1 An Introduction to Parallel Computer Architecture

This chapter is not designed for a detailed study of computer architecture. Rather, it is a cursory review of concepts that are useful to understand the performance issues in parallel programs. Readers may well need to refer to a more detailed treatise on architecture to delve deeper into some of the concepts.[1,2]

[1] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 2017

[2] José Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann, 2003

1.1 Parallel Organization

Question: What are execution engines and how are instructions executed?

There are two distinct facets of parallel architecture: the structure of the processors, *i.e.*, the hardware architecture, and the structure of the programs, *i.e.*, the software architecture. The hardware architecture has three major components:

1. Computation engine: they carry out program instructions.
2. Memory system: they provide ways to store values and recall them later.
3. Network: it forms the connection among processors and memory.

An understanding of the organization of each architecture and their interaction with each other is important to write efficient parallel programs. This chapter is an introduction to this topic. Some of these hardware architecture details can be hidden from application programs by well-designed programming frameworks and compilers. Nonetheless, a better understanding of these generally leads to more efficient programs. One must similarly understand the components of the program along with the programming environment. In other words, a programmer must ask:

1. How do the multiple processing units operate and interact with each other?
2. How is the program organized so it can start and control all processing units? How is it split into cooperating parts and how

do parts merge? How do parts cooperate with other parts (or programs)?

One way to view the organization of hardware as well as software is as graphs (See Section 1.6 and Section 2.3). Vertices in this graph represent processors or program components, and edges represent network connection or program communication. Often, implementation simplicity, higher performance, and cost-effectiveness can be achieved with restrictions on the structure of these graphs. The hardware and software architectures are, in principle, independent of each other. In practice, however, certain software organizations are more suited to certain hardware organizations. We will discuss these graphs and their relationship later in the textbook.

Another way to categorize the hardware organization was proposed by Flynn [3] and is based on the relationship between the instructions different processors execute at a time. This is popularly known as Flynn's taxonomy.

[3] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9): 948–960, 1972

SISD: Single Instruction, Single Data

A processor executes program instructions, operating on some input to produce some output. An SISD processor is a serial processor. A single sequence – or stream – of instructions operates on a single stream of operands, producing a single output stream. Note that it does not preclude an instruction operating on multiple operands, meaning a small number of operands may be processed at each step. For example, two input numbers may be added to produce one sum. We treat such an operand-set as a single item in a stream of operands. Similarly, the results of the operation form a single output stream.

SIMD: Single Instruction, Multiple Data

A SIMD (often pronounced sim.dee) processor indicates multiple simultaneous operations of a kind. It describes an architecture with a single stream of operations but multiple streams of operands. At each step, one operation in the stream is repeated on operands from all data-streams simultaneously. For each data-stream, an output is produced. This presumes the availability of multiple execution units performing the operation on multiple streams in parallel. For example, each pair in eight pairs of numbers may be added and eight sums produced. Thus, there are as many output streams as input streams. Such operations are sometimes referred to as vector operations. (Usually, the number of data-streams is limited by the number of execution units available, but also see SIMT in the summary at the end of the chapter.)

MIMD: Multiple Instruction, Multiple Data

MIMD refers to a general form of parallelism, where multiple independent operations are performed by a number of processors, each processing operands from its own stream. Each processor produces its own stream of output as well. Since the processors remain effectively independent of other processors in MIMD architecture, there is no requirement that the processors execute their steps simultaneously or remain in synchrony.

MISD: Multiple Instruction, Single Data

The only other possible category in this taxonomy has multiple processors, each with a separate instruction stream. All operate simultaneously on the same operand from a single data-stream. This is a rather specialized situation, and a general study of this category is not common. (Sometimes, the same data-stream is processed by different processors, either for redundancy, or with differing objectives. For example, in an aircraft, one instruction stream may be analysing data for anomaly, while another uses it to control pitch, and yet another simply encodes and records the data.) These can often be studied as multiple SISD programs.

Modern parallel computers are generally designed with a mix of SIMD and MIMD architectures. SIMD provides high efficiency at a lower cost because only a single instruction stream needs to be managed, but when vector operations are not required, meaning there is an insufficient number of data-streams available, the execution engines can be underutilized.

Another useful taxonomy is based on memory connectivity. Memory[4] contains addressable *words*, or data items. Given an address, it can fetch the word or overwrite it. If all processors are connected to the same memory, we call it a shared-memory system or *shared-memory architecture*. These CPU-memory connections need not be direct point-to-point, but could be via one or more intermediate routers. Thus, some parts of memory may be accessed directly, while others are accessed through intermediaries. This makes for non-uniform access to different parts of memory and is called NUMA[5] memory architecture.

[4] Memory includes the storage as well as its controlling hardware, *i.e.*, memory controller

[5] ***Defined :*** NUMA = Non Uniform Memory Access

The alternative is *distributed-memory architecture*, in which different processors have access to their own separate memory. While it may be possible to access memory of other processors as well, such access must be made through instructions executed on that remote processor. In contrast, even for NUMA style shared-memory organization, a processor can communicate with all shared memory by executing only its own instructions. and does not need a cooperating processor

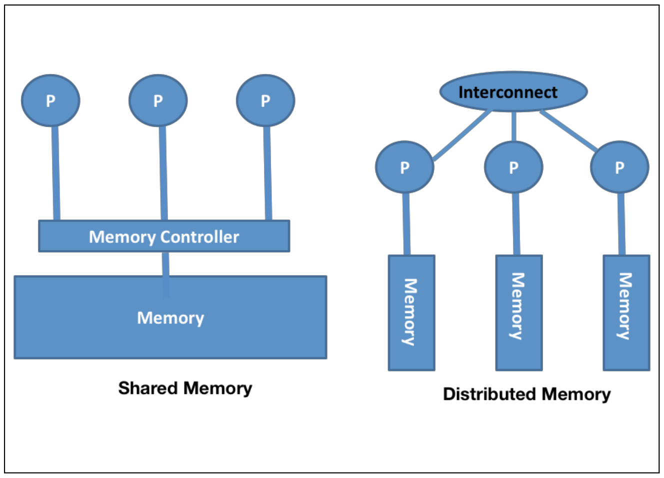

Figure 1.1: Shared-memory *vs* Distributed-memory architecture

to execute instructions on its behalf.

1.2 *System Architecture*

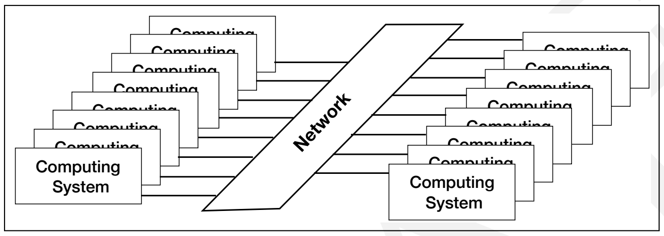

Figure 1.2: Parallel computing cluster

Highly parallel systems of the day have a hierarchical structure (see Figure 1.2). A cluster of computing systems are connected by a network. These systems are also called nodes. The network topology will be discussed later in this chapter. These systems usually each have their own operating system and name-space[6]. They may also share a common global name-space. The computing system itself contains a number of processors connected with each other in a more tightly-knit unit (See Figure 1.3). This may include central processing units (CPUs), Graphics processing units (GPUs), *direct memory access* (DMA) controllers, caches and an underlying memory system. A computing system is usually under the overall control of a single operating system, even though there may exist separate controllers for different components, each capable of executing independently from others[7]. Thus we have many processors within a single computing system as well. Further, multiple streams of data can be read from and written into the memory concurrently, and even in parallel.

[6] ***Defined :*** Name-space is a unique naming system where different objects do not have the same name or label. Labels across name-space are not necessarily unique.

[7] We do not delve into virtualization in this book, where cores and memory may be virtually partitioned into multiple nodes, each under the apparent control of a different operating system

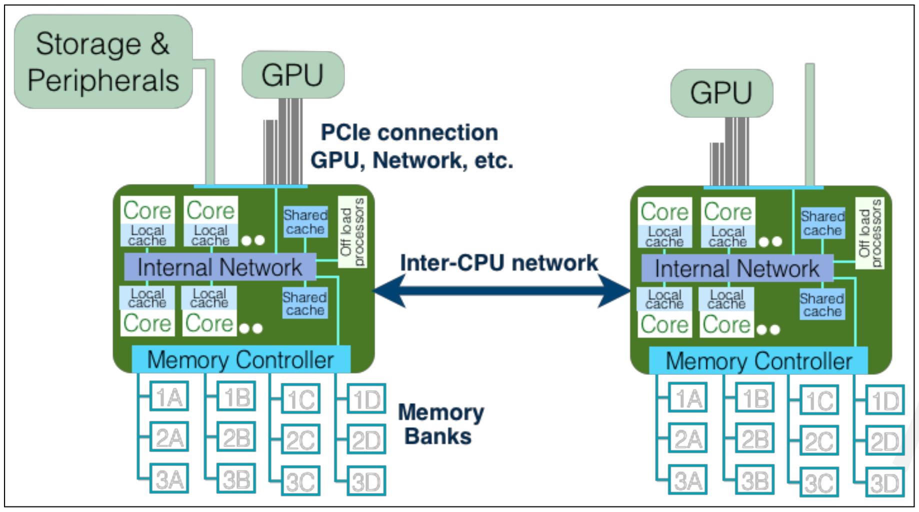

Figure 1.3: Computing system

Thus, both the cluster as well as a single node are common examples of the MIMD architecture.

1.3 *CPU Architecture*

We next focus on the computation core. It comprises registers[8] in addition to control and execution logic. Some registers are general purpose, and their addresses (or names) may be used in user programs. Others are for special purposes. Different parts of the core all perform their steps simultaneously in parallel and are synchronized to a CPU-wide clock. In principle, we may use this clock to measure time. In other words, at the same time would mean at the same clock tick or in the same clock cycle.

[8] ***Defined :*** A register is a small but fast local memory. A CPU has access to a small number of registers.

The core's main controller fetches streams of instructions and effects their execution, sometimes seeking assistance from other controllers. Instructions indicate the operations and the data on which to operate, *i.e.*, operands. Examples of operations are *add*, *read*, *write*, *branch*, etc. Operands are data or addresses, and provide input to the operation or store its output. These operands may be provided as literal values, or taken from a specified register or a specified memory address. The perpetual iteration of a core is as follows:

1. **Fetch** one or more instructions from the memory address stored in the program counter (PC) and update the PC. PC is a special purpose register, and is also called instruction pointer.

2. **Decode** one or more instructions to understand what operands and execution units are required and possible divide it into simpler sub-instructions (or micro-operations).

3. **Fetch** any required operands from memory into operand registers. Note that the operands of later instructions may become available earlier due to caches (see Caches later).

4. **Execute** the instruction on one of its appropriate execution units.

5. **Commit** or store output operand into memory or user registers if required.

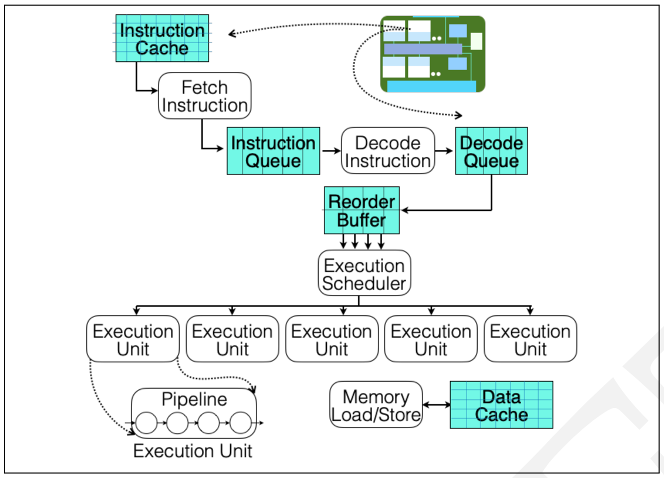

Figure 1.4: Computing core

The core's functional pipeline[9] is illustrated in Figure 1.4. Each stage of the pipeline passes its results on to the next stage on completion and immediately seeks its next task from the previous stage. The front-end of a core's controller fetches instructions from memory, decodes them, and schedules them on one of several execution units. These units are designed to perform logical and arithmetic operations on one or more pieces of data at a time. This front-end may fetch and interleave instructions from multiple code-streams[10]. A given code-stream's instructions are fetched usually in that stream's order. They are also fetched speculatively by predicting the outcomes of conditional branch instructions. A conditional branch continues execution sequentially as usual, or from a new address listed as an operand. The choice is determined by the value of a second operand associated with the instruction. For example, "BRANCH R1 R3" may reset PC to the address stored in register R1 if the value stored in R3 is 0.

[9] ***Defined :*** A pipeline is like an assembly line: a sequence of sub-operations that together complete a given operation.

[10] A code-stream may be thought of as a program.

The front-end stores the decoded instructions in order in a decode-buffer, which acts as a conduit to the execution engine. The execution engine allocates appropriate execution units to each instruction. It may re-order the instructions to achieve faster completion times. However, it completes, *i.e.*, retires, them in the order of the decode-buffer, committing the results of the execution. Multiple commits may occur in the same clock cycle. It is important to note that execution units themselves are pipelined, and a pipeline holds multiple instructions in its different stages simultaneously. However, an instruction cannot begin to be executed until its operands are available. Note that some input operand of an instruction may be the output of a previous one. Such operand is available only after that earlier instruction completes. The later instruction is said to *depend* on the earlier one as shown:

```
R1 = read address A;
R2 = read address B; // Independent of instruction 1: can begin before 1 is complete.
R3 = R1+R2; // Depends on instruction 1 and 2
```

And all that is only at a rather high level of abstraction. The point is that the architecture's details are intricate, but the following repercussions are important to note. The 'execution' of an instruction takes finite and variable time. Not only does a computing system have many cores, potentially executing different parts of the same program at any given instant, but each core also has multiple instructions in flight at any given time. These in-flight instructions do not necessarily follow each other sequentially through the various stages of the core's pipeline, but they retire sequentially. This parallel execution, or start, of multiple instructions in the same clock cycle is called *instruction level parallelism*.

From the discussion in this section, it should be clear that even a single core follows the MIMD principle at some level. It can indeed execute multiple instructions (on its multiple execution units) in the same step. Some of these execution units process only a single datastream and are examples of SISD. At the same time, some modern cores also contain execution units that are SIMD. Intel's AVX and AVX2 and nVIDIA's SMX are examples of such execution units.

1.4 *Memory and Cache*

Question: Where all is data stored? How is it fetched by various execution engines?

CPUs are invariably attached to large memory systems, which we sometimes refer to as the main memory. Main memory latency[11] is significantly larger than that of computation unit pipelines. Memory instructions are also processed, as shown in Figure 1.4. The execution of a memory *read* or *write* instruction started on a core does not complete for a relatively long period, possibly delaying the start of subsequent instructions.

[11] Latency is the time taken for an activity (like memory write) to complete since the time it is started (*i.e.* the write request is made).

Hence, it is common for hardware to maintain copies of a subset of the data in fast local memory, called *cache*. Indeed, an entire cache hierarchy – a series of caches – is maintained with an eye towards the cost. A cache too small may not be of much help, and a cache large enough to be helpful may be too expensive. Therefore the cache is often divided into multiple levels. Level 1 (L1 for short) cache is small but could have latency comparable to registers, with a high per-unit cost. Level 2 cache may be larger with a slightly higher latency and a slight lower per-unit cost, and so on. Often, higher-level caches are also shared by more cores.

If a given piece of data is in level i cache, it must also exist in level $i+1$. Thus, the same data has many proxies. The goal is to try and retain the frequently used data in lower levels, and to operate on that copy. Data re-use and locality of use within a program is a common reason why this is possible.

With a cache hierarchy, if a data item is not found in level i cache, *i.e.*, it is a *cache-miss*, it is allocated space in that cache. That space is populated by bringing the item from level $i+1$ (and recursively from higher levels if necessary). This means that any data previously resident in that allocated space in level i must be *evicted* first, possibly by updating its proxies at higher levels. The performance of a program's memory operations depends on the allocations and eviction policy. Some systems allow the program to control both policies. More often, though, a fixed policy is available.

For example, in *direct-mapped caches*, the cache-location of an item is uniquely determined by its memory address. Another item already occupying that location must be evicted to bring in the new item before the core can access it. In the more pervasive *associative caches*, an item is allowed to be placed in one of several cache locations. If all those candidate locations are occupied, one must be vacated to make space for the new item. The *cache replacement policy* governs which item is evicted. *FIFO eviction policy* (FIFO stands for First in first out) dictates that the item that came into the cache before other candidates is evicted. In *LRU eviction policy* (LRU stands for least recently used), the evicted cache entry is the one that was last accessed before every other candidate.

Even if a fixed policy is in effect, programs can be written to adapt to it. For example, a program may ensure that multiple SIMD cores that share a cache do not incessantly evict each other's data. Suppose direct-mapped cache addressing is used. In a cache with k locations, memory address m occupies cache location $m\%k$. This means that up to k contiguous memory items read simultaneously by k SIMD cores can co-exist in the cache. On the other hand, accesses to memory addresses A and $A+k$ conflict, and would evict each other.

When updating data resident in a cache, it can be written to its next higher level cache (*write-through* cache) before the write is considered complete. Alternate write policies also exist. For example, in *write-back* caches, data is written only into that cache level and the instruction completed. Writing to the higher cache levels is deferred until later. Write-back caches are simpler and complete updates faster, but can lead to harder cache coherence problems when memory is shared by multiple processors and multiple cache hierarchies.

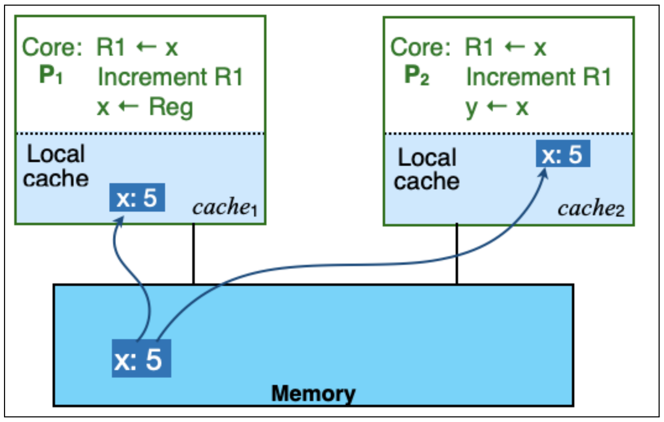

Figure 1.5: Cache coherence: two cores are shown with separate caches. R1 is a local register in each core. x and y are memory items.

Each cache level is divided into *cache-lines*: equal-sized blocks of contiguous bytes. The policies are implemented in terms of entire lines. So organizing a cache into lines helps reduce the hardware cost of the query about whether a data item accessed by a core is in that cache, *i.e.*, whether there is a cache-hit or a cache-miss. However, dealing in cache-lines means that an entire cache-line must be fetched in order to access a smaller memory item. This acts to prefetch certain data, in case the other items in that cache-line are accessed in the near future.

Caches impose significant complexity in parallel computing environments. Note in Figure 1.3 that each computing core has its own cache. These multiple cores may retain their own copies of some data, or write to it. This duplication can lead to different parts of the same program executing on those cores to see different – and hence inconsistent – data in the same memory location at the 'same time.' Such inconsistency is hardly surprising if each part assumes that there is only one data item in one memory location at a time. This consistency is called *cache coherence*. Coherence is maintained by ensuring that two cores do not modify their copies concurrently. If a core modifies its copy, other copies are invalidated or updated with the new value.

These updates cannot be instantaneous, meaning there are periods when the copies do not have the same values. However, it suffices to make them consistent before the next access of that memory item. If a memory item is updated through its proxies in multiple caches, those updates only need to be observed by the cores (*i.e.*, by readers executing on each core) to have been made in the same order. In other words, if a reader observes the update$A$ to have ocurred before update B to a location, no other reader may observe update B before update A.

Figure 1.5 demonstrates cache coherence. Cores P_1 and P_2 read x, whose initial value 5 is cached both in $cache_1$ and $cache_2$. P_1 now stores 6 in $cache_1$, which is propagated to $cache_2$. If the second access to x by P_2 happens before this update, it receives 5. Otherwise, it receives 6.

Recall that coherence ensures that any modification to an item is propagated to all its cached copies. The appearance is similar to the case where the item is directly accessed from the memory un-cached. This does not preclude two concurrent changes to an item leading to unpredictable results. For example, in figure 1.5, P_2 could write the value of its register R1 into x. This value would be 6 if P_2's *read* of x completes before P_1's *write* to x. P_1's increment would thus come undone. Furthermore, the interplay between cache-coherent accesses of two or more different items can also violate expectations that are routine in a sequential program. Such violations occur because the order in which updates to two items x and y become visible to one core is different from the order in which they may have been made,

We will later study this larger issue of memory-wide consistency in more detail in section 4.2. One must understand the type of memory consistency guaranteed by a parallel programming environment to design programs that execute correctly in that environment. In fact, some programming environments even allow incoherent caches in an attempt to bolster performance. After all, coherence comes at a performance cost. Such environments leave it to the program to manage consistency as needed. We will see such examples in Chapter 6.

Recall that caches operate in units of cache-lines, meaning coherence protocols deal in lines. If item x in a P_2's cache needs to be invalidated, its entire line — including items not written by P_1 — is invalidated. This is called *false-sharing* and is discussed in Chapter 6.

1.5 GPU Architecture

Graphics processing units, popularly called GPUs, are named so due to their historical roots in graphics processing. They nevertheless comprise general purpose parallel processors, which are used to acclerate parts of a program, and sometimes just to offload a subset of the work from the CPU. A computing system may have one or more GPUs, just as it may have one or more CPUs. CPUs on a system communicate through an inter-CPU network. GPUs may also communicate through an inter-GPU network. Finally, there is a third network connecting the CPUs to the GPUs. The design of a uniform and integrated structure for these networks are on the horizon, but multiple networks with divergent characteristics are common-place, and should be considered in parallel program design. The CPU-GPU network is usually much slower than the other two. A program that reduces CPU-GPU communication, then, would be more suited to this situation.

GPUs reside within a computing system and are usually connected to CPU cores through an internal PCI express network (see Figure 1.3). The general architecture of GPUs is shown in Figure 1.6. GPU cores are organized in a hierarchy of groups. GPU execution engines comprise SIMD cores. For example, one engine may consist of, say, 32 floating-point execution units, and all may be used to execute the next floating-point instruction in some instruction stream with its 32 data-streams. Just like CPU execution units, each core of the SIMD group consists of a pipeline of sub-units.

Similarly, another execution unit may cause *read* or *write* of, say, 32 memory addresses. GPUs have memory separate from the CPU memory. This memory is accessible by all GPU cores. Due to a higher number of concurrent operations, GPU memory pipelines tend to be even longer (*i.e.*, they are *deep pipelines*) than CPU's memory pipelines, even as the cache hierarchy may have fewer levels. On the other hand, GPU's execution unit pipelines are often shorter than CPU's, and the imbalance in GPU memory and compute latencies is significant. (See Section 6.5 for its impact on GPU programming.)

Stream Processors (SPs) are grouped into clusters variously called streaming multi-processors (SM), or compute-unit (CU). SPs within an SM usually share an L0 or L1 level cache local to that SM. In addition, SMs may also contain a user-managed cache shared by its cores. This cache is referred to as scratchpad, local data-share (LDS), or sometimes merely shared-memory. Sometimes, groups of SMs may be further organized into 'super-clusters,' for example, for sharing graphics-related hardware. Several of these super-clusters may share higher levels of cache. At other time, the processors of an SM (or CU) may be partitioned into multiple subsets, each subset operating in SIMD fashion. Thus, there is a hierarchy of cores and a hierarchy of caches. Again, due to the possible replication of data into multiple local caches, their coherence is an important consideration.

In terms of instruction execution, this GPU architecture is not substantially different from the CPU architecture shown in Figure 1.4. Only, there is a preponderance of SIMD execution engines in GPU but of SISD engines in CPU. The difference is larger in the organization of cores and the resulting design parameters. Much of this difference can be attributed to the fact the GPUs tend to carry many more execution units. They also are likely to have somewhat smaller memory and cache, particularly on a per-core basis. Many more simultaneous memory reads and writes need to be sustained by GPUs, and hence they need to lay a greater emphasis on efficient memory operations. The hierarchical organization of cores and caches into clusters aids this effort.

For example, each SM has a separate shared-memory unit (see block marked local cache in Figure 1.6), and each shared-memory unit may be further divided into several banks. Each bank of each unit can be accessed simultaneously. The SIMD nature of instructions allows a program to control the banks accessed by a single instruction and thus improve its memory performance. For example, a 32-core SIMD instruction could read up to 32 contiguous elements of an array in parallel if those elements reside in different banks. Similarly, all items accessed by an instruction could occupy the same cache-line.

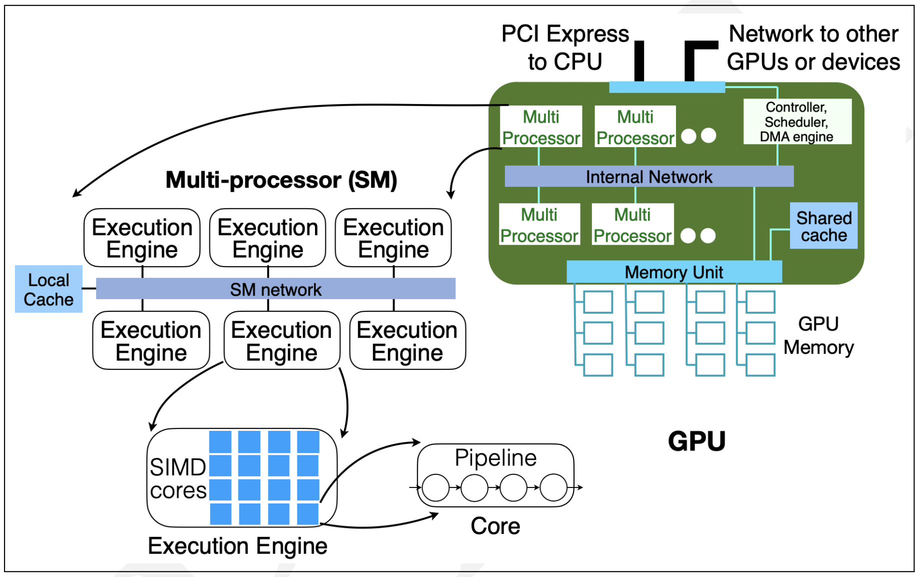

Figure 1.6: GPU architecture

1.6 *Interconnect Architecture*

Question: How is data communicated among execution engines?

A network inter-connects processors, which we can call network nodes. Note that these network nodes need not be units that directly execute program instructions, but they all have the ability to consume, produce, or collate data. The memory controller is an example. Multiple cores connect to the memory controller using a network. They send requests to the controller, whch returns the response after performing memory operations on the cores' behalf.

Sometimes, a unit contains multiple connections, each of which we may call end-points or *ports*. Transmission *links* connect the ports allowing messages to travel from one port to another. In general, the network structure can be represented as a graph, as discussed in section 1.1. Vertices of this graph may be the nodes themselves, or simply intermediate routers that forward data incoming on one link to another. Edges are the links. Networks containing such routers, or switches, are called switched or indirect networks. Such switches necessarily have multiple ports. On the other hand, nodes in a direct network themselves contain multiple ports (see Figure 1.9). It is common for general-purpose networks to employ modular design and populate ports into switches and employ switched networks. Internal, on-chip networks on devices like CPUs and GPUs can often be direct networks instead.

Routing

Messages are routed either using circuit switching or packet switching. For circuit switching, the entire path between the sender and the recipient is reserved and may not be shared by any other pair until that communication is complete. For packet switching, each switch routes incoming packets[12] 'towards' the recipient at each step. Sometimes switches are equipped with buffers to store and then forward packets in a later step. This is useful to resolve contention when two messages from two different sources arrive at the same time-step and are required to be forwarded onto the same link on the way to their respective destinations. One alternative is to drop one of the messages and require it to be re-transmitted. That is a high level overview. We will not discuss detailed routing issues in this book.

[12] ***Defined :*** A packet is a small amount of data. Larger 'messages' may be subdivided into multiple 'packets.' We will use these terms interchangeably.

Links

Most network topologies support bi-directional links that can carry data in both directions simultaneously. These are called *full-duplex links*. It is in many ways similar to having two *simplex links* instead – simplex links are unidirectional. In contrast, *half-duplex* links are bi-directional but carry data in only one direction at a time. In this section, we will not separately discuss the duplex variants of the described topologies, but it may be easier to understand the discussion assuming half-duplex links.

Types and Quality of Networks

The simplest network is a completely connected one: each node is directly connected to every other with a dedicated link (see Figure 1.7). However, if the number of nodes is n, up to $\frac{n(n-1)}{2}$ links would

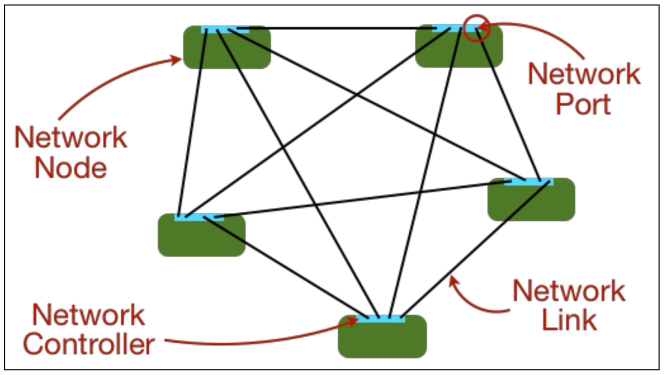

Figure 1.7: Completely connected network

be required in addition to $n-1$ ports per node. This is expensive. Another convenient interconnect is a bus (Figure 1.8): a pervasive channel to which each end-point attaches. This method is cost effective but hard to extend over large distances. Bus communication is also slowed by a large number of end-points, as only one end-point may send its data on a fully shared bus at one time, and some bus access arbitration is required for conflict-free communication.

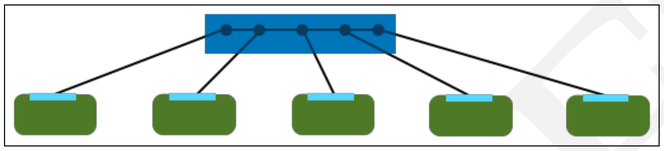

Figure 1.8: A Bus network

One measure of such conflict is whether a network is blocking. A nonblocking network exhibits no contention or conflict for any combination of sender-recipient pairs as long as no two senders seek to communicate with the same recipient. In other words, disjoint pairs can communicate simultaneously. A completely connected network is nonblocking, and a bus is blocking. Note that having separate paths between each pair is not required in packet-switched networks, but the independent paths do provide lower communication latency on average. Network latency is the time taken by a single packet to reach its destination.

There are many possible network designs. In general, as more links are added, less link-sharing is required, but the topology of the links can have a significant bearing on the types of traffic a network can handle well. An understanding of the available network's properties helps a programmer ensure that programs are designed to exploit its strengths and avoid its weaknesses. First, we need to a way to describe the properties of networks. The following metrics are useful:

1. Number of links: It is fair to assume that each link incurs a cost. Fewer links also imply simplicity of network layout. On the other

hand, the lack of links often leads to inefficient communication, and programs may benefit from reducing or batching communication.

2. Degree: The degree of a node refers to the number of links connected to it. It translates to the number of ports on the node. A large degree, particularly for computation nodes, increases network cost and complexity. It can also support higher concurrency among messages to different recipients. Formally, the degree of a network is the highest degree among its nodes. In common high-degree networks, many nodes have high degrees. However, extremes are possible, where only one or a few node have a high degree. For example, in a star network, a 'hub' is connected to every other node. Any programs on the hub in such cases have to account for its high degree. A hub can also be source of high contention, not unlike a bus.

3. Total bandwidth: The maximum rate at which data can be handled by the entire network is its bandwidth. For an l link network with link bandwidth b, the maximum network bandwidth is bl. However, in many practical networks, all links cannot be active at the same time, and the network bandwidth is usually smaller than bl. The bandwidth can be brought near bl with good routing protocols and contention resolution. Programs able to limit communication to the total bandwidth do not suffer from this bottleneck.

4. Minimum throughput: The maximum rate at which data can be sent between a given pair of nodes is called the pair's throughput. The minimum such throughput of any pair of nodes is called the minimum network throughput.

5. Diameter: The minimum number of links traversed by a message from node a to node b is the minimum path-length p_{ab} for that pair. The diameter of a network is the longest minimum path: $\max(p_{ij})$, among all node-pairs i, j in the network. The diameter indicates the maximum latency of messages. Programs that block while the communication completes can be severely limited by long latency.

6. Average path length: The lengths of paths between pairs of nodes can vary significantly from pair to pair. In such cases, the average path length is a useful metric. The minimum path-length between node-pairs, averaged across all pairs, is called the average network path length.

7. Bisection width: The minimum number of links that must be severed for one half of the nodes to be completely separated from the other half. While a high bisection width suggests robustness in the face of failing links, this metric also identifies communication bottleneck.

   Consider that in any equi-partition of n communicating nodes – each receiving as well as sending across half-duplex links in a given step. On average $\frac{n}{2}$ pairs would straddle partitions. Thus a bisection width less than $\frac{n}{2}$ is certain to block some pairs from communicating. Again, a bisection width of $\frac{n}{2}$ is not guaranteed to allow all pairs to proceed, as the network may be blocking, and there may be other conflicts along the paths between different pairs.

8. Bisection bandwidth: The minimum bandwidth available between two halves. As a global metric focusing on the bottleneck, the bisection bandwidth is generally more meaningful than the minimum bandwidth.

In addition to the above metrics, for switched networks, the number and complexity of switches (*e.g.*, the number of ports in each switch) are also important. Let us now evaluate a few common network topologies, particularly ones with a higher performance than the bus and a lower cost than the complete network.

Torus Network

A simple network that reduces the bus bottleneck is a ring (see Figure 1.9). Each node in a ring is connected to the next node in a wrap-around configuration. The two diagrams in Figure 1.9 show the direct and switched variants of the ring network. A ring uses n links to connect n nodes. With unidirectional communication, the latency can be as high as $n - 1$ with simplex, and $\frac{n}{2}$ with duplex links. The bisection bandwidth is also quite low: $2b$ for link bandwidth of b. We will improve these parameters by adding more links next.

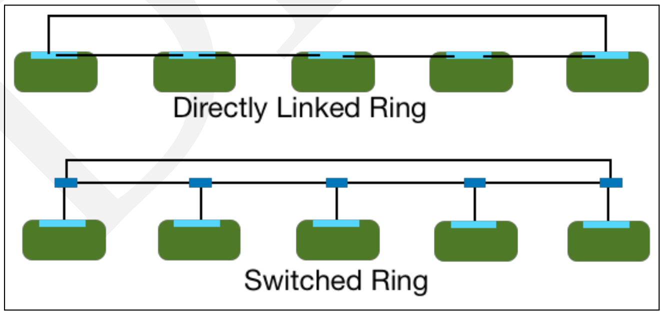

Figure 1.9: Ring network

A ring is a special case of the more general mesh or torus topology. A two-dimensional mesh is simply the nodes arranged in a 2D grid, with links connecting each node to the neighbors in its row as well as the neighbors in its column. Figure 1.10 shows a 3D mesh. If the corresponding nodes in extremal rows are linked to each other, and the extremal columns are similarly linked (as shown with dashed links in Figure 1.11), the network is called a 2D Torus. Ring is simply a 1D Torus.

A d-dimensional Torus network of n nodes has $k = \lceil \sqrt[d]{n} \rceil$ nodes along each dimension and dn links, each node having $2d$ ports. We call such tori k-way tori. The diameter of such a network is $\frac{kd}{2}$: the furthest node from a node is at a distance of $\frac{k}{2}$ along each dimension. The bisection width is $2k^{d-1}$: a $(d-1)$-dimensional slice through the middle would divide the nodes into two.

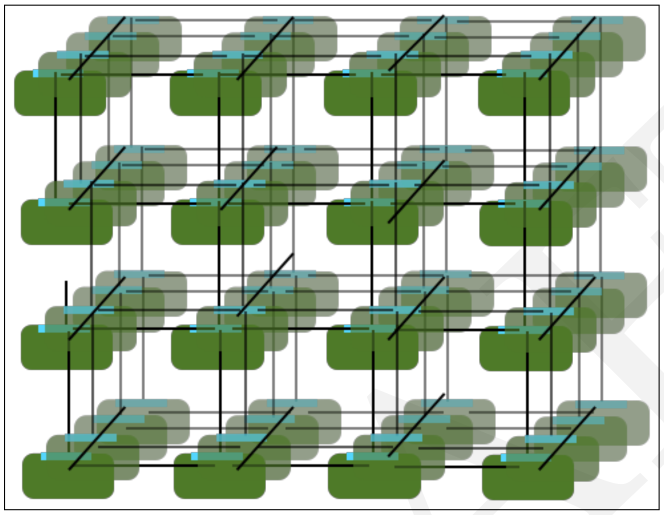

Figure 1.10: $4 \times 4 \times 4$ 3D Mesh

One benefit of the Torus is its short link lengths except for the wrap-around links – all dk^{d-1} of them. In the context of networks inside a chip, not only is the long delay in long links undesirable, variable delay in variable link lengths causes a significant impediment to speed and throughput. It is possible to lay tori out to alleviate the link length variability problem at a slight cost to the overall lengths. Figure 1.12 demonstrates one simple strategy, but we will not discuss these in detail here. Regardless, laying out high link counts, particularly on a plane, or on a few planar layers, or even in 3D, is quite a complicated.

Torus is a blocking network. Consider, for example, a message from node $(1, 1)$ to node $(2, 2)$ at the same time as a message from node $(2, 1)$ to node $(1, 2)$. Both must employ a common link (unless a longer path is taken but there may be similar conflicts on other edges). It is possible to create a nonblocking Torus network, but that requires many more links and additional switches.

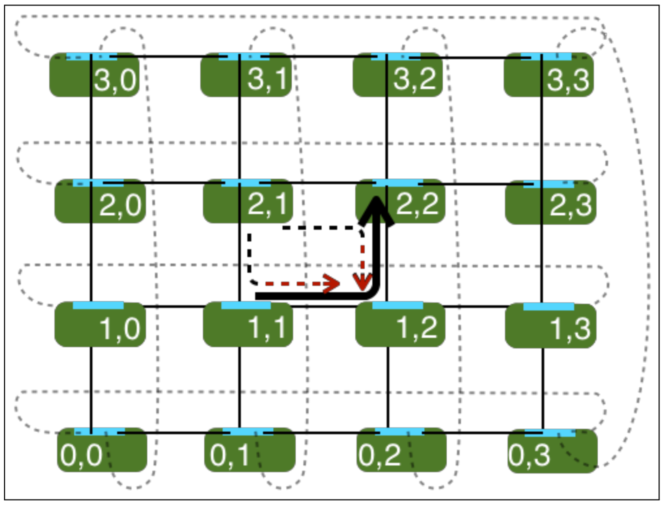

Figure 1.11: A 4×4 2D Torus, showing conflicting routes from node (1,1) to (2,2) and from node (2,1) to (1,2)

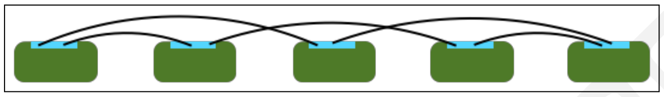

Figure 1.12: A 1D Torus layout with no long links

Hypercube Network

The Hypercube network [13] is an alternative to Torus. A Hypercube of dimension $d+1, d \geq 0$, is constructed by combining two copies of d-dimensional Hypercubes by mutually connecting by a link the i^{th} node of one copy to the i^{th} node of the other copy for all i (See Figure 1.13). A 0-dimensional Hypercube is a single node with no links and index 0. After combining, the nodes from one copy retain their previous index numbers and those from the other copy are renumbered to $2^d + i$, where i is a given node's previous index number. Thus, an n-node network is recursively constructed by adding $\frac{n}{2}$ links to two $\frac{n}{2}$ node networks.

[13] Jon S Squire and Sandra M Palais. Programming and design considerations of a highly parallel computer. In *Proceedings of the AFIPS spring joint computer conference*, May 1963

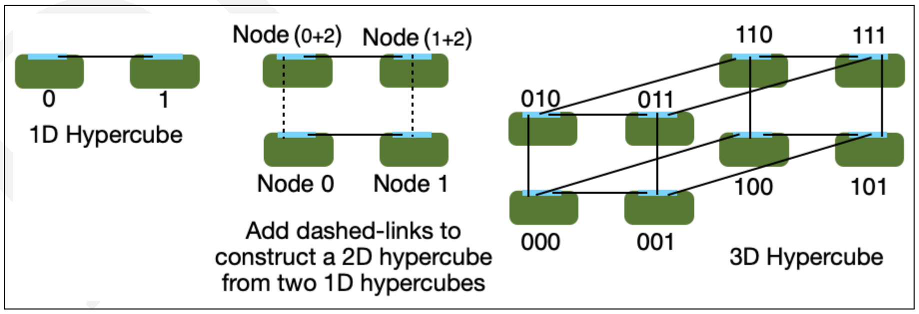

Figure 1.13: Hypercube network: Construction from 1-D to 2-D to 3-D Hypercubes is shown

By construction, a d-dimensional Hypercube has 2^d nodes. This restricts the number of nodes to a power of 2. It may be possible to delete some nodes and their links to allow any number of nodes, but the routing algorithm is simpler with the power of 2. The degree of each node is $\log n$ for an n-node network. This is somewhat high. The total number of links for a Hypercube of n nodes is $\frac{1}{2}(n \log n)$, and the bisection width is $\frac{n}{2}$. Hypercubes have a good performance at moderate cost and are a good choice for small to medium networks. For larger node counts, the node degree becomes unwieldy, and the power of 2 restriction precludes most node counts.

Cross-bar Network

The Cross-bar seeks to reduce the cost of the completely connected network. A Cross-bar switch has $2n$ ports connecting n nodes as shown in Figure 1.14.

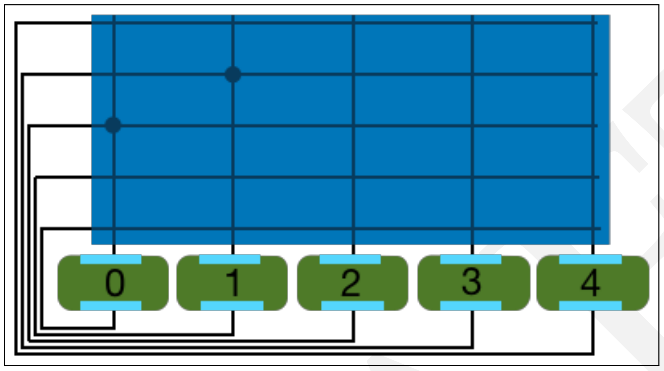

Figure 1.14: Cross-bar: dots designate that crossing wires are closed (meaning connected). Other crossings are open.

The Cross-bar switch can connect at most one pair of cross-wires in any row or column. For example, the connections depicted by dark circles in Figure 1.14[14] allow node 0 to communicate with node 2, at the same time node 1 communicates with node 3. Thus, up to $\frac{n}{2}$ separate pairs can be connected simultaneously, but one node can communicate with only one other node at a time. A Cross-bar requires $2n$ links to connect n nodes in addition to the $2n$ port Cross-bar switch. The bisection width is n. If the link bandwidth is b, the bisection bandwidth is nb. Cross-bar switches are also expensive due to the need for n^2 cross-wire connector switches. The data must also be able to travel larger distances on links with increasing n.

[14] In these diagrams, wire crossings do not depict junctions, except when a circle indicates a connection

Cross-bar is a nonblocking switch. Each source owns its column, and no other source may set any junction in its column. Similarly, every destination owns its row and no junction in row d is set unless d is a destination. Thus the junction in row s and column d is reserved for the exclusive use of s to d communications.

The complexity and expense of the Cross-bar can be ameliorated by using a modular multi-stage connector at the expense of latency.

Shuffle-exchange networks are one such class of multi-stage switches.

Shuffle-exchange Network

Shuffle-exchange networks are of many types. Let us consider an example. Omega network is a multi-stage network, as shown in Figure 1.15.

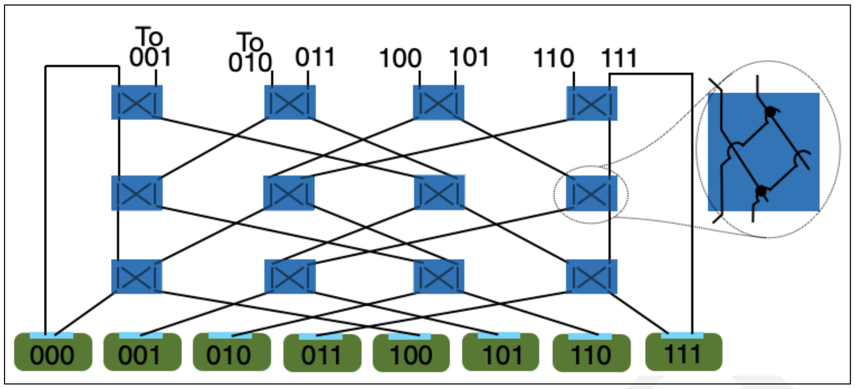

Figure 1.15: Omega network: Output of switches are shuffled into the inputs at the next stage. The first half of the links connect consecutively to the left input ports of the next level switches. The second half connect to the right ports.

All the switches used have a pair of input and a pair of output ports. Each switch can be separately controlled to either let both its inputs pass-through straight to its corresponding outputs or to swap them. This is really a 2×2 Cross-bar, also called a Banyan switch element, as shown in the inset in Figure 1.15 (although other implementations are possible). In the figure, cross-connects (or exchanges) are set up for swap (*i.e.*, cross). Connecting the other diagonal junctions instead would result in pass-through (*i.e.*, bar).

An n-node omega network requires $\log n$ stages[15], with $\frac{n}{2}$ switches per stage. The output of a stage is shuffled into the input of the next stage – the left half of the links connect consecutively to the left input of each switch and the right half of the links connect to the right input of consecutive switches. In other words, if we number the output from left to right, output i, for $i < \frac{n}{2}$, connects to switch i. Output i, for $i \geq \frac{n}{2}$, similarly connects to the right input of switch $\frac{i}{2}$.

[15] Logarithm base 2 is implied in this book.

Omega networks are examples of a family of multi-stage shuffle-exchange networks[16] like Butterfly[17] or Benes[18]. Different members of the family mainly have different shuffle patterns. Figure 1.16 shows a butterfly topology, for example. It contains $(\log n - 1)$ shuffle stages consisting of n exchange switches each. This leads to a slightly lower diameter than Omega ($\log n$ *vs* $\log n + 1$) and a higher bisection width ($2n$ *vs* n) at the cost of almost doubling the number of links ($2n \log n$ *vs* $n \log n + n$). One practical advantage of Omega network is that the shuffle pattern does not change from stage to stage allowing a more modular design. Also note that although the diagrams appar-

[16] H. S. Stone. Parallel processing with the perfect shuffle. *IEEE Trans. Comput.*, C-20(2):153–161, 1971

[17] Thomas J. LeBlanc, Michael L. Scott, and Christopher M. Brown. Large-scale parallel programming: experience with bbn butterfly parallel processor. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1988

[18] V. E. Benes. *Mathematical Theory of Connecting Networks and Telephone Trafic.* Academic Press, 1965

ently show uni-directional data flow, it does not have to be. This is demonstrated later in this chapter.

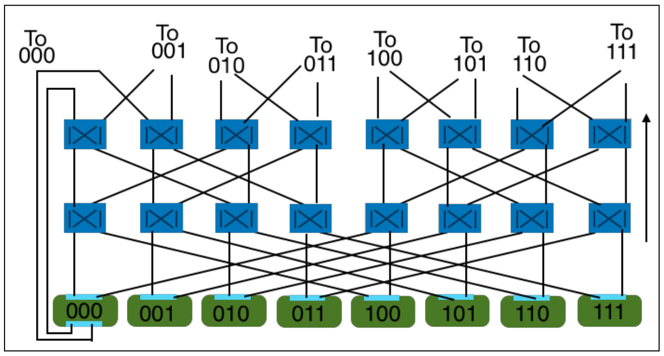

Figure 1.16: Butterfly Network

Clos Network

Clos networks take a different approach to reduce the cost of the Cross-bar. The main idea is to reduce the size and complexity by dividing the ports into smaller groups, say of size k, and use a Cross-bar within the smaller groups as shown in Figure 1.17. In a way, Clos is also a generalization of the shuffle-exchange network. Recall that exchange is but a 2×2 Cross-bar. Clos allows larger cross-bars. In this three-stage network, the shuffle is a perfect r-way shuffle, for a chosen r. The i^{th} output of switch j is connected to the j^{th} input of switch i of the next stage.

The bottom stage uses $k \times l$ cross-bars. The middle stage uses $r \times r$ cross-bars, $r = \lceil \frac{n}{k} \rceil$. The top stage uses $l \times k$ cross-bars. Clos has shown[19] that if $l \geq 2k - 1$, this network is nonblocking, retaining the contention-free routing of the Cross-bar. For a large number of ports n, Clos network requires multiple but significantly smaller cross-bars than a full $n \times n$ Cross-bar at the cost of a few more links. For example, a $1,024$ node Cross-bar requires $1,048,576$ cross-connects. In contrast, we can use 64 16×31 cross-bars in the first stage, 31 64×64 cross-bars in the second stage and 64 31×16 cross-bars in the third stage for a total of only $190,464$ cross-connects and $1,984$ additional links, albeit of smaller lengths than those inside a 1024×1024 Cross-bar.

[19] Charles Clos. A study of non-blocking switching networks. Technical Report 2, Bell Labs, 1953

Tree Network

One of the simplest networks to design and route is a binary tree as shown in Figure 1.18. Possibly, the root can be removed and its two children directly connected. Network complexity is small. The link count is only $2n - 3$ for n nodes. Switches are simple three-port connectors able to route between any two ports in one step. The tree

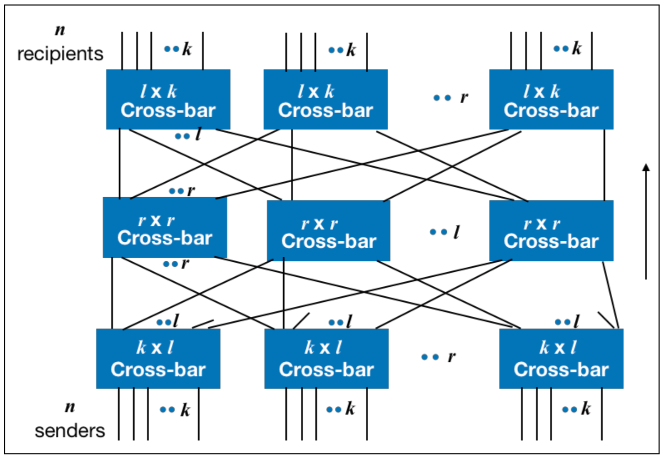

Figure 1.17: A Clos network

network has major bottlenecks owing to this simplicity. For example, the bisection width is just 1 – the link at the root can be severed to partition the nodes into equal halves. The diameter is $(2\log n - 1)$.

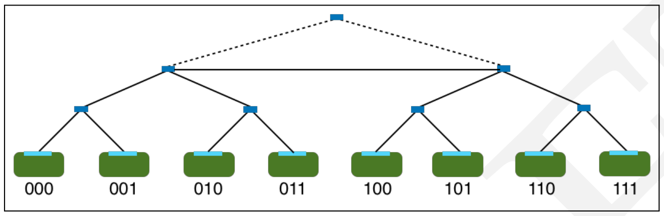

Figure 1.18: Tree network

Some of these bottlenecks can be improved by recognizing that the bottlenecks are worse near the top of the tree. The root switch must be used by all traffic going from the left subtree to the right subtree and *vice versa*. This problem can be addressed by adding more links at the higher levels of the tree. For example, double the number of links going up at the level above the leaf, quadruple above that, and so on. See Figure 1.19. So modified, it is called the Fat tree network. Of course, the tree need not necessarily be a binary tree but may have any degree $d > 1$.

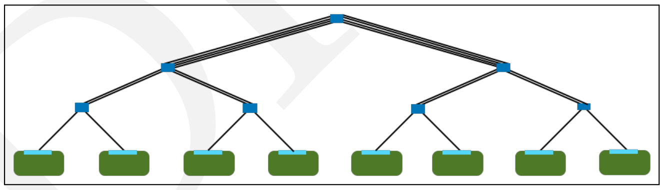

Figure 1.19: Fat tree network

Surprisingly, the Fat tree is a close relative of the Clos network. Let us quickly revisit the Clos network just described, where Clos network is presented as a uni-directional network communicating data upwards. Three stages are shown. Now, suppose we allow the links to be full-duplex and fold the figure down the middle, as shown in Figure 1.20. The middle stage $r \times r$ cross-bars of Figure 1.17 now looks like the top row of Figure 1.20. On folding, this row now has r 'output' links on the same side as r 'input' links, making $2r$ full-duplex links. All l switches at this stage are folded. After folding, the cross-bar in the top and the bottom stages of Figure 1.17 occupy the bottom row of Figure 1.20. Thus the two rows of Figure 1.20 together make for a root node with $2n$ links. Given duplex links, the network becomes symmetric. Any port can send or receive in a given step – possibly both if the links are full-duplex.

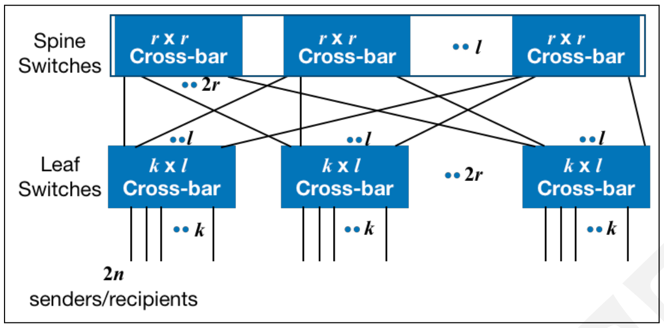

Figure 1.20: Folded Clos Network

We can re-interpret Figure 1.20 as demonstrated in Figure 1.21, integrating the l $r \times r$ cross-bars of the top row into single node with l links to each of the $2r$ nodes in the bottom row. The topology reduces to that of a three-level Fat tree topology. The root's degree is $2r$. and that of each switch in the bottom row is k. In this configuration, the root is often referred to as the spine and its children as leaf switches. If $l = k$, k links between each leaf switch and the spine are sufficient to support the combinded throughput of the leaf's k children.

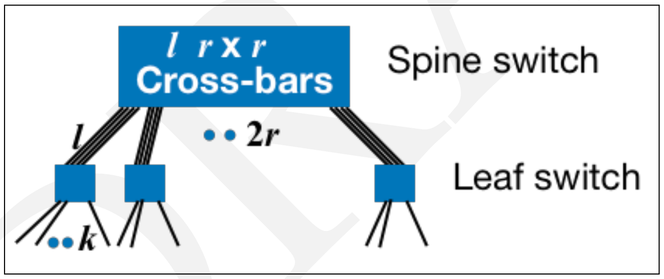

Figure 1.21: Clos = Fat tree network

Network Comparison

We have discussed a few popular network topologies in this section. Each has its pluses and minuses. Broadly speaking, the performance increases with increasing complexity and cost. Ideally, these details are hidden from a parallel application programmer, whose main concern should be to send data from a producer to a consumer. Nonetheless, certain topologies are more suited to specific communication patterns than others. A smart program exploits the knowledge of the underlying topology to situate consumers and producers in nodes that are more likely at a short communication distance. In general, shorter communications also incur – and cause – fewer conflicts. The following table lists some of the network metrics for selected topologies.

Table 1.1: Network Comparison

Network	Link count	Diameter	Bisection width
Completely connected	$\frac{n(n-1)}{2}$	1	$\frac{n}{2} \times \frac{n}{2}$
d-dimensional k-way Torus, $k^d = n$	nd	$\frac{dk}{2}$	$2k^{d-1}$
Fat Tree (Binary)	$n \log n$	$2 \log n$	$\frac{n}{2}$
HyperCube	$\frac{n \log n}{2}$	$\log(n)$	$n/2$

1.7 *Summary*

Parallel processors are ubiquitous. These include CPUs with near 10 or 20 cores, GPUs with a few thousand cores, and clusters with up to a million cores and more. Each core usually accepts a sequence of instructions and executes them ostensibly in that order. Each instruction may execute on a single set of operands (scalar operation) or an array of them at a time (vector operation). Some of these instructions read from or write to memory locations. Cores communicate with other cores through these memory operations. In some systems, communication ports connect cores to help them communicate. Often, cores communicate with these ports through special memory operations.

A parallel program that runs on multiple cores decides:

1. which core executes what instructions on what operands (and in what order)

2. which cores communicate what data with which other cores

This decision should account for heterogeneity: varying network characteristics between pairs of cores, or the difference in capabilities of the cores. At the same time, the interference of the cores' operations on each other has a major impact on the overall efficiency of a parallel program. For example, two cores sharing a common cache would impact the cache-hit rates. A core attempting to communicate with another must wait if that other core is busy completing a different operation. This chapter introduces the architectural framework for these decisions. The key lessons include:

- Sequential execution of instruction on a core is usually pipelined. Systems commonly allow out-of-order execution of certain instructions as long as the result is consistent with ordered execution.
- Overlapping the execution of multiple instructions allows better utilization of hardware components because they can be simultaneously busy operating on different parts of a program.
- Parallel MIMD cores execute independent instructions. SIMD cores all execute the same instruction simultaneously on different data.
- Somewhat refined terminology is also in vogue. SIMT – single instruction multiple threads – architecture allows a variable number of virtual cores to apparently execute an instruction 'simultaneously.' If the available number of physical cores is smaller than the requested SIMT-width, each SIMT instruction is serialized into multiple SIMD instructions.
- Similarly, SPMD (single program multiple data) and MPMD (multiple programs multiple data) are variants of MIMD, except the definition works at the level of the entire user program, rather than that of individual instructions. For example, in an SPMD architecture, the same program is executed on multiple cores, and at each core it operates on its own data. The executions together solve a problem. It is up to the program to determine at the time of execution which part of the solution each core undertakes.
- Processors may share memory, or maintain private memory whose data is communicated using a network.
- Memory-caches are used to improve access times to a subset of data. These caches may be directly managed by the user program or transparently managed by the underlying system.
- Different parts of a program may share caches and may interfere with residency of each other's data in the shared cache.
- In a parallel computing system, cores may be hierarchically organized into groups (and subgroups). Groups may have different architecture and capabilities from each other. For example, a computing system in a networked cluster of systems may comprise a group of CPU cores, all sharing a common memory, a group of GPU cores sharing another memory, and a network between the two groups.

- Large networks have a structured topology, such as tree, torus, or cube.
- Network latency and throughput are not necessarily uniform in a network connecting many processors. For example, in a network, processor P_1 may have a shorter communication distance to processor P_2 than any other processor. If more communication happens between topologically close-by pairs than between far-away pairs, the total communication time can be lower.

Textbooks on parallel architecture [20] are good sources for a deeper study of these topics. GPU architecture evolves at such a fast paces that any textbook [21] quickly becomes out of date. However, architecture vendors usually release white papers and programming guides, which are up-to-date sources of detailed information. A detailed analysis of design and performance issues of interconects have been discussed in several books [22].

Large scale computing systems have many components. They add up to large power consumption. That is a major concern in high-performance computing. A large number of components also translates into a large chance of failure: even one component failing could abort a long-running program if the failure is not handled. Significant effort is devoted to designing low-power and fault-tolerant architecture. Programs designed to take advantage of these features can reduce power consumption and can respond to certain failures. These topics are out of the scope of this book, but several overviews of recent techniques have been published [23]. Please refer to these to learn about such topics.

[20] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 2017; Smruti R. Sarangi. *Computer Organisation and Architecture*. McGraw Hill India, 2017; and Kai Hwang. *Computer Architecture and Parallel Processing*. McGraw Hill Education, 2017

[21] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010; and Nicholas Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley Professional, 2013

[22] F. Thomson Leighton. *Introduction to parallel algorithms and architectures: Arrays Trees Hypercubes*. Morgan Kaufmann, 1992; José Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann, 2003; and Sudhakar Yalamanchili. *Interconnection Networks*, pages 964–975. Springer US, Boston, MA, 2011. ISBN 978-0-387-09766-4

[23] Sparsh Mittal. A survey of architectural techniques for near-threshold computing. *ACM Journal on Emerging Technologies in Computing Systems*, 12 (4), 2015; Sangyeun Cho and Rami Melhem. On the interplay of parallelization, program performance, and energy consumption. *Parallel and Distributed Systems, IEEE Transactions on*, 21:342–353, 04 2010; Kenneth Obrien, Ilia Pietri, Ravi Thouti Reddy, Alexey L Lastovetsky, and Rizos Sakellariou. A survey of power and energy predictive models in hpc systems and applications. *ACM Computing Surveys*, 50(3), 2017; K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang. Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures. In *2012 41st International Conference on Parallel Processing*, pages 48–57, 2012; Ifeanyi P. Egwutuoha, David Levy, Bran Selic, and Shiping Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3): 1302—-1326, 2013; and Martin Radetzki, Chaochao Feng, Xueqian Zhao, and Axel Jantsch. Methods for fault tolerance in networks-on-chip. *ACM Computing Surveys*, 46(1), 2013

Exercise

1.1. What is the NUMA memory configuration?

1.2. What is false-sharing?

1.3. What are the reasons multiple levels of cache may be employed?

1.4. Does memory in a UMA configuration with four attached cores require four ports for the four cores to attach to? If a single port exists, how can four the cores connect to the same port?

1.5. What is the instruction fetch-commit pipeline?

1.6. Once the instruction is decoded, operands require fetching from memory. This fetch takes variable time (due to cache misses, coherence protocols, memory contention, etc.). Thus, operands of instruction $i+1$ may arrive before those of instruction i do. What is the condition in which instruction $i + 1$ may be started before instruction i?

1.7. Consider the conditions when instruction i and instruction $i + 1$ of a sequential program may be swapped by a compiler without affecting the result. Can they be swapped also if another program may access the same memory? Explain with an example.

1.8. Suppose the execution of an instruction is divided into 7 pipelines stages: $stage_1$ to $stage_7$. Each stage is able to complete its operation in a single clock-cycle. What is the instruction latency? Suppose a new instruction may starts only two clock-cycles after the previous one does. What is the maximum execution throughput?

1.9. What is the difference between a switch and a port?

1.10. What is the role of instructions execution engines in a switched network?

1.11. All SIMD cores perform the same operation in any clock cycle. However, branches can complicate this. For example, consider a group of 32 SIMD cores executing the following program. (id is the core number in the range 0..31).

```
1  int aincr = b[id] - a[id];
2  int s = sign(aincr); // s is 0 if aincr is +ve and 1 if it is -ve
3  if(s) {
4    b[id] = b[id] + bfactor;
5    a[id] = a[id] + afactor * aincr;
6  } else {
7    a[id] = a[id] - afactor * aincr;
8  }
```

All cores can execute the test on line 3 and then branch to their corresponding lines (4 or 7), depending on the result of the test. However, if some cores take the branch to line 4 and others to line 7, they have different instructions to execute next. The groups execute them taking turns. In each turn, the non-executing subset remains idle. In the example above, the first group executes line 4 and the second execute lines 7 and 8, leading to a total of six separate instructions.

Rewrite the code so cores do not separate (or diverge) into grops, and all execute only five lines in total. Assume that the multiplication and addition on a line can be performed together in one instruction.

1.12. Which of the following two pieces of code is more cache-friendly (meaning they use caches well), assuming that the matrix `mat` is layed out in the row-major order in memory.

```
for(int r=0; r<m; r++)     for(int c=0; c<n; c++)
   for(int c=0; c<n; c++)   for(int r=0; r<m; r++)
      mat[r][c] *= factor;    mat[r][c] *= factor;
}                                    }
```

1.13. Assume a single level cache with a cache-line of 16 integers. What is the total number of memory operations performed in the following code? What percent of those operations are cache-hits? Assume the cache holds 1024 lines, and there is only one processor. Assume direct-mapping of addresses, such that the integer at index i always maps to the cache location $i\%160$, given that the cache can hold up to 160 integers. (This would be in the cache-line number $(i\%60)/16$.)

```
void func(int *a, int *b) {
   for(int i=0,j=16; i<40; i++,j+=8)
      a[i] += b[j];
}
```

1.14. What is the hit-rate in exercise 1.13, if the cache can only hold 10 lines. Assume FIFO eviction policy.

1.15. Let two cores, respectively with id = 0 and id = 1, share memory but maintain their own caches as described in exercise 1.14, with 10 cache-lines each and using direct-mapping with FIFO eviction policy. What is the maximum and the minimum cache-hit rate for the following code in each core?

```
void func(int *a, int *b) {
   for(int i=0,j=16; i<40; i++,j+=16)
      a[id] += b[j*id];
}
```

1.16. Explain the statement: "CPU instruction to send data to GPU is executed with the help of DMA controllers."

1.17. A GPU has a single address on the CPU PCI-express network, to which CPUs may send instructions and data. Recall that a GPU has many SIMD-units, meaning different SIMD-units may execute different instructions. In the SPMD model, a single program, *i.e.*, a single set of instructions, is sent to the GPU by one CPU, and all SIMD-units execute this program at their own pace.

What are the reasons the progress of these SIMD-units can get arbitrarily out of pace with each other?

1.18. One way to increase the memory bandwidth is to have many memory units so that each can be read in parallel. For example, the SM on a GPU may contain 32 banks to support a 32-wide SIMD instruction. All banks are accessible to each SIMD-core so that they can share memory. However, only one data item can be requested from one bank in a single clock-cycle. If two different items are requested by two cores in the same cycle, these requests are in conflict, and they are issued serially. Assume that `data[i]` resides in bank `i`%32, Also sssume that `i, j,` and `tmp` are local to each core and reside, respectively, in three of the registers of that core.

```
int data[][33]
   for(int i=0; i<32; i++)
   for(int j=i+1; j<32; j++)
      int tmp = data[i][id];
      data[i][id] = data[id][i];
      data[id][i] = tmp;
}
```

Prove that the code above causes no bank conflict.

1.19. Assign addresses to nodes on a cube-network so that any two nodes directly connected by a link have addressed that differ in exactly one bit. Suppose the neighbor that is different in bit i is called neighbor i.

Now devise a routing algorithm to send a packet from the node with address A to that with address B. Assume that the routing always takes the shortest path. What is the maximum number of links traversed by any packet?

1.20. Consider the addressing scheme shown in Figure 1.18 but for a tree network with degree 32. Devise the routing algorithm. Find the maximum number of links traversed by any packet.

1.21. For the tree network in Exercise 1.20 find the maximum network latency observed if each device sends one packet to one other device. Assume that a packet takes one time-step to traverse each link. Two packets may not traverse the same link at any time-step. In addition, a node may only perform a single operation on a single link at one time-step. Thus, it may accept one packet at any time-step on one of its links, or send send one out on one link.

1.22. For a $16 \times 16 \times 16$ 3-D Torus network, where each node is addressed with ite 3-D coordinate (i, j, k), devise the routing algorithm to send a packet from node (i_s, j_s, k_s) to node (i_d, j_d, k_d) using the shortest route.

1.23. Show that the Butterfly network shown in Figure 1.16 is equivalent to an Omega network.

1.24. Reimagine the folded Clos network in Figure 1.20 with $2r \times 2r$ crosbars at the spine level and $(l + k) \times (l + k)$ cross-bars at the leaf level so that incoming data on any full-duplex link can be routed to any other. Design a nonblocking network supporting 648 computing systems. You may choose appropriate r, l, and k.