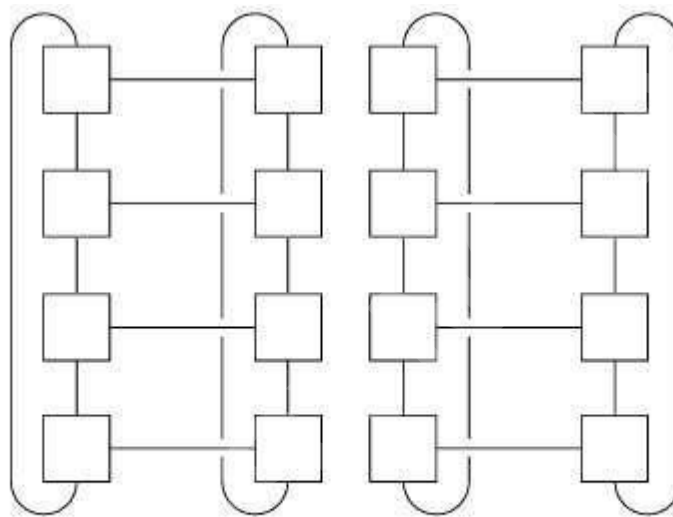


FIGURE 2.10

A bisection of a toroidal mesh.

- **Bisection Bandwidth:** A related measure that sums the bandwidth of the links in the bisection.

Theoretical Ideals:

- **Fully Connected Network:** The ideal direct interconnect where every switch is connected to every other switch (Figure 2.11). It has the highest connectivity but is impractical for all but the smallest systems.
- **Hypercube:** A highly connected and practical direct interconnect built inductively from simpler structures (Figure 2.12). It offers better connectivity than a ring or mesh but is more expensive to construct.

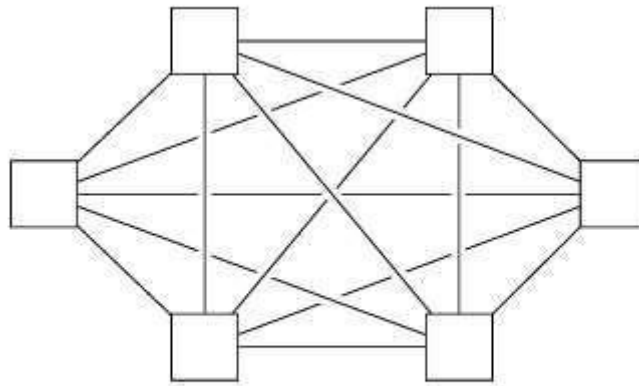


FIGURE 2.11

A fully connected network.

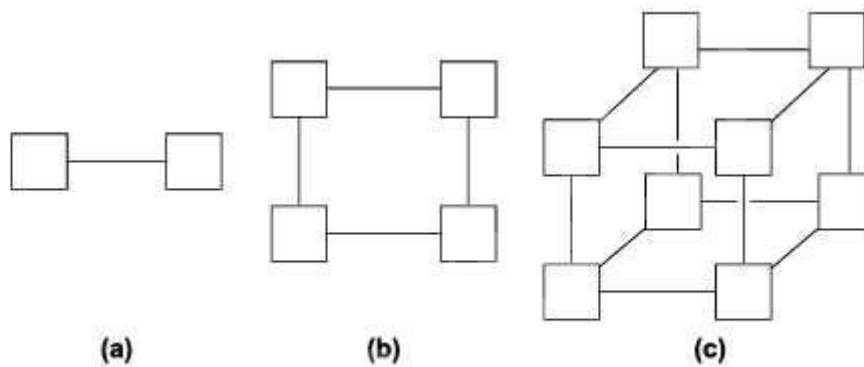


FIGURE 2.12

(a) One-, (b) two-, and (c) three-dimensional hypercubes.

Indirect Interconnects

In an indirect interconnect, some switches may not be directly connected to a processor (**Figure 2.13**).

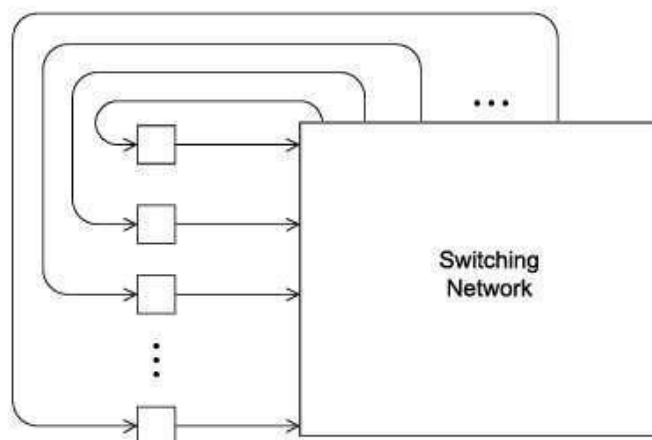


FIGURE 2.13

A generic indirect network.

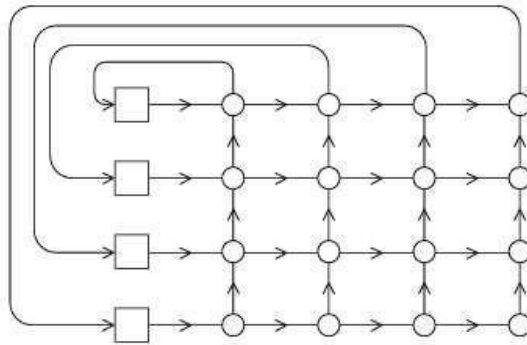


FIGURE 2.14

A crossbar interconnect for distributed-memory.

Examples:

- **Crossbar:** A distributed-memory crossbar uses unidirectional links to allow all processors to communicate simultaneously, as long as they are not sending messages to the same destination processor (**Figure 2.14**).
- **Omega Network:** This is a less expensive indirect network that uses 2x2 switches (**Figure 2.15, 2.16**). Unlike a crossbar, it has communications that cannot occur simultaneously, but its cost is significantly lower.

Performance Metrics: Latency and Bandwidth

- **Latency:** The time it takes for data to start arriving at its destination after the source begins transmitting. It is the time for the first byte to be received.
- **Bandwidth:** The rate at which data is received after the first byte has arrived. It is usually measured in bytes or bits per second.

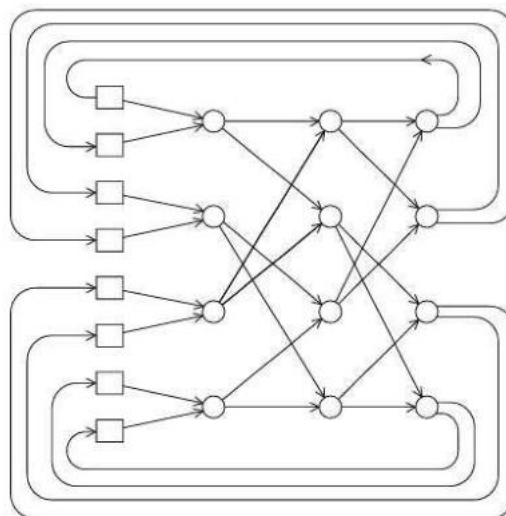
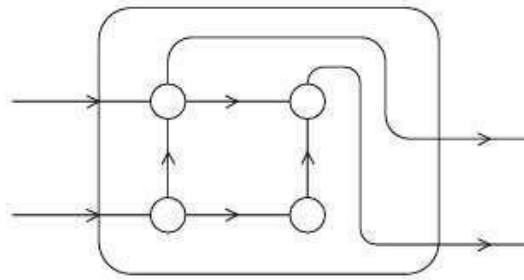


FIGURE 2.15

An omega network.

**FIGURE 2.16**

A switch in an omega network.

The total time to transmit a message is a combination of these two factors: $\text{time} \approx \text{latency} + (\text{message_size} / \text{bandwidth})$.

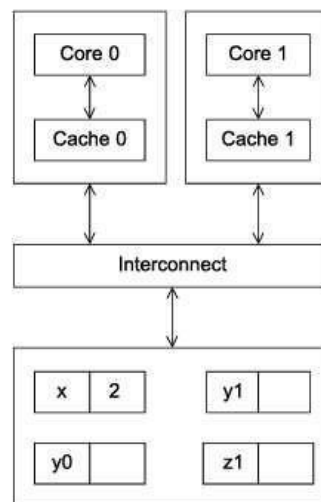
Cache coherence

The cache coherence problem arises in shared-memory systems where each core has its own private cache. Since CPU caches are managed by hardware, a core has no direct control over them.

The Problem:

When multiple cores have a cached copy of a shared variable, and one core updates its copy, the other cores may not see this update.

This can lead to unpredictable behavior and incorrect results. The text gives an example where a core might use an outdated value of a shared variable x from its cache, even after another core has updated the value in main memory. **Figure 2.17** illustrates a shared-memory system with two cores and two caches.

**FIGURE 2.17**

A shared-memory system with two cores and two caches.

Solutions to Cache Coherence

There are two main approaches to ensuring that all cores have a consistent view of shared data:

1. Snooping Cache Coherence:

- **How it works:** It is based on a shared bus or interconnect that supports broadcasts. When a core updates a variable in its cache, it broadcasts this information. Other cores "snoop" the bus, see the broadcast, and mark their local copy of that variable's cache line as invalid, forcing them to get the updated version later.
- **Limitation:** Snooping is not scalable to very large systems because a broadcast across a large network is extremely expensive and causes performance to degrade.

2. Directory-Based Cache Coherence:

- **How it works:** This approach uses a distributed data structure called a **directory** to track the status and location of each cache line. When a variable is updated, the directory is consulted, and only the specific cores that have a cached copy of that variable are contacted to invalidate their copies.
- **Trade-off:** This method requires substantial additional storage for the directory but is much more scalable than snooping because it avoids expensive broadcasts.

The Problem of False Sharing

False sharing is a performance issue that arises because caches operate on entire cache lines, not individual variables.

How it occurs:

It happens when two or more cores repeatedly access different, independent variables that are located within the **same cache line**.

The Consequence:

When one core updates its variable, the entire cache line is invalidated for the other cores.

Even though they are accessing different variables, the other cores must then fetch a fresh copy of the entire cache line from memory, causing a significant performance penalty.

The Fix:

The text suggests using temporary, local storage for each core to perform calculations before copying the results to shared memory, which reduces the frequency of accessing the shared cache line.

This avoids the performance degradation without affecting the correctness of the results.

Shared-memory vs. distributed-memory

Shared-Memory Systems

Definition: In a shared-memory system, multiple processors (or cores) are connected to a single memory system. All processors can access any memory location, and they coordinate their work implicitly by reading from and writing to shared data structures.

Advantages:

- Programmers often find them more appealing to program, as they can coordinate work implicitly through shared data rather than explicit message passing.

Disadvantages:

- The primary disadvantage is the high cost of scaling the interconnect.
- Bus-based interconnects become inefficient as more processors are added due to increased contention.
- Large crossbar interconnects are very expensive to build.

Uses:

- Shared-memory systems are suitable for systems with only a few processors due to the interconnect scaling limitations.

Distributed-Memory Systems

Definition: In a distributed-memory system, each processor has its own private memory. The processors are connected by an interconnection network and must communicate with each other explicitly by sending messages to coordinate their work.

Advantages:

- Interconnects like the hypercube and toroidal mesh are relatively inexpensive.

- They are highly scalable, allowing systems with thousands of processors to be built.
- They are well-suited for problems requiring vast amounts of data or computation.

Disadvantages:

- The concept of explicitly sending messages can be less appealing to programmers, which implies a more complex programming model.

Uses:

- These systems are used for solving problems that require immense computational power and large amounts of data.

Coordinating the processes/threads

1. For simple problems, just divide the work.

Some problems are "embarrassingly parallel," meaning they can be solved by simply dividing the work among processes or threads. A simple array addition is given as an example:

```
double x[n] , y[n];  
...  
for ( int i = 0; i < n; i++)  
    x [i] += y[i];
```

For such problems, the programmer only needs to:

- Divide the work so that each process/thread gets roughly the same amount (a process called **load balancing**).
- Minimize the amount of communication required between processes/threads.

2. For most problems, additional coordination is needed.

The vast majority of problems are more complex and require additional coordination between processes and threads. For these problems, programmers also need to:

- **Arrange for synchronization:** This involves ensuring that processes or threads wait for each other at certain points in the program before proceeding.
- **Arrange for communication:** This involves setting up the transfer of data between processes or threads.

The text notes that synchronization and communication are often interrelated. For example, in distributed-memory programs, synchronization can be achieved implicitly through communication, while in shared-memory programs, communication can be achieved implicitly through synchronization.

Shared-memory

Shared-Memory Programs

In shared-memory programs, variables can be either shared or private. Communication between threads is done implicitly through shared variables, which can be accessed by any thread. Private variables are accessible only by a single thread.

Dynamic vs. Static Threads

Dynamic Threads:

A master thread forks worker threads as needed to handle specific tasks. Once a worker thread completes its task, it terminates. This approach efficiently uses system resources because threads are only active when they are needed.

Static Threads:

All threads are forked at the beginning of the program and run until all work is completed. While this may be less efficient in terms of resource use if threads are idle, it avoids the time overhead of repeatedly forking and joining threads, potentially leading to better performance.

Nondeterminism and Race Conditions

Nondeterminism:

This occurs in asynchronous MIMD systems, where a program's output can vary from one run to another even with the same input. For example, a simple printf statement can have its output order change from run to run:

```
...  
printf("Thread %d > my_x = %d\n", my_rank , my_x);  
...
```

Race Conditions:

This is a serious type of nondeterminism that can lead to program errors. A race condition occurs when multiple threads try to access and modify a shared resource simultaneously, and the outcome depends on which thread "wins the race." A simple example is two threads trying to update a shared variable x:

```
my_val = Compute_val(my_rank);  
x += my_val;
```

The `x += my_val` operation is not atomic and can lead to an incorrect final value for x if both threads try to update it at the same time.

Solutions to Race Conditions

To prevent race conditions, a **critical section** is used. This is a block of code that can only be executed by one thread at a time, ensuring **mutual exclusion**.

Mutex (Mutual Exclusion Lock):

The most common mechanism for enforcing mutual exclusion. A thread must "lock" a mutex before entering a critical section and "unlock" it upon exiting. For the example above, a mutex would be used like this:

```
my_val = Compute_val(my_rank);  
Lock(&add_my_val_lock);  
x += my_val;  
Unlock(&add_my_val_lock);
```

This ensures that only one thread at a time can execute the `x += my_val` statement.

Busy-Waiting: A thread enters a loop to repeatedly check a condition until it is met. An example using a shared variable `ok_for_1` to control execution order is:

```
if (my_rank == 1)  
    while (!ok_for_1); /* Busy-wait loop */  
x += my_val; /* Critical section */  
if (my_rank == 0)  
    ok_for_1 = true;
```

This method can be wasteful of resources, as the core is busy doing no useful work while waiting.

Thread Safety

A function is **thread-safe** if it can be safely used in a multithreaded program without causing errors. Functions written for serial programs, especially those that use **static local variables**, can be a source of errors in multithreaded environments.

The C function strtok is a classic example of a function that is not thread-safe because its internal state (a static variable) can be corrupted if multiple threads call it concurrently.

Distributed-memory

Distributed-Memory Systems

In distributed-memory systems, cores can only directly access their own private memories. Communication between processes is explicit and often handled through a message-passing API, most commonly the Message-Passing Interface (MPI).

Message-Passing

Message-passing provides functions (at a minimum, send and receive) for processes to communicate.

Example: A simple send/receive operation between two processes:

```
char message[100];
...
my_rank = Get_rank();
if (my_rank == 1) {
    sprintf(message, "Greetings from process 1");
    Send( message, MSG_CHAR, 100, 0);
} else if (my_rank == 0) {
    Receive(message, MSG_CHAR, 100, 1);
    printf("Process 0 > Received: %s\n", message);
}
```

Characteristics: Programs are typically **SPMD (Single Program, Multiple Data)**, where all processes run the same executable but perform different actions based on their rank.

Pros: It is a powerful and versatile API used on the most powerful computers in the world.

Cons: It is very low-level and requires significant effort to parallelize a serial program, often described as the "assembly language of parallel programming."

One-Sided Communication

Also known as remote memory access, this paradigm allows a single process to update another process's remote memory without the second process's explicit participation (i.e., without a matching receive call).

Advantages: This can simplify communication and reduce overhead by eliminating the need for two-sided synchronization.

Challenges: It can be difficult to ensure the safety and correctness of memory access. For example, the sending process needs a way to know it's safe to write to remote memory, and the receiving process needs a way to know the data has been updated. This can introduce complex issues and errors that are hard to track down.

Partitioned Global Address Space (PGAS) Languages

PGAS languages provide a shared-memory programming model for distributed-memory hardware, aiming to simplify programming while maintaining high performance.

Core Concept: The language creates a single logical address space that spans the private memories of all processes.

How it Works: To prevent poor performance from accessing remote memory, PGAS languages give the programmer tools to control how shared data is distributed. Programmers can ensure that a process primarily accesses data that is local to its own memory, which is much faster.

Example: In a vector addition, a programmer can define the arrays as shared but then ensure that each process only works on the portion of the array that is located in its own local memory:

```
shared int n = . . . ;
shared double x[n] , y[n];
private int i, my_first_element , my_last_element;
. . .
for (i = my_first_element; i <= my_last_element; i++)
    x [i] += y[i];
```

This approach allows the programmer to use a familiar shared-memory-like syntax while still manually managing data locality for performance.