This chapter is not designed for a detailed study of computer architecture. Rather, it is a cursory review of concepts that are useful for understanding 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.\(^1\)

### 1.1 Parallel Organization

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

1. Computation engine: it carries out program instructions.
2. Memory system: it provides ways to store values and recall them later.
3. Network: it forms the connections 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

\(^1\) Hennessy and Patterson, *Computer Architecture*. Duato et al., *Interconnection Networks*. 
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 Sections 1.6 and 2.3). Vertices in these graphs 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 starting in section 2.3.

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

**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 [single instruction, multiple threads] 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 analyzing 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 (see Figure 1.1). Memory\(^3\) 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 nonuniform access to different parts of memory and is called NUMA\(^4\) memory architecture.

---

\(^3\)Memory includes the storage as well as its controlling hardware, that is, memory controller.

\(^4\)Defined: NUMA = Non-Uniform Memory Access
4 Introduction to Parallel Programming

The alternative is distributed-memory architecture, in which different processors have access to their own separate memory. While it may be possible to access the 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 the shared memory by executing only its own instructions and does not need a cooperating processor to execute instructions on its behalf.

1.2 System Architecture

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 in Section 1.6. These systems usually, each, have their own operating system and name-space. 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)\(^6\) 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.

\[\text{Figure 1.3} \quad \text{Computing system}\]

Thus, both cluster as well as 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.

\(^7\) 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.
The core’s main controller fetches streams of instructions and affects their execution, sometimes seeking assistance from other controllers. Instructions indicate the operations and data on which to operate, which we call operands. Examples of operations are *add, read, write, branch*, and so on. 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. For example, “ADDM R1, R2, 3” may mean “considering the value in general register R2 as an address, fetch the value from that memory address, add 3 to it, and store the result in R1.” 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 increment 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 optionally divide it into simpler sub-instructions (or micro-operations).

3. **Fetch** any required operand from memory into operand registers. Note that the operands of later instructions may become available earlier due to caches (see Caches in Section 1.4).

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 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 the 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. 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 the PC to the address stored in register R1 if the value stored in R3 is 0.

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 reorder the instructions to achieve faster completion times. However, it retires, meaning completes, 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 consist of pipelined sub-units holding multiple instructions in 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 the earlier instruction is completed. The later instruction is said to depend on the earlier one as shown:

1. \( R1 = \text{read address } A; \)
2. \( R2 = \text{read address } B; \) // Independent of instruction 1: can begin before 1 is complete.
3. \( R3 = R1 + R2; \) // Depends on instruction 1 and 2

The description above is only at a rather high level of abstraction. Architectural 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.

[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.
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 data-stream 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

CPUs are invariably attached to large memory systems, which we sometimes refer to as the main memory. Main memory latency 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 that started on a core is not completed for a relatively long period, possibly delaying the start of subsequent instructions.

Hence, it is common for the hardware to maintain copies of a subset of the data in fast local memory, called a cache. Indeed, an entire cache hierarchy – a series of caches – is maintained with an eye on 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 a latency comparable to registers, with a high per-unit cost. A level 2 cache may be larger with a slightly higher latency and a slightly 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 the 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 the lower levels, and to operate on that copy. Data reuse 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 the level \( i \) cache, meaning 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.

---

11 Latency is the time taken for an activity (like memory write) to complete from the time it is started (i.e. the write request is made).
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 the 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 a direct-mapped cache addressing is used. In a cache with $k$ locations, the memory address $m$ occupies the cache location $m \mod k$. This means that up to $k$ contiguous memory items can coexist in the cache and can be read simultaneously by $k$ SIMD cores. 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 is 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.

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, that is, whether there is a cache-hit or a cache-miss. However, dealing in

![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](image)
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 occurred 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 be 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