2.3 Dichotomy of Parallel Computing Platforms

In the preceding sections, we pointed out various factors that impact the performance of a serial or implicitly parallel program. The increasing gap in peak and sustainable performance of current microprocessors, the impact of memory system performance, and the distributed nature of many problems present overarching motivations for parallelism. We now introduce, at a high level, the elements of parallel computing platforms that are critical for performance oriented and portable parallel programming. To facilitate our discussion of parallel platforms, we first explore a dichotomy based on the logical and physical organization of parallel platforms. The logical organization refers to a programmer's view of the platform while the physical organization refers to the actual hardware organization of the platform. The two critical components of parallel computing from a programmer's perspective are ways of expressing parallel tasks and mechanisms for specifying interaction between these tasks. The former is sometimes also referred to as the control structure and the latter as the communication model.

2.3.1 Control Structure of Parallel Platforms

Parallel tasks can be specified at various levels of granularity. At one extreme, each program in a set of programs can be viewed as one parallel task. At the other extreme, individual instructions within a program can be viewed as parallel tasks. Between these extremes lie a range of models for specifying the control structure of programs and the corresponding architectural support for them.

Example 2.10 Parallelism from single instruction on multiple processors

Consider the following code segment that adds two vectors:

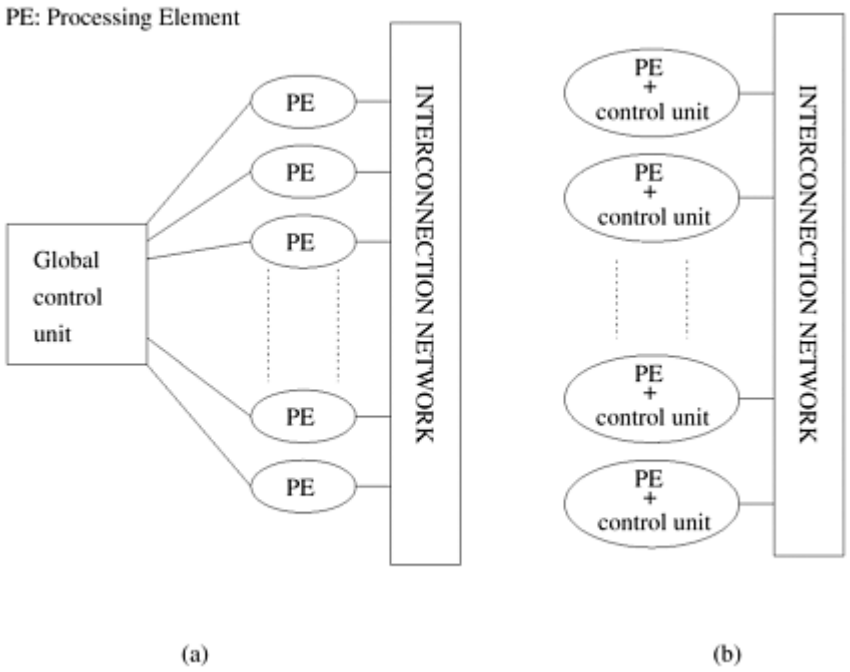
```
1   for (i = 0; i < 1000; i++)
2           c[i] = a[i] + b[i];
```

In this example, various iterations of the loop are independent of each other; i.e., `c[0] = a[0] + b[0]; c[1] = a[1] + b[1];`, etc., can all be executed independently of each other. Consequently, if there is a mechanism for executing the same instruction, in this case `add` on all the processors with appropriate data, we could execute this loop much faster. ■

Processing units in parallel computers either operate under the centralized control of a single control unit or work independently. In architectures referred to as *single instruction stream, multiple data stream* (SIMD), a single control unit dispatches instructions to each processing unit. Figure 2.3(a) illustrates a typical SIMD architecture. In an SIMD parallel computer, the same instruction is executed synchronously by all processing units. In Example 2.10, the `add` instruction is dispatched to all processors and executed concurrently by them. Some of the earliest parallel computers such as the Illiac IV, MPP, DAP, CM-2, and MasPar MP-1 belonged to

this class of machines. More recently, variants of this concept have found use in co-processing units such as the MMX units in Intel processors and DSP chips such as the Sharc. The Intel Pentium processor with its SSE (Streaming SIMD Extensions) provides a number of instructions that execute the same instruction on multiple data items. These architectural enhancements rely on the highly structured (regular) nature of the underlying computations, for example in image processing and graphics, to deliver improved performance.

Figure 2.3. A typical SIMD architecture (a) and a typical MIMD architecture (b).



While the SIMD concept works well for structured computations on parallel data structures such as arrays, often it is necessary to selectively turn off operations on certain data items. For this reason, most SIMD programming paradigms allow for an "activity mask". This is a binary mask associated with each data item and operation that specifies whether it should participate in the operation or not. Primitives such as `where (condition) then <stmnt> <elsewhere stmnt>` are used to support selective execution. Conditional execution can be detrimental to the performance of SIMD processors and therefore must be used with care.

In contrast to SIMD architectures, computers in which each processing element is capable of executing a different program independent of the other processing elements are called *multiple instruction stream, multiple data stream* (MIMD) computers. Figure 2.3(b) depicts a typical MIMD computer. A simple variant of this model, called the *single program multiple data* (SPMD) model, relies on multiple instances of the same program executing on different data. It is easy to see that the SPMD model has the same expressiveness as the MIMD model since each of the multiple programs can be inserted into one large `if-else` block with conditions specified by the task identifiers. The SPMD model is widely used by many parallel platforms and requires minimal architectural support. Examples of such platforms include the Sun Ultra Servers, multiprocessor PCs, workstation clusters, and the IBM SP.

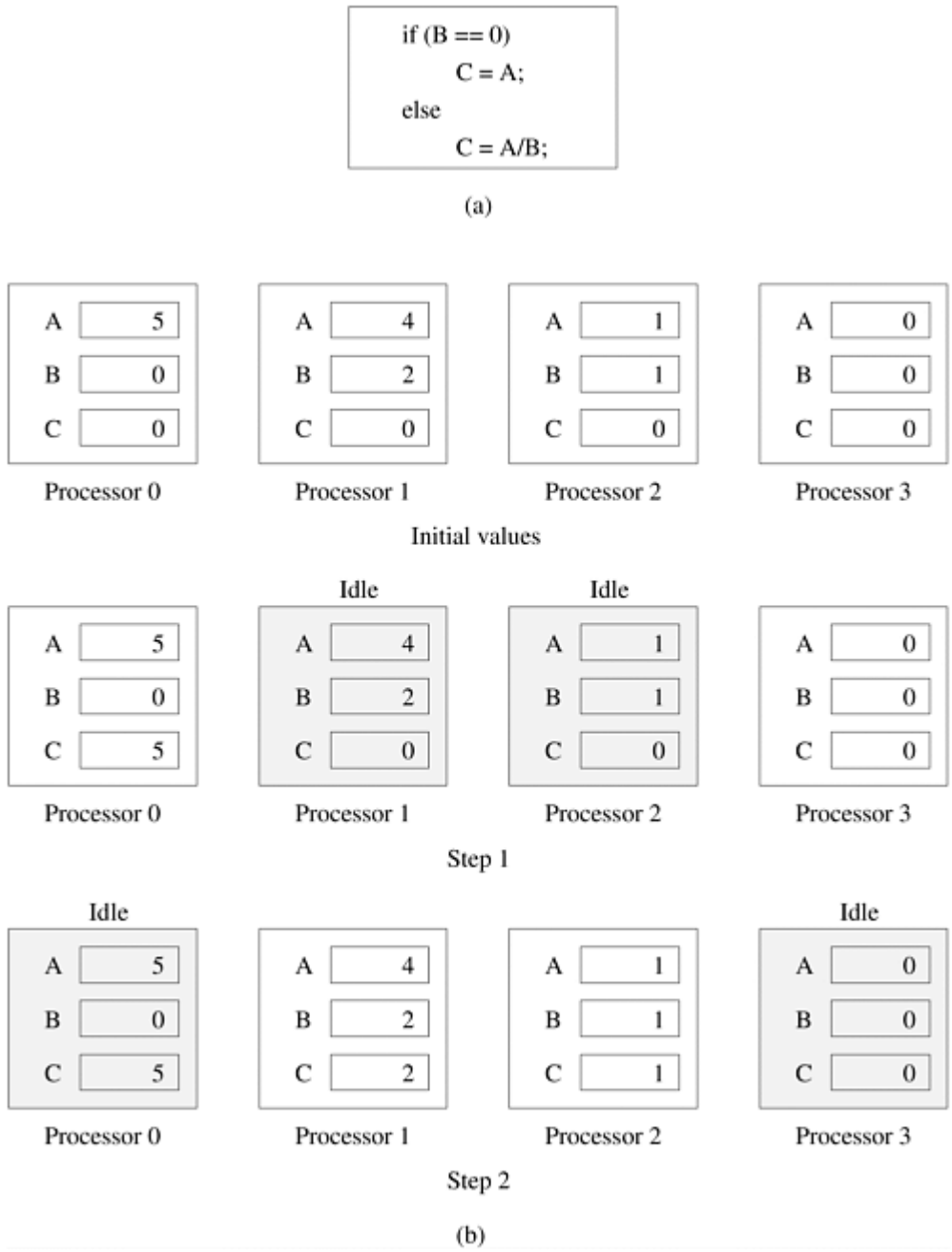
SIMD computers require less hardware than MIMD computers because they have only one global control unit. Furthermore, SIMD computers require less memory because only one copy of the program needs to be stored. In contrast, MIMD computers store the program and operating system at each processor. However, the relative unpopularity of SIMD processors as general purpose compute engines can be attributed to their specialized hardware architectures,

economic factors, design constraints, product life-cycle, and application characteristics. In contrast, platforms supporting the SPMD paradigm can be built from inexpensive off-the-shelf components with relatively little effort in a short amount of time. SIMD computers require extensive design effort resulting in longer product development times. Since the underlying serial processors change so rapidly, SIMD computers suffer from fast obsolescence. The irregular nature of many applications also makes SIMD architectures less suitable. Example 2.11 illustrates a case in which SIMD architectures yield poor resource utilization in the case of conditional execution.

Example 2.11 Execution of conditional statements on a SIMD architecture

Consider the execution of a conditional statement illustrated in Figure 2.4. The conditional statement in Figure 2.4(a) is executed in two steps. In the first step, all processors that have B equal to zero execute the instruction $C = A$. All other processors are idle. In the second step, the 'else' part of the instruction ($C = A/B$) is executed. The processors that were active in the first step now become idle. This illustrates one of the drawbacks of SIMD architectures. ■

Figure 2.4. Executing a conditional statement on an SIMD computer with four processors: (a) the conditional statement; (b) the execution of the statement in two steps.



2.3.2 Communication Model of Parallel Platforms

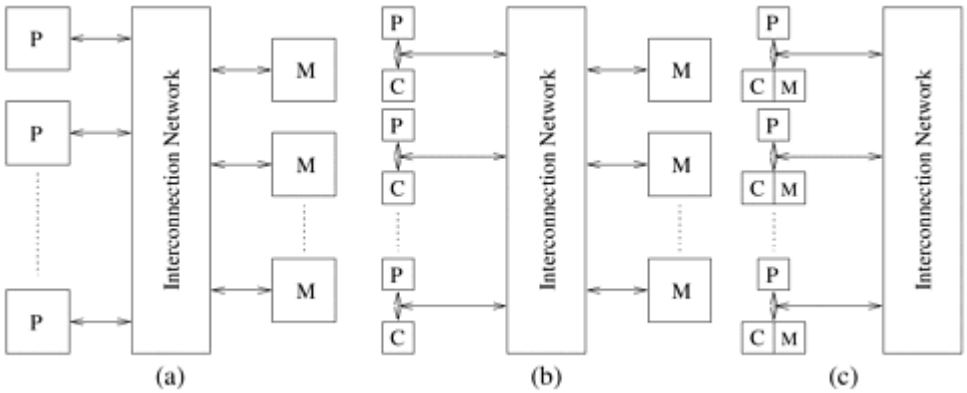
There are two primary forms of data exchange between parallel tasks – accessing a shared data space and exchanging messages.

Shared-Address-Space Platforms

The "shared-address-space" view of a parallel platform supports a common data space that is accessible to all processors. Processors interact by modifying data objects stored in this shared-address-space. Shared-address-space platforms supporting SPMD programming are also

referred to as *multiprocessors*. Memory in shared-address-space platforms can be local (exclusive to a processor) or global (common to all processors). If the time taken by a processor to access any memory word in the system (global or local) is identical, the platform is classified as a *uniform memory access* (UMA) multicomputer. On the other hand, if the time taken to access certain memory words is longer than others, the platform is called a *non-uniform memory access* (NUMA) multicomputer. Figures 2.5(a) and (b) illustrate UMA platforms, whereas Figure 2.5(c) illustrates a NUMA platform. An interesting case is illustrated in Figure 2.5(b). Here, it is faster to access a memory word in cache than a location in memory. However, we still classify this as a UMA architecture. The reason for this is that all current microprocessors have cache hierarchies. Consequently, even a uniprocessor would not be termed UMA if cache access times are considered. For this reason, we define NUMA and UMA architectures only in terms of memory access times and not cache access times. Machines such as the SGI Origin 2000 and Sun Ultra HPC servers belong to the class of NUMA multiprocessors. The distinction between UMA and NUMA platforms is important. If accessing local memory is cheaper than accessing global memory, algorithms must build locality and structure data and computation accordingly.

Figure 2.5. Typical shared-address-space architectures: (a) Uniform-memory-access shared-address-space computer; (b) Uniform-memory-access shared-address-space computer with caches and memories; (c) Non-uniform-memory-access shared-address-space computer with local memory only.



The presence of a global memory space makes programming such platforms much easier. All read-only interactions are invisible to the programmer, as they are coded no differently than in a serial program. This greatly eases the burden of writing parallel programs. Read/write interactions are, however, harder to program than the read-only interactions, as these operations require mutual exclusion for concurrent accesses. Shared-address-space programming paradigms such as threads (POSIX, NT) and directives (OpenMP) therefore support synchronization using `locks` and related mechanisms.

The presence of caches on processors also raises the issue of multiple copies of a single memory word being manipulated by two or more processors at the same time. Supporting a shared-address-space in this context involves two major tasks: providing an address translation mechanism that locates a memory word in the system, and ensuring that concurrent operations on multiple copies of the same memory word have well-defined semantics. The latter is also referred to as the *cache coherence* mechanism. This mechanism and its implementation are discussed in greater detail in Section 2.4.6. Supporting cache coherence requires considerable hardware support. Consequently, some shared-address-space machines only support an address translation mechanism and leave the task of ensuring coherence to the programmer. The native programming model for such platforms consists of primitives such as `get` and `put`. These primitives allow a processor to get (and put) variables stored at a remote processor.

However, if one of the copies of this variable is changed, the other copies are not automatically updated or invalidated.

It is important to note the difference between two commonly used and often misunderstood terms – shared-address-space and shared-memory computers. The term shared-memory computer is historically used for architectures in which the memory is physically shared among various processors, i.e., each processor has equal access to any memory segment. This is identical to the UMA model we just discussed. This is in contrast to a distributed-memory computer, in which different segments of the memory are physically associated with different processing elements. The dichotomy of shared- versus distributed-memory computers pertains to the physical organization of the machine and is discussed in greater detail in Section 2.4. Either of these physical models, shared or distributed memory, can present the logical view of a disjoint or shared-address-space platform. A distributed-memory shared-address-space computer is identical to a NUMA machine.

Message-Passing Platforms

The logical machine view of a message-passing platform consists of *p* processing nodes, each with its own exclusive address space. Each of these processing nodes can either be single processors or a shared-address-space multiprocessor – a trend that is fast gaining momentum in modern message-passing parallel computers. Instances of such a view come naturally from clustered workstations and non-shared-address-space multicomputers. On such platforms, interactions between processes running on different nodes must be accomplished using messages, hence the name *message passing*. This exchange of messages is used to transfer data, work, and to synchronize actions among the processes. In its most general form, message-passing paradigms support execution of a different program on each of the *p* nodes.

Since interactions are accomplished by sending and receiving messages, the basic operations in this programming paradigm are `send` and `receive` (the corresponding calls may differ across APIs but the semantics are largely identical). In addition, since the send and receive operations must specify target addresses, there must be a mechanism to assign a unique identification or ID to each of the multiple processes executing a parallel program. This ID is typically made available to the program using a function such as `whoami`, which returns to a calling process its ID. There is one other function that is typically needed to complete the basic set of message-passing operations – `numprocs`, which specifies the number of processes participating in the ensemble. With these four basic operations, it is possible to write any message-passing program. Different message-passing APIs, such as the Message Passing Interface (MPI) and Parallel Virtual Machine (PVM), support these basic operations and a variety of higher level functionality under different function names. Examples of parallel platforms that support the message-passing paradigm include the IBM SP, SGI Origin 2000, and workstation clusters.

It is easy to emulate a message-passing architecture containing *p* nodes on a shared-address-space computer with an identical number of nodes. Assuming uniprocessor nodes, this can be done by partitioning the shared-address-space into *p* disjoint parts and assigning one such partition exclusively to each processor. A processor can then "send" or "receive" messages by writing to or reading from another processor's partition while using appropriate synchronization primitives to inform its communication partner when it has finished reading or writing the data. However, emulating a shared-address-space architecture on a message-passing computer is costly, since accessing another node's memory requires sending and receiving messages.