Module-1

Introduction to parallel programming, Parallel hardware and parallel software – Classifications of parallel computers, SIMD systems, MIMD systems, Interconnection networks, Cache coherence, Shared-memory vs. distributed-memory, Coordinating the processes/threads, Shared-memory, Distributed-memory.

1. INTRODUCTION

1. Parallel hardware

Multiple issue and pipelining allow different functional units to execute simultaneously, so they can be considered parallel hardware. However, since this parallelism is not usually visible to the programmer, we treat them as extensions of the basic von Neumann model. For our purposes, parallel hardware is limited to systems where the programmer must modify their source code to explicitly exploit parallelism.

1.1 Classification of parallel computers

We use two independent ways to classify parallel computers. The first is Flynn's taxonomy, which categorizes systems based on the number of instruction streams and data streams they handle, distinguishing between single instruction stream (SIMD) and multiple instruction stream (MIMD) systems. The second classification focuses on memory access, differentiating between shared memory systems, where cores share and coordinate via common memory, and distributed memory systems, where each core has private memory and communicates over a network.

1.2 SIMD Systems

Single Instruction, Multiple Data (SIMD) systems are a type of parallel system.

1. SIMD systems apply the same instruction simultaneously to multiple data streams.
2. Conceptually, a SIMD system has one control unit and multiple datapaths.
3. The control unit broadcasts an instruction to all datapaths, each of which either executes the instruction on its data or remains idle.
4. For example, in vector addition, SIMD can add elements of two arrays, x and y, elementwise in parallel.

Consider the loop: `for (i = 0; i < n; i++) x[i] += y[i];`

5. If the SIMD system has **n datapaths**, each datapath i can load x[i] and y[i], perform the addition x[i] += y[i], and store the result back in x[i].
6. If the system has **m datapaths** where m < n, the additions are executed in blocks of m elements at a time.

   For example, if m = 4 and n = 15, the system processes elements in groups: 0–3, 4–7, 8–11, and 12–14.

7. In the last group (elements 12–14), only three data paths are used, so one data path remains idle. The requirement that all data paths execute the same instruction or stay idle can reduce SIMD performance.

   For instance, if we want to add only when y[i] is positive:
   ```
   for (i = 0; i < n; i++)   if
   (y[i] > 0.0) x[i] += y[i];
   ```

   some data paths may be idle depending on the condition, leading to inefficiency.

In this setup, each datapath loads an element of y and checks if it is positive; if so, it performs the addition, otherwise it remains idle while others continue. Classical SIMD systems require all datapaths to operate synchronously, waiting for each instruction to be broadcast, and they cannot store instructions for later execution.

SIMD is well-suited for parallelizing simple loops over large data arrays, a form of parallelism known as **data-parallelism**, where processors apply similar instructions to different data subsets. While SIMD can be highly efficient for data-parallel problems, it struggles with other types of parallelism. Historically, SIMD systems were prominent in the 1990s but declined, with modern uses mainly in **vector processors**, GPUs, and desktop CPUs incorporating SIMD features.

Vector Processors

Vector processors are defined by their ability to operate on arrays or vectors of data, unlike conventional CPUs that handle individual scalar elements. Modern vector processors typically exhibit the following characteristics:

- ***Vector registers*:** Vector registers are designed to store multiple operands as vectors and perform operations on all elements simultaneously. The system defines a fixed vector length, typically ranging from 4 to 256 elements of 64 bits each.

- ***Vectorized and pipelined functional units*:** Vectorized and pipelined functional units apply the same operation to each element within a vector, or to corresponding pairs of elements in two vectors. This means vector operations follow the SIMD (Single Instruction, Multiple Data) model.
- ***Vector instructions*:** operate on entire vectors instead of individual scalar values. If the vector length is vector_length, a simple loop like

  ```
  for (i = 0; i < n; i++) x[i] += y[i];
  ```

  can be executed using just one load, add, and store operation per block of vector_length elements. In contrast, a conventional system performs these operations for each individual element, making vector instructions much more efficient for such tasks.
- ***Interleaved memory:*** Interleaved memory consists of multiple independent banks that can be accessed separately. After accessing one bank, there is a delay before it can be used again, but other banks can be accessed without waiting. Distributing vector elements across these banks allows for minimal or no delay when loading or storing successive elements.
- ***Strided memory access and hardware scatter/gather:*** Strided memory access occurs when a program accesses elements of a vector at fixed intervals. For example, accessing the first, fifth, ninth elements, and so on, represents a stride of four. Scatter/gather involves reading or writing elements at irregular intervals within a vector. For instance, accessing the first, second, fourth, and eighth elements demonstrates scatter/gather access. To improve performance, typical vector systems include specialized hardware that accelerates both strided access and scatter/gather operations.

Vector processors are fast and user-friendly for many applications, with compilers that effectively identify code suitable for vectorization. These compilers also detect loops that cannot be vectorized and often explain why, helping users decide if the code can be rewritten for vectorization. Vector systems offer very high memory bandwidth and efficiently use every loaded data item, unlike cache-based systems. However, they struggle with irregular data structures and have limited **scalability** in handling larger problems, as increasing vector length is challenging. Instead, modern systems typically scale by adding more vector processors, while long-vector processors remain custom-made and expensive, and commodity systems offer limited support for short vectors.

Graphics processing units

Real-time graphics APIs represent object surfaces using points, lines, and triangles, which are then converted into pixels through a graphics processing pipeline. Several stages of this pipeline are programmable and controlled by shader functions, which are typically short snippets of code, often just a few lines of C. These shader functions are inherently parallel because they can be applied independently to multiple elements, such as vertices, in the graphics stream. Since nearby elements usually follow the same control flow, GPUs optimize performance using SIMD parallelism. Modern GPUs achieve this by incorporating many datapaths, often 128 or more, within each processing core.

1. Processing a single image can require hundreds of megabytes of data, so GPUs need very high data movement rates and rely heavily on hardware multithreading to avoid memory stalls.
2. Some GPU systems can store the state of over a hundred suspended threads per executing thread, with the number of threads depending on shader function resource usage.
3. GPUs need many threads processing large amounts of data to keep datapaths busy, which can lead to poor performance on smaller problems.
4. GPUs are not purely SIMD systems; while datapaths on a core use SIMD, current GPUs can run multiple instruction streams per core and have many cores capable of independent instruction streams, blending SIMD and MIMD features.
5. GPUs can use shared or distributed memory, with some systems combining both by having groups of cores sharing memory blocks and communicating over a network.
6. GPUs are increasingly popular for high-performance computing, supported by specialized programming languages to help users harness their power.

1.3 MIMD Systems

Multiple Instruction, Multiple Data (MIMD) systems support multiple instruction streams running concurrently on multiple data streams. Typically, MIMD systems consist of independent processing units or cores, each with its own control unit and datapath.
MIMD systems are typically asynchronous, allowing processors to operate independently at their own pace without a global clock. Without programmer-imposed synchronization, processors— even if running the same instructions—may be executing different statements at the same time. There are two main types of MIMD systems: **shared-memory** and **distributed-**

me mory systems. **In shared-memory systems**, multiple autonomous processors are connected to a common memory through an interconnection network, allowing each processor to access any memory location. Communication between processors typically occurs implicitly by accessing shared data structures. In contrast, **distributed-memory systems** pair each processor with its own private memory, and these processor-memory pairs communicate explicitly over a network. Communication in distributed-memory systems usually involves sending messages or using special functions to access another processor's memory. See Figs. 1.1 and 1.2

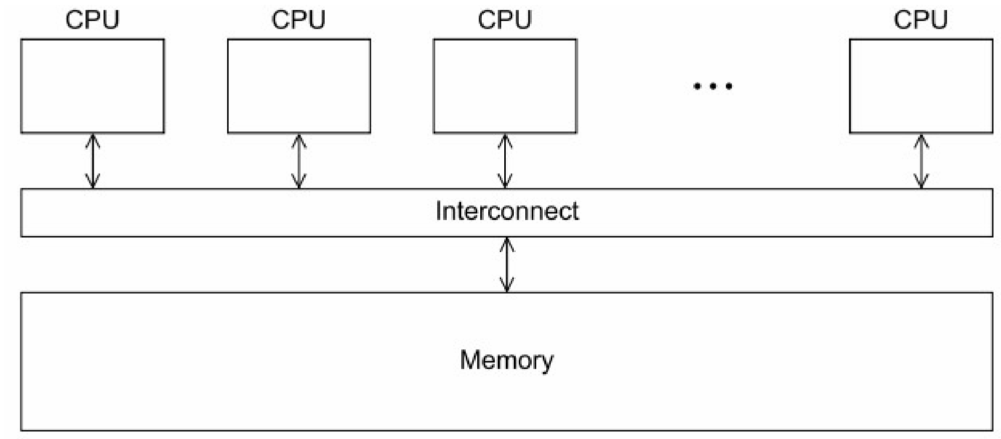


Fig 1.1: A shared-memory system

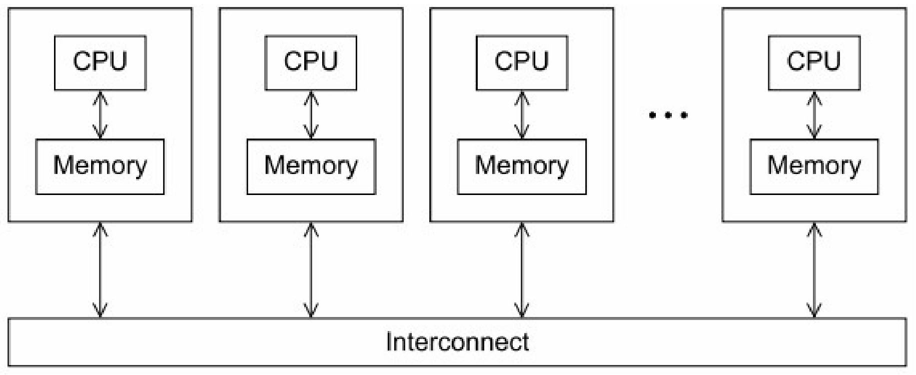


Fig 1.2: A distributed-memory system

Shared-memory systems

The most widely available shared-memory systems use one or more multicore processors.

a multicore processor has multiple CPUs or cores on a single chip. Typically, the cores have private level 1 caches, while other caches may or may not be shared between the cores. In shared-memory systems with multiple multicore processors, the interconnect may either link all processors directly to a common main memory, or assign each processor its own memory block. In the latter case, processors access each other's memory blocks via specialized hardware integrated into the processors themselves. See Figs. 1.3 and 1.4

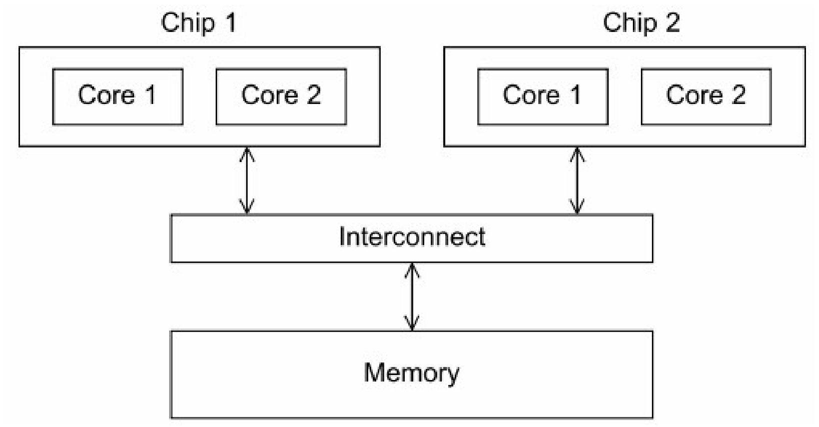


Fig 1.3: A UMA multicore system.

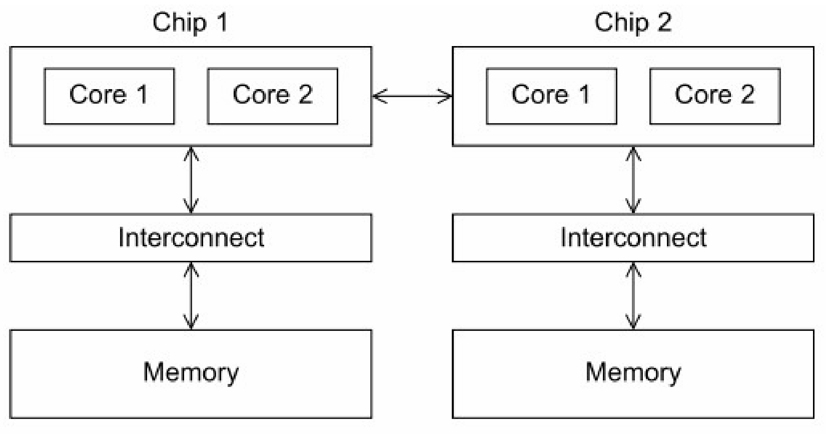


Fig 1.4: A NUMA multicore system.

In systems where all cores access memory with equal latency, the memory access time remains uniform regardless of which core accesses which memory location. This configuration is known as a **Uniform Memory Access (UMA)** system. In contrast, when each core has faster access to its own local memory block and slower access to other cores' memory, the system is referred to as a **Non-Uniform Memory Access (NUMA)** system. While UMA systems are generally simpler to program due to consistent access times, NUMA systems offer performance benefits by providing quicker access to local memory. Additionally, NUMA architectures can scale more effectively, allowing support for larger overall memory capacity than UMA systems.

Distributed-memory systems

The most common type of distributed-memory systems today is known as clusters. These are built from multiple standard computing systems—such as personal computers connected through standard networking technologies like Ethernet. Typically, each node within a cluster is itself a shared-memory system containing one or more multicore processors. To differentiate these from purely distributed-memory systems, they are often referred to as hybrid systems, and it is now generally assumed that cluster nodes have shared-memory capabilities. In contrast, a grid connects geographically dispersed computers and provides the infrastructure to function as a single distributed-memory system, often consisting of heterogeneous hardware across different nodes.

1.4 Interconnection Networks

The interconnect is crucial to the performance of both shared- and distributed-memory systems; even with extremely fast processors and memory, a slow interconnect can significantly hinder the performance of most parallel programs. While some interconnects share common features, their differences are substantial enough to warrant separate consideration for shared- and distributed-memory architectures.

Share-memory internconnects

In earlier shared-memory systems, a bus was commonly used to connect processors and memory. A bus consists of shared communication lines and control hardware, allowing multiple devices to connect easily and cost-effectively. However, because all devices share the same communication lines, contention increases as more devices are added, leading to performance degradation. This means that with many processors, delays in accessing memory become frequent. As a result, modern systems are moving from buses to **switched interconnects** to handle larger processor counts more efficiently.

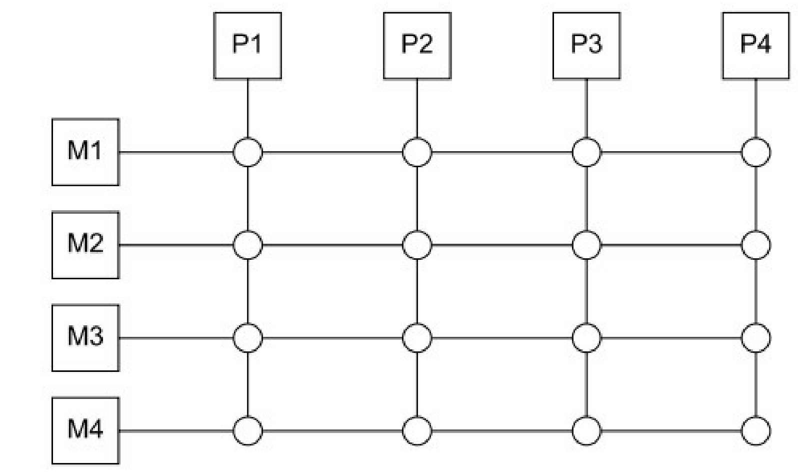


Fig 1.5 (a): A crossbar switch connecting four processors (Pi) and four memory modules (Mj)

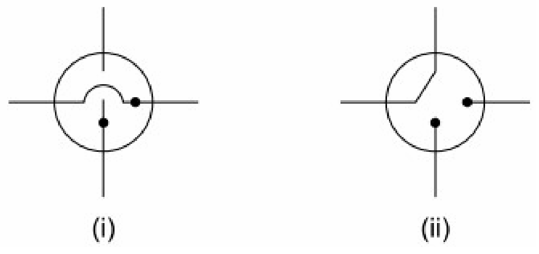


Fig 1.5 (b): Configuration of internal switches in a crossbar

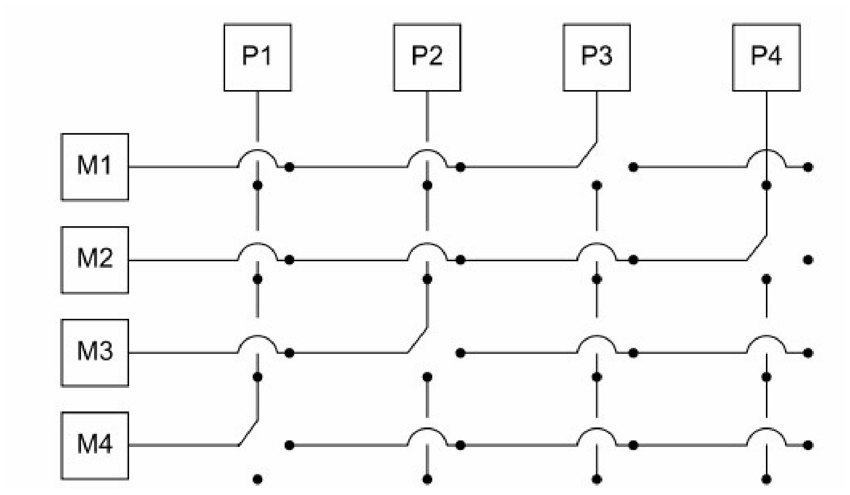


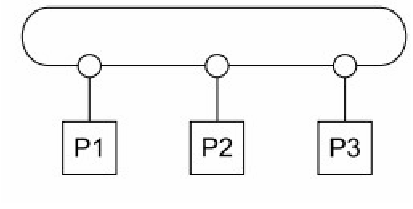
Fig 1.5 (c): Simultaneous memory accesses by the processors.

- **Switched interconnects** control the flow of data between connected devices using switches, allowing for efficient and organized communication. A **crossbar** is a basic yet powerful example of a switched interconnect.
- As illustrated in **Fig. 1.5 (a)**, it connects cores or memory modules (represented as squares) through switches (shown as circles) using **bidirectional communication links** (lines).
- Each switch in the crossbar can be conFigd in one of two ways, as illustrated in **Fig. 1.5 (b)**.
- When the system has at least as many memory modules as processors, data conflicts occur only if two processors try to access the same memory module simultaneously.
- **Fig. 1.5 (c)** demonstrates a valid switch configuration where P1 writes to M4, P2 reads from M3, P3 reads from M1, and P4 writes to M2 without any conflicts. Crossbars support simultaneous communication between multiple devices, making them significantly faster than bus-based systems. However, due to the higher cost of switches and links, crossbar systems are more expensive, especially when compared to bus-based systems of similar size.

Distributed-memory interconnects

Distributed-memory interconnects are generally classified into two types: direct interconnects and indirect interconnects. In a direct interconnect, each switch is directly linked to a processormemory unit, and the switches themselves are interconnected to form the network. Fig 1.6 illustrates two examples of direct interconnects—a ring topology in Fig. 1.6 (a) and a twodimensional toroidal mesh in Fig. 1.6 (b). In these diagrams, circles represent switches, squares represent processors, and lines denote bidirectional communication links. One basic metric used to evaluate a direct interconnect is the number of switch-to-switch links, since processor-toswitch links may have different speeds and are not typically included in this count. For instance, in the ring shown in Fig. 1.6 (a), we count 3 switch-to-switch links, while the toroidal mesh in

Fig. 1.6 (b) includes 18 such links, even though the total number of physical links may be higher.



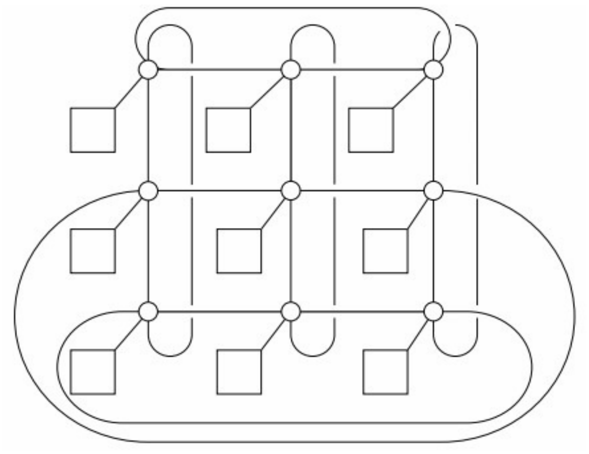
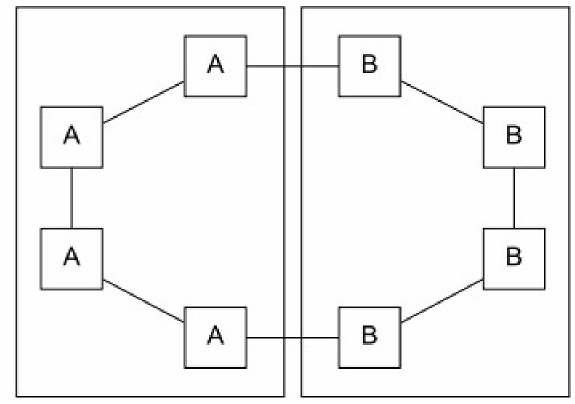
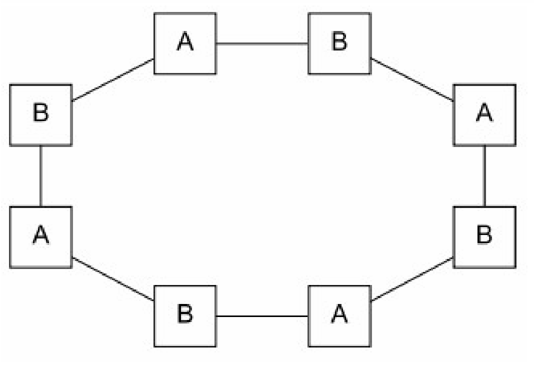


Fig 1.6 (a): A ring **Fig 1.6 (b):** A toroidal mesh

- A ring interconnect is an improvement over a simple bus as it allows multiple communications to occur at the same time; however, it may still result in communication delays if certain processors have to wait for others to finish.
- In contrast, a toroidal mesh supports a greater number of simultaneous communication paths due to its more complex structure, as shown in Fig. 1.6 (b).
- The toroidal mesh is more expensive than a ring because each switch must handle five links instead of three, and for *p* processors, it requires 2p links, while a ring only needs *p* links.
- To better visualize these interconnects, Fig. 1.7 (a) groups nodes and switches together, while Fig. 1.7 (b) separates them to illustrate four simultaneous communications.
- Despite appearing to support four concurrent communications in Fig. 1.7 (b), the bisection width, which estimates worst-case connectivity, is only two—meaning only two communications can occur across a system split in half.
- Overall, the mesh interconnect supports more flexible and parallel communication patterns than a ring, offering higher connectivity and better scalability.





Two bisections of a ring:

Fig 1.7 (a): Only two communications can take place between the halves

Fig 1.7 (b): Four simultaneous connections can take place.

The **bisection width** of a network can also be determined by removing the **minimum number of links** required to divide the system into two equal halves. The total number of links removed in this process represents the bisection width. For a **square two-dimensional toroidal mesh** with $p=q^2$ nodes (where q is even), this division can be achieved by cutting the middle horizontal links and the wraparound horizontal links, as shown in **Fig. 1.8**. This suggests that the bisection width is at most $2q = 2\sqrt{p}$. This is the smallest possible number of links, and the bisection width of a square two-dimensional toroidal mesh is $2\sqrt{p}$

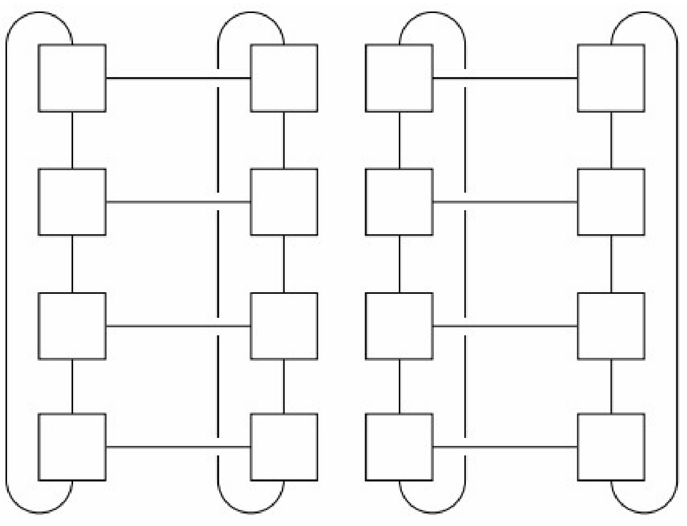


Fig 1.8: A bisection of a toroidal mesh

Link bandwidth refers to the rate at which data can be transmitted, typically measured in megabits or megabytes per second. **Bisection bandwidth**, a metric for network quality, is similar to bisection width but instead sums the bandwidth of the links connecting two halves of a network. For instance, if each link in a ring has a bandwidth of 1 gigabit per second, the ring's bisection bandwidth would be **2 gigabits (or 2000 megabits) per second**.

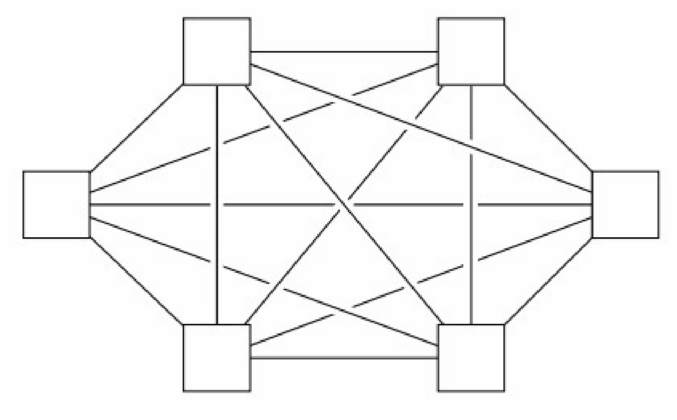


Fig 1.9: A fully connected network

A fully connected network, shown in Fig. 1.9, represents the ideal direct interconnect where each switch is directly linked to every other switch. This type of network has a high **bisection width of $p^2/4$**, offering maximum communication capacity. However, it becomes impractical for large systems, as it requires $p^2/2-p/2$ links and each switch must support **p** connections. As a result, it is considered more of a **theoretical benchmark** than a feasible design and is often used to evaluate the performance of more realistic interconnects.

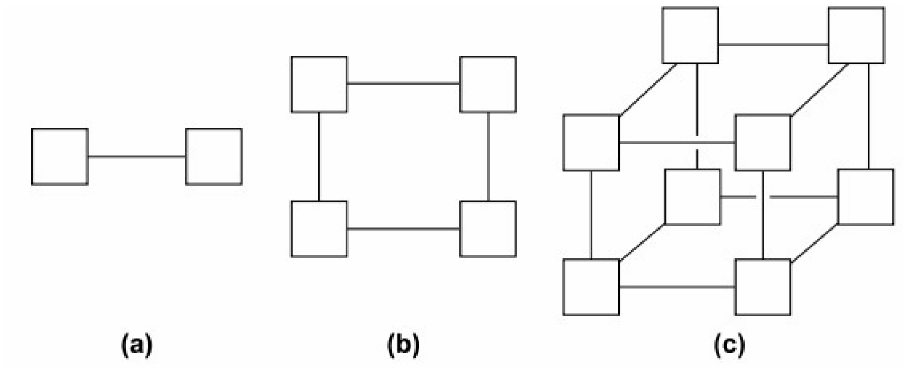


Fig 1.10: (a) One-, (b) two-, and (c) three-dimensional hypercubes

The **hypercube** is a highly connected direct interconnect that has been implemented in real systems. It is built inductively: a 1-dimensional hypercube connects two processors, and each higher-dimensional hypercube is formed by linking corresponding switches of two lowerdimensional hypercubes, as shown in Fig. 1.10. A **d-dimensional hypercube** contains p= 2^d nodes, with each switch connected to a processor and **d** other switches. While its **bisection width** is p/2, offering greater connectivity than rings or toroidal meshes, hypercube switches are more complex and costly, requiring $1+d=1+\log_2(p)$ wires compared to just five in a toroidal mesh.

- **Indirect interconnects** are an alternative to direct interconnects, where switches may not be directly linked to processors.
- These networks are typically illustrated with **unidirectional links**, where each processor has one outgoing and one incoming link connected to a switching network, as shown in Fig. 1.11.
- **Crossbar** and **omega networks** are common examples of simple indirect interconnects.
- In a distributed-memory crossbar (see Fig. 1.12), unidirectional links allow for structured data flow between processors.
- As long as no two processors try to send data to the same destination simultaneously, all processors can communicate with others in parallel, maximizing throughput.

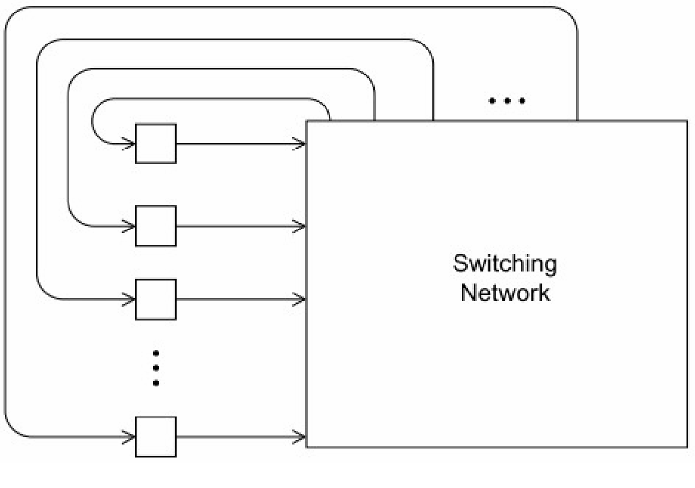


Fig 1.11: A generic indirect network.

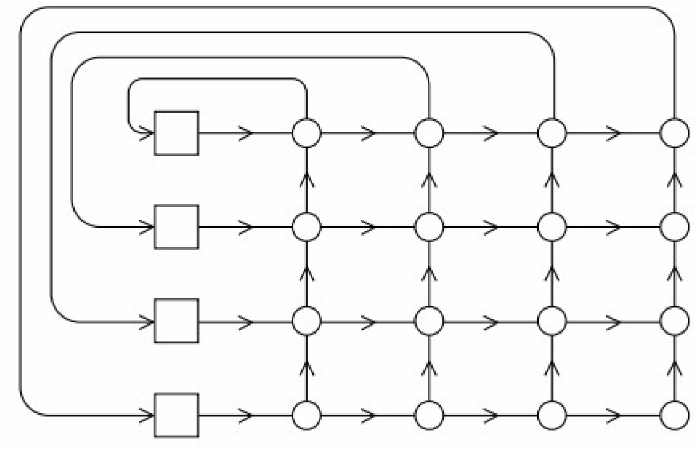


Fig 1.12: A crossbar interconnect for distributed-memory.

An **omega network**, shown **in Fig. 1.13**, is composed of multiple two-by-two crossbar switches (see **Fig. 1.14**), but unlike a full crossbar, it does not support all communications simultaneously. For instance, if processor 0 sends data to processor 6, processor 1 cannot concurrently send data to processor 7. However, the omega network is **more cost-efficient**, requiring only 1/2plog$_2$(p) of the 2X2 crossbar switches, totalling 2plog$_2$(p) switches—significantly fewer than the p^2 switches used in a full crossbar.

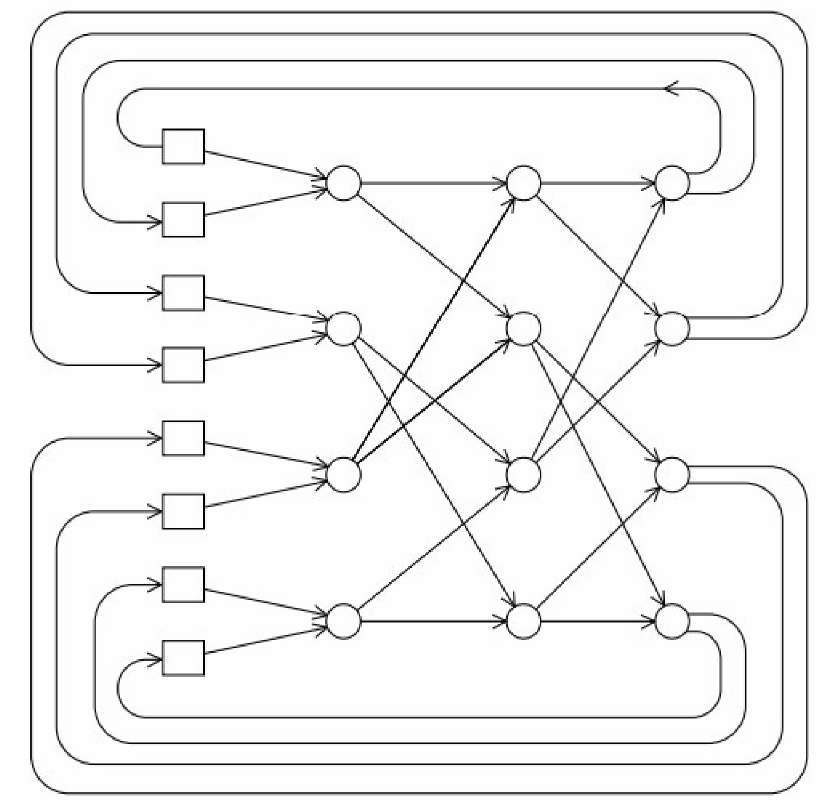


Fig 1.13: An omega network.

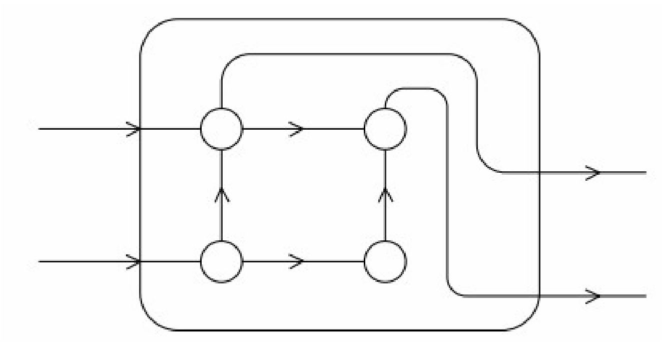


Fig 1.14: A switch in an omega network.

Latency and Bandwidth

When data is transmitted—whether between memory and cache, cache and register, or across nodes in a distributed system—we are primarily concerned with latency and bandwidth. **Latency** is the time delay between the start of transmission and the arrival of the first byte at the destination, while **bandwidth** refers to the rate at which data is received after the first byte. Together, these factors determine the total time to send a message of n bytes across an interconnect with latency l seconds and bandwidth b bytes per second. message transmission time = l + n/b

1.5 Cache coherence

CPU caches are controlled by hardware, meaning programmers do not manage them directly.

This leads to important implications in shared-memory systems, especially when each core has its own private cache. For instance, in a system with two cores and separate caches (see Fig. 1.15), there is no issue as long as both cores are only reading shared data.

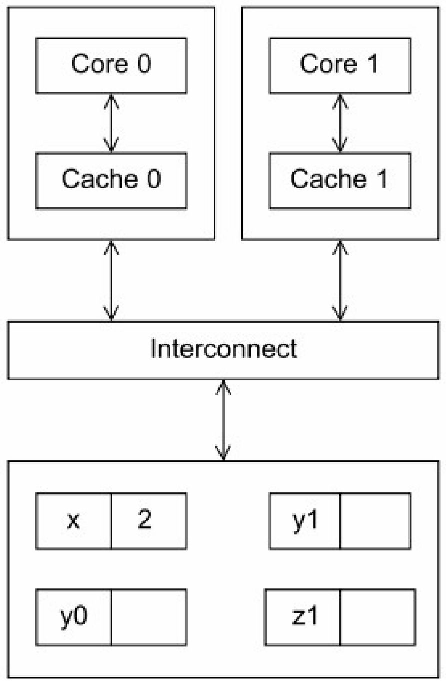


Fig 1.15: A shared-memory system with two cores and two caches.

- The shared variable x is initialized to **2**, while y0 (Core 0), y1, and z1 (Core 1) are private variables owned by their respective cores.
- At **time 0**, both cores read from x:
  - Core 0 sets y0 = x, so y0 becomes **2**.
  - Core 1 sets y1 = 3 * x, so y1 becomes **6** (3 × 2).
- At **time 1**, Core 0 updates x = 7. This change may **only reflect in Core 0's cache**, depending on the cache behavior.
- At **time 2**, Core 1 computes z1 = 4 * x. However, since it may still have the **old value** of x = 2 cached, z1 could become **8** (4 × 2), instead of the expected **28** (4 × 7).
- This inconsistency arises because **Core 1 may not see the update made by Core 0**, unless its cache is invalidated or refreshed.
- This issue occurs **regardless of write policy**:
  - In **write-through**, main memory is updated, but Core 1's cache may still have stale data.
  - In **write-back**, Core 0's cache update may not be written to main memory immediately, leaving Core 1 unaware of the change.
- The core issue is known as the **cache coherence problem**, where **shared variables cached across multiple processors can become inconsistent**, and one core's update is not automatically reflected in another's cache.

Because **programmers lack direct control** over cache updates, the outcome of parallel code involving shared variables like x becomes **unpredictable** unless cache coherence protocols are enforced. **Snooping cache coherence**

- Cache coherence is typically maintained through two main strategies: **snooping and directory-based coherence protocols.**
- In snooping cache coherence, based on bus systems, all cores can observe data transmissions on the shared interconnect.
- When Core 0 updates a variable in its cache and broadcasts that update, other cores like Core 1, which are snooping the bus, can detect the change and invalidate their outdated cache copies.
- The broadcast doesn’t specify the exact variable updated but rather indicates that the cache line containing the data has changed.
- Snooping doesn't require a traditional bus—it just needs a broadcast-capable interconnect so each processor can receive updates from others.
- Snooping works with both write-through and write-back caches: in write-through, updates are visible through regular memory writes; in write-back, extra communication is required since changes stay in cache until explicitly written back to memory.

Directory-based cache coherence

Snooping cache coherence requires a broadcast every time a variable is updated, which becomes inefficient in large systems due to the high cost of broadcasting.

- As the number of cores increases, the performance degrades, making snooping unsuitable for scalable architectures.
- In systems with distributed memory but a unified address space (like in Fig.1.2), cores can access each other’s memory directly, though remote memory access is slower than local access.
- While such systems can support many cores, snooping becomes impractical, since broadcasting updates across a large interconnect introduces significant delays.

Directory-based cache coherence addresses scalability issues by maintaining a directory that tracks the status of each cache line in the system.

- This directory is often distributed, with each core or memory module managing the status of cache lines stored in its local memory.
- When a cache line is read, the corresponding directory entry is updated to reflect which cores hold copies of that line.

During a write operation, the directory is consulted, and all cores with that cache line are notified to invalidate their copies, ensuring coherence.

False Sharing

It's important to note that CPU caches work at the level of cache lines, not individual variables, since they are implemented in hardware. This behavior can significantly impact performance, sometimes negatively—such as when repeatedly calling a function like f (i, j) and adding the results to a vector.

```
int i, j, m, n;
double y[m];

/* Assign y = 0 */
. . .

for (i = 0; i < m; i++)
   for (j = 0; j < n; j++)
      y[i] += f(i,j);
```

```
/* Private variables */
int i, j, iter_count;

/* Shared variables initialized by one core */
int m, n, core_count
double y[m];

iter_count = m/core_count

/* Core 0 does this */
for (i = 0; i < iter_count; i++)
   for (j = 0; j < n; j++)
      y[i] += f(i,j);

/* Core 1 does this */
for (i = iter_count; i < 2*iter_count; i++)
   for (j = 0; j < n; j++)
      y[i] += f(i,j);
```

The given loop structure computes values using a function f(i, j) and accumulates them into elements of an array y, with the outer loop easily parallelizable by dividing iterations among multiple cores.

- If there are core_count cores, each can be assigned approximately m/core_count iterations of the outer loop to process independently.
- In a system with 2 cores, m = 8, and assuming doubles are 8 bytes and cache lines are 64 bytes, the array y fits entirely within a single cache line (since 8 doubles × 8 bytes = 64 bytes).

Though cores operate on different indices of y, the fact that all elements reside in the same cache line leads to a performance issue.

- When one core updates an element in y, the cache line is invalidated for the other core, forcing it to fetch the updated line from memory, despite accessing non-overlapping elements.
- This unnecessary invalidation and memory traffic is known as false sharing, where cores appear to share data due to hardware-level cache behavior, causing inefficient cache utilization.

1.5 Shared-memory vs. distributed-memory

- While shared-memory MIMD systems are easier for programmers to use due to implicit data sharing, scaling them in hardware is costly and complex—especially as interconnect contention increases with more processors.
- Interconnects like buses become inefficient with many cores due to access conflicts, and crossbars, while better performing, are too expensive to scale.
- In contrast, distributed-memory systems using scalable interconnects like hypercubes or toroidal meshes are more cost-effective and can support thousands of processors, making them ideal for large-scale computational problems.

2. Parallel Software

First, let's clarify the terminology: in shared-memory programs, a single process typically forks multiple threads, so we refer to threads performing tasks. In contrast, distributed-memory programs involve multiple processes, and when a concept applies to both models, we refer to processes/threads carrying out tasks.

2.1 Coordinating the processes/threads

In rare cases, achieving excellent parallel performance is straightforward—for example, when simply adding two arrays.

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

To parallelize this task, we simply divide the array elements among the available processes or threads. For instance, with *p* processes or threads, each one handles a distinct block of elements— for example, thread 0 manages elements 0 to *n/p* − 1, thread 1 handles *n/p* to 2*n/p* − 1, and so on.

1. When dividing work among processes or threads, it's important to ensure that

(a) each one gets approximately the same amount of work and

(b) communication between them is kept to a minimum.

This division process, known as load balancing, becomes especially important when the workload isn't known ahead of time and is generated during execution. Although these considerations are often straightforward, they are crucial for efficient parallel execution. Transforming a serial program into a parallel one is called **parallelization**, and when the work can be easily divided, the program is known as **embarrassingly parallel**—a term that, despite its name, reflects a situation that should be celebrated, not avoided.

Unfortunately, most problems are not easily parallelized and require more effort to solve efficiently. As discussed earlier, solving such problems typically involves coordinating the work of processes or threads. This includes

(2) Arranging for synchronization and

(3) Enabling communication among them.

2.2 Shared memory

As mentioned earlier, shared-memory programs use both shared and private variables—shared variables are accessible by all threads, while private ones are typically limited to a single thread. Since threads communicate through shared variables, this communication is generally implicit rather than explicitly defined.

Dynamic and static threads Dynamic threads

- In many environments, shared-memory programs follow a dynamic threading model.
- This usually involves a master thread and a varying number of worker threads that exist only as needed.
- The master thread handles incoming work requests (e.g., over a network), forks a worker thread to process each request, and the worker terminates and rejoins the master after completing its task.
- This approach efficiently utilizes system resources, as threads consume resources only while actively running.

Static threads

- The static thread paradigm involves creating all threads at once after the master thread performs any necessary setup.
- These threads remain active and run until all assigned work is completed, after which they join the master thread.
- The master thread may then handle cleanup tasks, such as freeing memory, before terminating.
- Although this approach can be less resource-efficient—since idle threads still occupy resources—it avoids the overhead of repeatedly forking and joining threads.
- If system resources are sufficient, the static paradigm can offer better performance and aligns more closely with the common model used in distributed-memory programming, making it a practical and consistent choice.

Nondeterminism

In any MIMD system where processors run asynchronously, nondeterminism is common, meaning the same input can lead to different outputs. This occurs because multiple threads execute independently and complete statements at varying rates, causing program results to differ between runs. For example, consider two threads with IDs 0 and 1, each storing a private variable my_x—thread 0 has the value 7, and thread 1 has 19. Both threads then execute the same code, illustrating how timing differences affect outcomes.

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

Then the output could be

```
Thread 0 > my_x = 7
Thread 1 > my_x = 19
```

but it could also be

```
Thread 1 > my_x = 19
Thread 0 > my_x = 7
```

The output from one thread can even be mixed up with the output from another, making it unpredictable. This happens because threads run independently and interact with the operating system, causing the time to complete tasks to vary each time the program runs, so the order of statement completion can't be predicted. In many cases, this nondeterminism isn't a problem—for example, when output is labeled with the thread's rank, the order doesn't really matter. However, in shared-memory programs, nondeterminism can cause serious errors. For instance, if two threads each compute an **int** stored in their private variable my_val and both try to add it to a shared variable x initialized to 0, the final result may be incorrect due to unpredictable execution order.

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

Time	Core 0	Core 1
0	Finish assignment to `my_val`	In call to `Compute_val`
1	Load `x = 0` into register	Finish assignment to `my_val`
2	Load `my_val = 7` into register	Load `x = 0` into register
3	Add `my_val = 7` to `x`	Load `my_val = 19` into register
4	Store `x = 7`	Add `my_val` to `x`
5	Start other work	Store `x = 19`

1. **Parallel Access to Shared Variable**: Both Core 0 and Core 1 are attempting to **update the same shared variable x** concurrently using their own private values my_val.
2. **Sequence of Operations**: Each core performs a sequence—**load x, load my_val, add, store the result**—which looks independent but affects the same memory location.
3. **Race Condition**: Since both cores operate on x at nearly the same time, the **final value of x depends on the order of execution**, leading to **unpredictable results**.

4. **Incorrect Outcome**: In the provided example, both cores read x = 0 initially. So, **one store overwrites the other**, and the correct result (x = 26) is lost—either x = 7 or x = 19 is stored.
5. **Definition of Race Condition**: This situation is called a **race condition**, where **two or more threads compete to update a shared resource**, and the result depends on the execution timing.
6. **Need for Atomicity**: The operation x += my_val must be **atomic**, meaning **no other thread can access or modify x until the operation is complete**.
7. **Critical Section**: To ensure atomicity, the x += my_val statement must be placed in a **critical section**, which allows **only one thread at a time** to execute it.
8. **Mutual Exclusion (Mutex)**: A **mutex (mutual exclusion lock)** is used to enforce that only one thread can enter the critical section at a time.
9. **Locking Mechanism**: A thread **must acquire the lock** before entering the critical section and **release it** after completing the update.
10. **Hardware and Software Support**: Mutexes are supported at the **hardware and OS levels**, and it's the **programmer's responsibility** to use them to avoid race conditions and ensure **thread-safe operations**.

This example illustrates why synchronization is essential in **shared-memory parallel programming** to prevent **unintended side effects and bugs**.

Thus to ensure that our code functions correctly, we might modify it so that it looks something like this:

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

Using a mutex ensures that only one thread can execute the statement x += my_val at a time, preventing race conditions. This does not enforce a specific order—either thread 0 or thread 1 may execute the update first. However, the mutex introduces **serialization**, meaning that part of the code becomes sequential. To maintain performance, it's important to **minimize** both the number and length of such **critical sections**.

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

In the given code, thread 1 remains in the busy-wait loop while (!ok_for_1) until thread 0 sets ok_for_1 = true. This approach is easy to understand and implement but inefficient, as the waiting thread keeps checking the condition without doing productive work, consuming CPU cycles. **Semaphores**, like mutexes, help with synchronization but offer more flexibility for certain situations. Monitors provide an even higher-level synchronization method by ensuring only one thread at a time can execute the object's methods.

Thread Safety

In most situations, parallel programs can use functions originally designed for serial execution without issues. However, exceptions arise when functions use static local variables, which, unlike regular local variables, persist between function calls and are shared among threads. This can lead to unexpected behavior if multiple threads access or modify the static variable simultaneously.

For instance, the C function *strtok* uses a static pointer to manage substring parsing across multiple calls, which makes it unsuitable for concurrent use by different threads. If thread 0 and thread 1 call *strtok* on different strings at overlapping times, the internal state could be overwritten, leading to incorrect or lost data. Such functions are not thread safe, and developers must be cautious when using serial-only functions in multithreaded programs to avoid data corruption or bugs.

2.3 Distributed memory

In distributed-memory programs, each core (or processor) can only access its own local memory and not the memory of other cores directly. To communicate between cores, message-passing is the most common approach, so most distributed systems use it. Although other APIs exist, message-passing is by far the most popular and widely supported. Interestingly, even on sharedmemory systems, programmers can simulate a distributed-memory setup by dividing memory logically and using message-passing libraries. Unlike shared-memory programs that use threads, distributed-memory programs typically run as separate processes because the cores may be on different machines with their own operating systems. There usually isn't a built-in way to start one process that automatically spawns threads across multiple machines, so separate processes are used instead.

Message-passing

A message-passing API provides essential functions like Send and Receive, allowing processes to communicate by identifying each other using ranks from 0 to p − 1, where p is the total number of processes. In the given pseudocode, each process starts by calling my_rank = Get_rank(); to find its rank.

If my_rank == 1, process 1 uses sprintf(message, "Greetings from process 1"); to create a message and then sends it to process 0 using Send(message, MSG_CHAR, 100, 0);.

If my_rank == 0, process 0 receives the message from process 1 using Receive(message, MSG_CHAR, 100, 1);. After receiving, process 0 prints the message with printf("Process 0 > Received: %s\n", message);.

This simple pseudcode illustrates how message-passing allows processes to communicate using ranks and message buffers.

```
char message [100];
 ...
my_rank  =  Get_rank  ();
if ( my_rank == 1) {
sprintf ( message , " G r e e t i ngs 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 ) ;  }
```

There are a few important points to highlight here. First, the program segment follows the SPMD model, where all processes run the same executable but perform different tasks based on their ranks. Second, the variable message refers to different memory locations in each process, which is why names like my_message or local_message are often used. Finally, we assume that process 0 can write to stdout, which is generally supported by most message-passing APIs, even if not explicitly stated.

The behavior of the *Send* and *Receive* functions can vary, and most message-passing APIs offer multiple versions of each. In the simplest case, a call to *Send* blocks until the matching *Receive* starts, meaning the sending process waits until the receiver is ready. Alternatively, *Send* might copy the message into its own buffer and return immediately after the copy is complete. Typically, *Receive* blocks the receiving process until the message is fully received. However, there are other variations for both *Send* and *Receive* depending on the specific API.

- Message-passing APIs usually offer many additional functions beyond basic send and receive.

- These include collective communication operations like **broadcast** (one process sends data to all) and **reduction** (combining results from all processes, e.g., summing values).
- APIs may also provide tools for managing processes and handling complex data structures.
- The most widely used message-passing API is the **Message-Passing Interface (MPI).**

Message-passing is a powerful and flexible approach for writing parallel programs, and it's used in nearly all applications running on the world's most powerful computers. However, it operates at a very low level, requiring programmers to manage many details manually. Parallelizing a serial program with message-passing often means rewriting most of the code, and data structures must either be replicated or explicitly distributed across processes. Because of this complexity and the difficulty of making incremental changes, message-passing is sometimes referred to as

"the assembly language of parallel programming," prompting efforts to create higher-level distributed-memory APIs.

One-sided communication

In message-passing, communication requires two processes: one to call a send function and another to call a matching receive function. Every exchange depends on the explicit involvement of both sides. In contrast, **one-sided communication—or remote memory access**—allows a single process to access or update data in another process's memory without needing that process to actively participate. This simplifies communication and can reduce costs by avoiding the synchronization and overhead involved in two-process coordination. It also saves resources by eliminating one of the function calls, either *send* or *receive*.

It should be noted that some of these advantages may be hard to realize in practice. For example

- If process 0 copies a value into process 1's memory, it must first ensure it's safe to do so to avoid overwriting important data.

- Process 1 also needs a way to know when the update has occurred.
- This can be managed by synchronizing both processes before and/or after the copy operation.
- Alternatively, process 0 can set a flag variable after copying, and process 1 can **poll** this flag to detect when the new data is ready.
- However, polling increases overhead, as process 1 must repeatedly check the flag until the copy is confirmed.
- Remote memory operations without explicit interaction can lead to subtle bugs that are difficult to detect and debug.

Partitioned global address space languages

Many programmers prefer shared-memory programming over message-passing or one-sided communication, so several groups are working on languages that bring shared-memory features to distributed-memory systems. However, this approach is more complex than it appears. Simply treating all memory in a distributed system as one large shared memory would lead to poor or unpredictable performance. This is because a process might access either local memory, which is fast, or remote memory, which can be hundreds or thousands of times slower. Efficient use of such systems requires careful management of where and how memory is accessed.

As an example, consider the following pseudocode for a shared-memory vector addition:

```
shared int n = . . . ;
shared double x[n], y[n];
private int i, my_first_element, my_last_element;
my_first_element = . . . ;
my_last_element = . . . ;

/* Initialize x and y */
. . .

for (i = my_first_element; i <= my_last_element; i++)
   x[i] += y[i];
```

1. The code defines two shared arrays x[n] and y[n], and assigns each process a specific portion of these arrays based on its rank using my_first_element and my_last_element.
2. After initialization, each process independently performs additions on its assigned section: x[i] += y[i].
3. If each process's assigned elements are stored in memory local to the core it's running on, the performance will be efficient.

4. However, if all elements of x are located on one core (e.g., core 0) and all of y on another (e.g., core 1), then accessing remote memory repeatedly during the addition will lead to significant performance degradation.

Partitioned Global Address Space (PGAS) languages offer features similar to shared-memory programming but give the programmer more control over data placement. Private variables are stored in the local memory of the core running the process, helping to reduce unnecessary remote memory access. Additionally, the programmer can explicitly define how shared data structures, such as arrays, are distributed across processes' local memories.