Parallel Processing Systems

Computing System Laboratory

January 13, 2012

Contents

1 Parallel Processing Platforms **3**
- 1.1 PRAM: The ideal parallel platform . . . 3
- 1.2 Flynn's Taxonomy . . . 4
- 1.3 Shared Memory Platforms . . . 5
- 1.4 Distributed Memory Platforms . . . 6

2 Analytical Modelling **9**
- 2.1 Performance evaluation metrics . . . 9
- 2.2 Amdahl's law . . . 10
- 2.3 Scalability . . . 10
- 2.4 Performance modelling . . . 11
  - 2.4.1 Modelling computation time . . . 11
  - 2.4.2 Modelling communication time . . . 12
  - 2.4.3 Modelling idle time . . . 13

3 Parallel Programming: Design **15**
- 3.1 Computation and data partitioning . . . 15
- 3.2 Task interaction . . . 18
- 3.3 Mapping tasks to processes . . . 20
- 3.4 Communication and synchronization . . . 22

4 Parallel Programming: Implementation **25**
- 4.1 Parallel programming models . . . 25
  - 4.1.1 Shared address space . . . 25
  - 4.1.2 Message passing . . . 26
- 4.2 Parallel programming constructs . . . 27
  - 4.2.1 SPMD . . . 27
  - 4.2.2 Fork / Join . . . 28
  - 4.2.3 Task graphs . . . 29
  - 4.2.4 Parallel *for* . . . 29
- 4.3 Languages, libraries and tools . . . 31
  - 4.3.1 POSIX Threads (Pthreads) . . . 31
  - 4.3.2 OpenMP . . . 31
  - 4.3.3 Cilk . . . 32
  - 4.3.4 Threading Building Blocks . . . 32
  - 4.3.5 Java threads . . . 32
  - 4.3.6 Message passing interface (MPI) . . . 32
  - 4.3.7 PGAS languages . . . 33

5 Programming for Shared Memory **35**
- 5.1 Hardware concepts . . . 35
- 5.2 Data sharing . . . 37
- 5.3 Synchronization . . . 38
- 5.4 Memory bandwidth saturation . . . 39

6 Programming for Distributed Memory **41**
6.1 Hardware concepts 41
6.2 Data distribution 41
6.3 Communication 42
6.4 Resource sharing 43

Chapter 1

Parallel Processing Platforms

1.1 PRAM: The ideal parallel platform

An important step in designing programs for serial computation is algorithmic asymptotic analysis. This gives us a solid view of the algorithmic behaviour at large input and forms a good basis for comparison of various algorithms. The goal of asymptotic analysis is to categorize algorithms in large complexity classes (using the "*Big O*" notation) without focusing on "constants" that differentiate execution behaviour to a smaller extent. To perform algorithmic analysis for serial computing, one needs a *model of computation* which can be defined in terms of an abstract computer, e.g., the *Random Access Machine (RAM)* or the *Turing machine*, and/or by postulating that certain operations are executed in unit time. To perform "Big O" analysis for serial algorithms, the idealized RAM or Turing machines and any state-of-the art serial CPU following the general *von Neumann* architectural model can be considered equivalent (recall: we do not care about constants).

Theoretical computational models and algorithmic asymptotic analysis for real-life serial systems has been a nice success story for the coupling of theory and practice. Reasonably, one would decide to extend this strategy to parallel computation as well, i.e., define an idealized parallel platform, perform algorithmic analysis for this platform and use this analysis for algorithmic design and evaluation on real hardware. A natural extension of the serial model of computation RAM is called *Parallel Random Access Machine (PRAM)* and consists of p processors and a global memory of unbounded size that is uniformly accessible to all processors. All processors access the same address space. In PRAM synchronization and communication come at zero cost while the machine provides any problem-dependent number of processors.

Since in PRAM all processors have a common view of global memory, common access needs to be further defined. Thus, four PRAM models can be considered:

1. Exclusive Read Exclusive Write (*EREW-PRAM*): Every memory cell can be read or written to by only one processor at a time. This is the most weak PRAM model enforcing parallel algorithms to be designed with processors operating on disjoint data sets at each time step. Any concurrent access is illegal (e.g. the program will terminate or have undefined behaviour).

2. Concurrent Read Exclusive Write (*CREW-PRAM*): Multiple processors can read a memory cell but only one can write at a time. Thus, algorithms can be designed with read-only

data sets accessed by all processors and disjoint write data sets per processor at each time step.

3. Exclusive Read Concurrent Write (*ERCW-PRAM*): Multiple processors can write a memory cell but only one can read at a time. Clearly, this model is of no theoretical or practical use and thus is never considered.

4. Concurrent Read Concurrent Write (*CRCW-PRAM*): Multiple processors can read and write a memory cell. While concurrent reads (as in the CREW case) can be dealt by serialization without altering the semantics of the algorithm, concurrent writes need further elaboration. Again we distinguish four cases:

   - *Common*: All processors write the same value, otherwise the operation is illegal.
   - *Arbitrary*: Only one arbitrary attempt is successful, others retire.
   - *Priority*: Processor with highest rank performs a successful write.
   - *Reduction*: A collective operation (e.g. *sum*, *and*, *max*) is applied to all data before written to the memory cell.

Unlike RAM, PRAM makes several simplifying assumptions that deflect it from real platforms. Apart from the severe assumption of the unlimited number of processors, PRAM implies a uniform and switched access to shared memory. Thus for m memory cells and p processors (recall: even if not unlimited, p should be quite large) memory-processor connectivity would require mp switches. For a reasonable memory size, such a switching network would be extremely expensive to realize in practice. Although shared-memory systems have many common features with PRAM, such hardware and cost considerations limit the number of processors that can be included in these systems, making them of limited use for complexity analysis. Thus, PRAM has mostly theoretical value and finds little space in algorithmic design for parallel computing platforms.

1.2 Flynn's Taxonomy

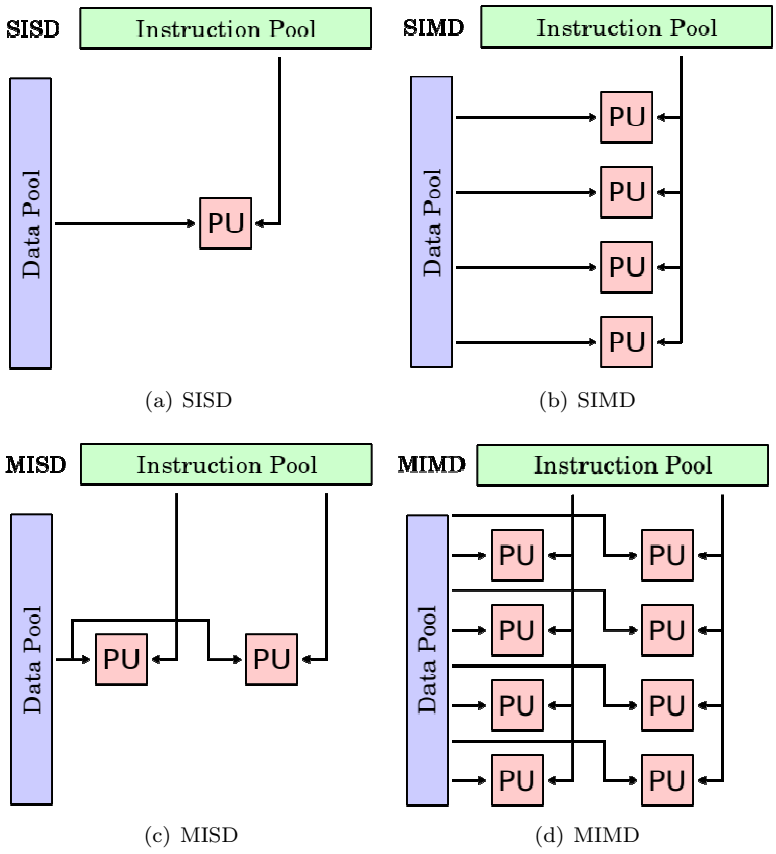
Flynn's taxonomy is a classification of computer architectures, proposed by Michael J. Flynn in 1966. The four classifications defined by Flynn are based upon the number of concurrent instruction (or control) and data streams available in the architecture (see Figure 1.1):

- Single Instruction, Single Data stream (SISD) A sequential computer which exploits no parallelism in either the instruction or data streams. Single control unit (CU) fetches single Instruction Stream (IS) from memory. The CU then generates appropriate control signals to direct single processing element (PE) to operate on single Data Stream (DS) i.e. one operation at a time. Examples of SISD architecture are the traditional uniprocessor machines like a PC (currently manufactured PCs have multiple processors) or old mainframes.

- Single Instruction, Multiple Data streams (SIMD). A computer which exploits multiple data streams against a single instruction stream to perform operations which may be naturally parallelized. For example, an array processor or GPU.

- Multiple Instruction, Single Data stream (MISD). Multiple instructions operate on a single data stream. Uncommon architecture which is generally used for fault tolerance. Heterogeneous systems operate on the same data stream and must agree on the result. Examples include the Space Shuttle flight control computer.

- Multiple Instruction, Multiple Data streams (MIMD). Multiple autonomous processors simultaneously executing different instructions on different data. Distributed systems are generally recognized to be MIMD architectures; either exploiting a single shared memory space or a distributed memory space.



(a) SISD (b) SIMD

(c) MISD (d) MIMD

Figure 1.1: Flynn's taxonomy

1.3 Shared Memory Platforms

Since theoretical computational models have not been very useful in designing parallel algorithms for real systems, programmers and algorithm designers need to resort to abstractions of actual parallel processing platforms. This has the advantage of leading to realistic analyses, but the disadvantage of being dependent on current hardware. There is no guarantee that algorithms developed for current systems will be efficient in future ones. Figure 1.2 demonstrates the typical organization of a uniprocessor system. One CPU with a cache memory (denoted $\$$) is connected to main memory (M) with a memory interconnect and to the peripherals with an I/O interconnect. We can consider parallel processing platforms as an extension of this organization.

Depending on which interconnect we choose to attach multiple systems on, two major families of parallel platforms are derived: shared memory systems and distributed memory systems.

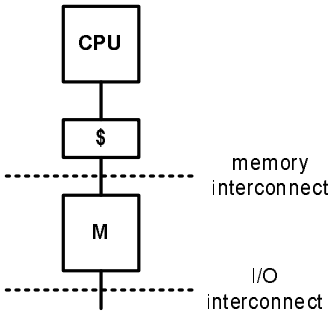


Figure 1.2: Typical organization of a uniprocessor system.

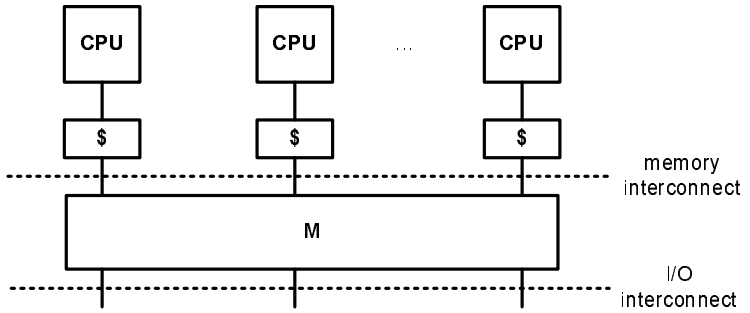
Shared memory systems are built by attaching multiple CPUs on the memory interconnect. If all memory chunks are equally distant from all CPUs, the system is called *Uniform Memory Access (UMA)* (Figure 1.3(a)). If some memory banks are closer to some processors the system is called *Non-Uniform Memory Access (NUMA)* (Figure 1.3(b)). Shared memory systems in general provide the desired characteristic of shared address space between programs that execute on multiple CPUs. Data structures can be shared among CPUs that are able to communicate with standard load/store operations to main memory. Due to this common view of memory, shared memory systems are traditionally considered as parallel platforms that make parallel programming easier. This is to some extent correct, since parallel programming for shared memory systems can be supported by moderate modifications to serial programming, due to the efficient access of shared data. On the other hand, to deal with race conditions and data dependencies on shared data requires special care and is extremely error-prone and counter-productive.

The most common memory interconnect technology is the bus. Buses have the desirable property that the cost of the network scales linearly as the number of processors p. This cost is typically associated with the bus interfaces. Furthermore, the distance between any two nodes in the network is constant. Buses are also ideal for broadcasting information and greatly facilitate the implementation of snooping protocols for cache coherence. However, the bandwidth of a bus is bounded, thus there is a physical limit to the number of nodes that can be attached in a bus without greatly sacrificing performance. Typical bus-based systems are limited to dozens of nodes.

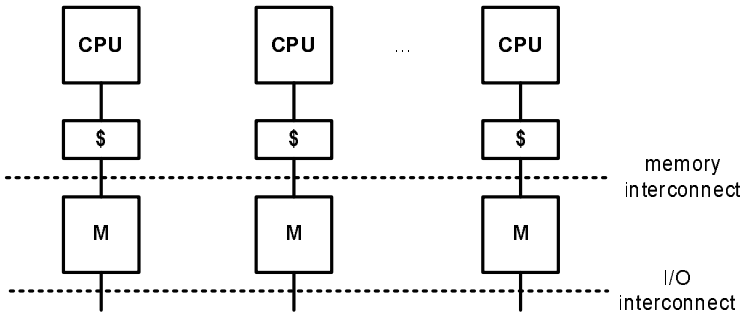
An alternative to the bus for shared-memory systems is a crossbar network connecting p processors with b memory banks. This family of interconnects employs a grid of switches that enable concurrent communication between the nodes attached to the grid. The total number of switching nodes required to implement such a network is pb. As the number of processing nodes rises the switching complexity is difficult to realize at high data rates. Crossbar networks are not scalable in terms of cost.

1.4 Distributed Memory Platforms

Enterprise parallel applications have enormous needs for processing power and request thousands, even millions of processing cores. Clearly, this demand cannot be met by shared-memory



(a) UMA



(b) NUMA

Figure 1.3: Shared-memory systems

systems. To create systems that can scale up to thousands of cores processing nodes are connected on the I/O bus as shown in Figure 1.4. In this paradigm, systems may be built upon commodity interconnection technology like Gbit Ethernet or more advanced low, latency and high-bandwidth networks like Myrinet or Infiniband. Large supercomputers are also built around custom interconnects designed and implemented to cover the specific needs of these systems.

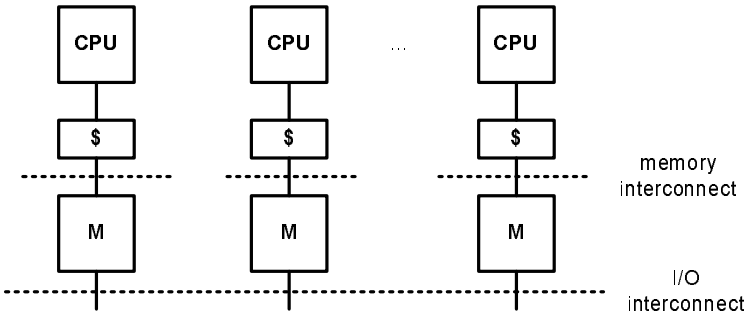


Figure 1.4: Distributed-memory system

In distributed memory platforms there is no view of global memory and thus processors need to communicate explicitly through the communication network. This can be realized by programming models that adopt a message-passing approach exposed to the programmer. Processes have a view of local memory and exchange of information is made with explicit calls to communication libraries, typically occurring both at the sender and receiver sides. Although a difficult, effort-consuming and error-prone approach, message-passing is the dominant parallel programming paradigm for distributed memory systems nowadays. Alternatively, a software

stack called Distributed Shared Memory (DSM) provides a shared address space implemented on the distributed physical memory of the platform, but such approach suffers from severe performance overheads. Recently, a novel programming model followed by Partitioned Global Address Space (PGAS) languages assumes a global memory address space that is logically partitioned and a portion of it is local to each processor. The PGAS model is the basis of Unified Parallel C (UPC), Co-array Fortran, Titanium, Fortress, Chapel and X10. The PGAS model envisions to achieve the programmability of the shared address space model together with the scalability of the message-passing model. PGAS languages need strong hardware support, especially at the communication network that is expected to support very fast one-sided communication.

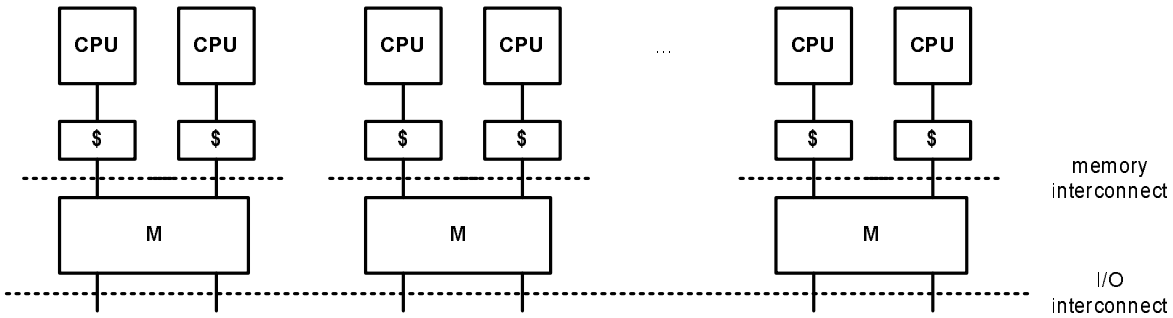


Figure 1.5: Hybrid system

The widespread of the multicore technology together with the good scalability properties of distributed memory systems has normally led to the adoption of a hybrid, two-level parallel architecture where shared memory systems are interconnected to form a large-scale parallel system as depicted in Figure 1.5. In this case, either pure message-passing or a hybrid programming model can be applied.

Chapter 2

Analytical Modelling

As mentioned in the previous section, sequential algorithms are evaluated in terms of execution time expressed as a fraction of the size of its input. The execution time of a parallel algorithm depends not only on the input size but also on the number of the processing elements used. To make things even more complicated, the parallel execution time depends also on the architecture and hardware characteristics of the parallel platform, e.g. memory organization (shared vs. distributed), memory hierarchy (size and organization of cache memories), interconnection network, hardware support for synchronization etc. This means that if we parallelize an initial sequential algorithm that executes in time T_s across p processors, we cannot expect a parallel execution time that equals $\frac{T_s}{p}$. This is due to three reasons:

1. Processor interaction: In the typical case, processors need to interact in order to synchronize or communicate. Processor interaction is a major source of performance overhead in parallel programs.

2. Processor idle times: Equal distribution of computation across p processors is not always feasible. In several cases load imbalance may occur between processors leading to processor idle times and subsequent performance degradation.

3. Algorithmic issues: A large number of sequential algorithms are inherently serial. This means that in several cases in order to parallelize a serial algorithm with execution time T we need to resort to an alternative algorithm with execution time T' and $T' > T$. This of course needs to be considered when evaluating the performance gains of parallelization.

2.1 Performance evaluation metrics

The following evaluation metrics are important to assess the effectiveness of a parallelization process:

- The *parallel execution time* denoted T_p is the overall execution time that elapses from the moment a parallel computation starts to the moment the last processing elements completes its work.

- The *total parallel overhead* expresses the extra work carried out by the parallel execution. Since pT_p is the total total work carried out by the parallel execution (p workers working for T_p time) and T_s is the useful work, then the total parallel overhead is $T_o = pT_p - T_s$.

- The *Speedup* is the most descriptive of the performance metrics used to assess the effectiveness of parallelization. Speedupe S is defined as:

$$S = \frac{T_s}{T_p}$$

where T_s is the serial time of the *best performing serial algorithm* and T_p is the parallel execution time with p processors. Thus, speedup gives a good view of "how many times faster is the parallel program compared to the serial one". Ideally, programs would have *linear speedup*, i.e. $S = p$, but in typical cases $S < p$, while in special cases programs may achieve *super-linear speedup* ($S > p$).

- The *Efficiency* metric is a measure of the fraction of time devoted by each processing element to the execution of the parallel algorithm. Efficiency E is defined as:

$$E = \frac{S}{p}$$

Similar to the properties of speedup, perfect efficiency is 1 while in typical cases it holds $E < 1$.

2.2 Amdahl's law

Amdahl's law is a fundamental law in parallel computing. It is used to find the maximum expected improvement to an overall system when only part of the system is improved. It is often used in parallel computing to predict the theoretical maximum speedup using multiple processors. Amdahl's law states that if f is the proportion of a program that can be made parallel (i.e. benefit from parallelization), and $1 - f$ is the proportion that cannot be parallelized (remains serial), then the maximum speedup that can be achieved by using p processors is:

$$S_{max} = \frac{1}{(1-f) + \frac{f}{p}}$$

For example, if 50% of an algorithm can be parallelized, then the maximum speedup that we could expect for 4 processors is $S_{max} = \frac{1}{\frac{1}{2}+\frac{1}{8}} = 1.6$. Although simple in its essence, Amdahl's law provides the initial and most significant guide to parallelization and in general to code optimization: any optimization approach should focus on the part of the algorithm that dominates the execution time. This is in line with the famous quote by D. Knuth: "*We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil*".

2.3 Scalability

Scalability is the property of parallel programs to increase their performance as the number of processing nodes increases. There is no strict definition of scalability, but looking at the two extremes, a program with linear speedup "scales well", while on the other hand when a program fails to increase performance or even degrades performance for $p > p_0$ we say that "scalability breaks at p_0". In the context of high performance computing there are two common notions of scalability. The first is *strong scaling*, which is defined as how the solution time varies with

the number of processors for a fixed total problem size. The second is *weak scaling*, which is defined as how the solution time varies with the number of processors for a fixed problem size per processor.

2.4 Performance modelling

A good performance model, like a good scientific theory, is able to explain available observations and predict future circumstances, while abstracting unimportant details. Amdahl's law, empirical observations, and asymptotic analysis do not satisfy the first of these requirements. On the other hand, conventional computer system modelling techniques, which typically involve detailed simulations of individual hardware components, introduce too many details to be of practical use to parallel programmers. We discuss performance modelling techniques that provide an intermediate level of detail. These techniques are certainly not appropriate for all purposes: they consider simplified parallel architectures and do not take into account, for example, cache behaviour. However, they have been proven useful in a wide range of parallel algorithm design problems.

The performance models considered here specify a metric such as execution time T as a function of problem size N, number of processors p and other algorithm and hardware characteristics:

$$T = f(N, p, \dots)$$

As mentioned above, the execution time of a processor can be decomposed into computation time (T_{comp}), communication time (T_{comm}) and idle time (T_{idle}). Thus for processor j it holds:

$$T^j = T^j_{comp} + T^j_{comm} + T^j_{idle}$$

We defined the execution time of a parallel program as the time that elapses from when the first processor starts executing on the problem to when the last processor completes execution. Thus, the parallel execution time can be modelled as:

$$T = max\big(T^j_{comp} + T^j_{comm} + T^j_{idle}\big),\ j = 1 \dots p$$

A good metric is also the average of the execution times of all processors:

$$T = \frac{1}{p} \sum_{j=1}^{p} \big(T^j_{comp} + T^j_{comm} + T^j_{idle}\big),\ j = 1 \dots p$$

2.4.1 Modelling computation time

The computation time of an algorithm is the time spent performing computation rather than communicating or idling. If we have a sequential program that performs the same computation as the parallel algorithm, we can determine by timing that program. Otherwise, we may have to implement key kernels. Computation time will normally depend on some measure of problem size, whether that size is represented by a single parameter N or by a set of parameters $N_1, N_2 \dots$. If the parallel algorithm replicates computation, then computation time will also depend on the number of tasks or processors. In a heterogeneous parallel computer (such as a workstation network), computation time can vary according to the processor on which computation is performed.

Computation time will also depend on characteristics of processors and their memory systems. For example, scaling problem size or number of processors can change cache performance or the effectiveness of processor pipelining. As a consequence, one cannot automatically assume that total computation time will stay constant as the number of processors changes.

A simple and straightforward way to model computation time (or at least an upper bound of it) is to extract the number of operations (ops) required by an algorithm and multiply it by the CPU speed provided by the vendor (in sec/op). Thus, one estimate of T_{comp} is:

$$T_{comp} = \text{ops of Algorithm} \times \text{CPU speed}$$

The above model assumes that the CPU can be fed with data by the memory subsystem at a rate that can always cover the CPU's needs. However, this is not the case in modern systems, especially in those containing multiple cores. Hence, we want a model that relates processor performance to off-chip memory traffic. Towards this goal, we use the term "operational intensity" (OI) (in operations/byte) to mean operations per byte of DRAM traffic, defining total bytes accessed as those bytes that go to the main memory after they have been filtered by the cache hierarchy. That is, we measure traffic between the caches and memory rather than between the processor and the caches. Thus, operational intensity predicts the DRAM bandwidth needed by a kernel on a particular computer. In this case, an upper bound of the computational time is provided by the *Roofline model* as:

$$T_{comp} = \text{ops of Algorithm} \times max\big(\text{CPU speed}, \frac{1}{\text{Memory bandwidth} \times \text{OI}}\big)$$

2.4.2 Modelling communication time

The communication time between two nodes in a distributed-memory platform can be decomposed into the sum of the time to prepare a message for transmission and the time taken by the message to traverse the network to its destination. The principal parameters that determine the communication latency are as follows:

1. *Startup time (t_s)*: The startup time is the time required to handle a message at the sending and receiving nodes. This includes the time to copy the data from user space to the communication engine, prepare the message (header, trailer, error correction, etc.), execute the routing algorithm and establish the interface between the sender and receiver nodes. This delay is incurred once per message.

2. Per-hop time (t_h): After a message leaves a node, it takes a finite amount of time to reach the next node in its path within the communication network. The time taken by the header of a message to travel between two directly-connected nodes in the network is called the per-hop time. The per-hop time is related to the latency incurred by the interconnect's hardware.

3. Per-word transfer time (t_w): This time is related to the channel bandwidth of the interconnection network (b). Each word takes $t_w = \frac{1}{b}$ to traverse the link.

The communication cost for a message of size m transferred between two nodes that are l hops away is:

$$T_{comm} = t_s + lt_h + mt_w$$

In general, it is very difficult for the programmer to consider the effect of the per-hop time. Many message passing libraries like MPI offer little control to the programmer on the mapping of processes to physical processors. Even if this control was granted to the programmer, it would be quite cumbersome to include such parameters in the design and implementation of the algorithm. On the other hand, several network architectures rely on routing mechanisms that include a constant number of steps (e.g. 2). This means that the effect of the per-hop time can be included in t_s. Finally, in typical cases it holds $t_s \gg t_h$ or $mt_w \gg t_h$ and since l can be relatively small, the effect of the per-hop time can be ignored, leading to this simplified model for the communication cost of a single message:

$$T_{comm} = t_s + mt_w$$

Many communication scenarios in real-life applications involve collective communication including multiple nodes. Broadcast (one-to-all) and reduction (all-to-one) operations are based on point-to-point communication, but are implemented organizing the participating nodes in a tree and utilizing concurrent point-to-point communication between the processes as shown in Figure 2.1 for 8 processes. Thus, in this case the communication cost is modelled as:

$$T_{comm} = (t_s + mt_w) \log p$$

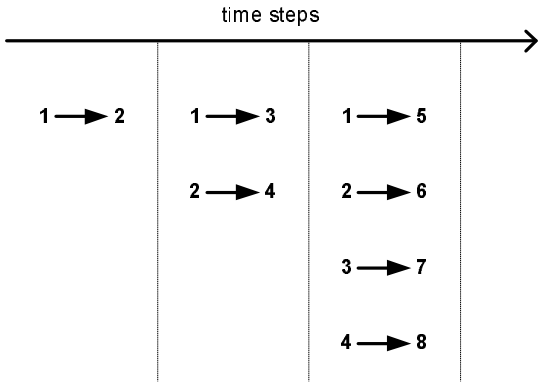


Figure 2.1: Broadcast steps for 8 processes

2.4.3 Modelling idle time

Both computation and communication times are specified explicitly in a parallel algorithm; hence, it is generally straightforward to determine their contribution to execution time. Idle time can be more difficult to determine, however, since it often depends on the order in which operations are performed.

A processor may be idle due to lack of computation or lack of data. In the first case, idle time may be avoided by using load-balancing techniques. In the second case, the processor is idle while the computation and communication required to generate remote data are performed. This idle time can sometimes be avoided by structuring a program so that processors perform other computation or communication while waiting for remote data. This technique is referred to as overlapping computation and communication, since local computation is performed concurrently with remote communication and computation. Such overlapping can be achieved in two ways. A

simple approach is to create multiple tasks on each processor. When one task blocks waiting for remote data, execution may be able to switch to another task for which data are already available. This approach has the advantage of simplicity but is efficient only if the cost of scheduling a new task is less than the idle time cost that is avoided. Alternatively, a single task can be structured so that requests for remote data are interleaved explicitly with other computation.

Chapter 3

Parallel Programming: Design

In this chapter we review some of the basic principles in the design of parallel programs. We need to point out that in parallel programming there is no well established methodology towards designing and implementing "good" parallel code. Driving forces in this process are high-performance evaluated in terms speedup, efficiency and scalability as described in the previous chapter, and code productivity evaluated in terms of reduced implementation cost, maintainability and portability.

We can distinguish four major steps in the design of parallel programs:

1. Computation and data partitioning
2. Design of task interaction
3. Mapping of tasks to processes
4. Orchestration of communication/synchronization

At this point we need to clarify the key terms *task*, *process* and *processor*. We can think of a task as a distinct unit of work, a process as a logical computing agent executing tasks and a processor as the hardware unit that physically executes computations. Thus we may say that tasks are *assigned* or *mapped* to processes and processes are *scheduled* on processors.

3.1 Computation and data partitioning

The first step in the parallelization design is to detect tasks from the serial algorithm that "seem" to be able to validly execute concurrently. The actual verification of parallelism will be carried out in the next step. Additionally, one needs to identify the data that need to be allocated in each task. There are two main approaches to carry out partitioning: one *task centric* and one *data centric*.

In the task centric approach partitioning starts from the definition of tasks with data partitioning following. This is the most general and flexible approach, essentially capable of handling all practical cases. As an example consider the computational kernel of matrix-vector multiplication shown in Algorithm 1 and Figure 3.1. These are some of the task centric approaches one could follow to define tasks:

- We consider as task the inner product of input vector x with matrix row $A[i][*]$ to calculate output vector element $y[i]$.
- We consider as task the multiplication of an element of input vector x ($x[j]$) with an element of the matrix ($A[i][j]$). A different task is considered the summation of all products of $x[j]$ with $A[i][j]$ to calculate $y[i]$.
- We consider as task the multiplication of a submatrix of A of size $B \times B$ with the relevant subvector of x. A different task is considered the summation of all previous subproducts to calculate a subvector of y.

Algorithm 1: Matrix-vector multiplication.

Input: A: matrix of size $N_1 \times N_2$
Input: x: vector of size N_2
Output: y: vector of size N_1

```
for i ← 0 to N1 do
    y[i]=0.0;
for i ← 0 to N1 do
    for j ← 0 to N2 do
        y[i]+=A[i][j]*x[j];
```

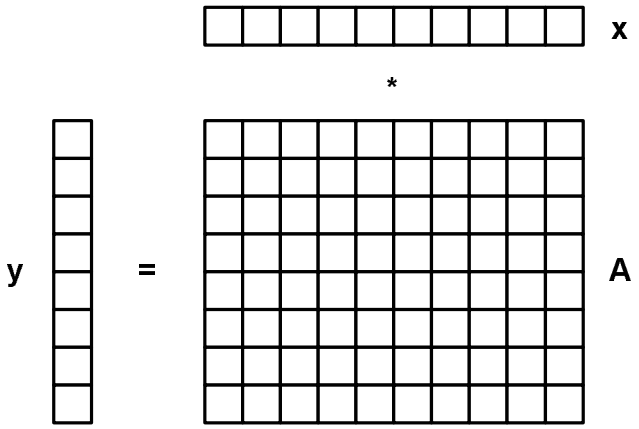


Figure 3.1: Matrix-vector multiplication

On the other hand, in the data centric approach one would start by partitioning the data and then form tasks to work on this data distribution. All tasks in this case are identical and perform operations on different sets of data. On the matrix-vector multiplication example one would choose to partition the matrix by element, by rows, by columns or by blocks and define a task as the computations performed using this chunk of data. Note that the data centric approach is not as general as the task centric approach and is suitable for regular data structures like matrices and vectors and algorithms derived from numerical linear algebra, where the data distribution leads to a straightforward parallel design and implementation. In several cases the data centric approach constructs tasks that are completely independent and can operate with little or no interaction at all. As a general rule of thumb, one could start the design by considering the data centric approach first, and test if this can lead to a fluent parallel design

with load balanced and regularly interacting tasks. If this is not the case, then one can resort to the more general task centric approach.

The sharing attributes of distributed data fall into three major categories: shared, distributed or replicated:

- *Shared* data structures can be supported only by shared address space programming models. It is a convenient way of "partitioning" data (actually no partitioning is required in this case) since no initialization or finalization is needed to distribute and collect data respectively. It is a straightforward approach for read-only data but requires special care when shared data are written by distinct tasks. This is what makes programming for shared address space models look easy, but actually being extremely cumbersome and error prone.

- *Distributed* data structures are used to completely separate working sets between the execution tasks. This is the main approach for message passing programming models. The distribution and collection of data structures at the beginning and at the end of the execution incurs significant overhead to the programmer, but, on the other hand, since tasks operate on disjoint data sets their interaction becomes much more clear. One can state that programming with distributed data structures is more time consuming but less error-prone.

- *Replicated* data structures are used to copy data to the local address spaces of tasks. This is a good practice for small read-only data structures whose distribution would incur extra programming overhead without any significant gain. In several cases, replicated data may be used as a design choice to replicate computation as well. In general, this approach is followed when computation replication is more efficient than computation distribution coupled with the necessary communication.

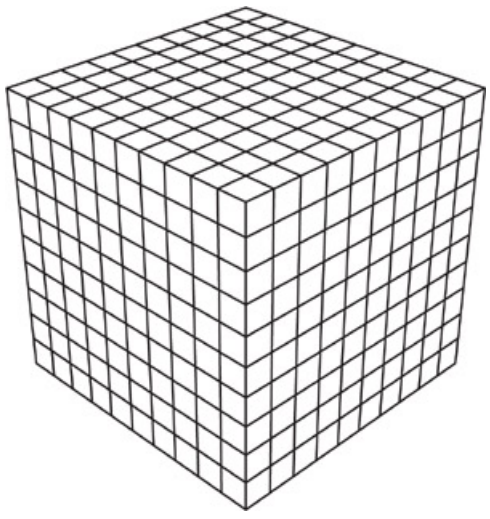
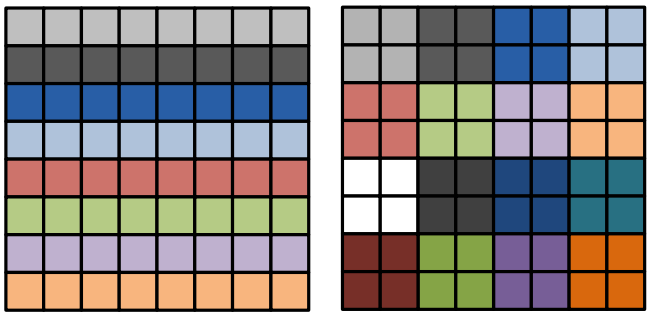


Figure 3.2: 3-dimensional computational grid

Regular data structures like algebraic matrices and N-dimensional computational grids (see Figure 3.2) are typical in applications for parallel computing. Naturally, these data structures are represented as multidimensional arrays in high-level programming languages. To parallelize algorithms involving such data structures (e.g. matrix operations, solutions of Partial Differential Equations) designers typically use the data-centric approach and distribute the data structures

dividing one or more of the array dimensions. Figure 3.3 1 and 2-dimensional distributions for a two dimensional array, also called *row-wise* and *block-wise*. The choice of dimension(s) to be distributed in an N-dimensional array affects several parameters in the execution behavior of an application (memory access, communication, etc) and should be given serious attention.



(a) 1-dimensional (b) 2-dimensional

Figure 3.3: Array distributions

3.2 Task interaction

Apart from some exceptional cases of *ridiculously parallel* applications, tasks need to interact with each other. As in all cases of cooperation, participating entities need to exchange information or synchronize their work. In the case of parallel computing, task interaction needs to take place when tasks share data and in particular, when this sharing involves modification of shared data. We say that a *race condition* occurs when multiple tasks access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place.

Algorithm 2 shows the pseudocode of a kernel counting the prime numbers from 1 to N. Function *isPrime(i)* decides if number i is prime, and is computationally intensive. For the sake of our example, we can consider a naive distribution of this algorithm in two tasks, task A calculating the number of primes from 1 to $N/2$ and task B from $N/2+1$ to N. We can verify that there exists a race condition in the access of the output variable *prime_count*. Recall that the operation *prime_count++* is typically the abbreviation of the following more complex code:

```
tmp = prime_count;
tmp = tmp + 1;
prime_count = tmp;
```

Algorithm 2: Pseudocode for prime number counting.

Input: N: search primes from 1 to N
Output: *prime_count*: number of primes found in $1 \ldots N$

```
for i ← 1 to N do
    if isPrime(i) then
        prime_count++
```

In this code fragment, *prime_count* is shared among the tasks, but *tmp* is a local variable to each task. Now imagine that tasks A and B have successfully found a prime number each, and

proceed concurrently to update the counter *prime_count* which holds the value k. They might simultaneously read k from *prime_count*, set their local *tmp* variables to $k+1$ and both write $k+1$ to *prime_count*. This clearly is not what we expect from the parallel execution of task A and task B. To keep the semantics of the sequential algorithm in this case, we need to orchestrate the concurrent access to the shared variable *prime_count*, e.g. by enforcing *mutual exclusion* to the two tasks, i.e. enabling only one task to enter the *critical section* that modifies the shared variable.

Apart from the general case of race conditions, in several cases tasks need to have an ordered access to shared variables, i.e. there exists a *dependence* between task A and task B. Dependencies between tasks can be represented by *task graphs* which are directed acyclic graphs, with the nodes representing tasks and the edges representing dependencies between tasks (see Figure 3.4). Task graphs can be also enhanced by node labels that describe task effort and edge labels that describe amount of data (e.g. communication) that need to be transferred between tasks.

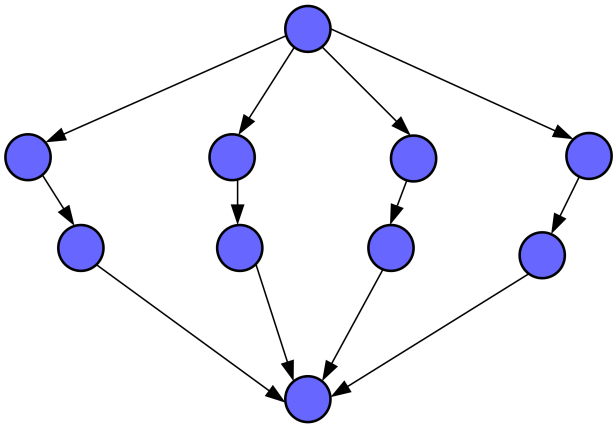


Figure 3.4: A task graph: nodes represent tasks and edges represent dependencies.

Algorithm 3 shows the kernel calculating the inner product between vectors x and y. We can distribute this work by assigning tasks A and B the partial inner product of half vectors (distributing the first half and second half of vectors x and y to tasks A and B respectively) and task C the combination of the result (the summation of the partial dot products). The task graph in this simple example in shown in Figure 3.5 where we can see that there is an ordering between tasks: tasks A and B must execute before task C, since there is a dependence from task A/B to task C.

Algorithm 3: Vector inner product.

Input: x: vector of size N
Input: y: vector of size N
Output: *inner_prod*: scalar holding the inner product

inner_prod = 0 **for** $i \leftarrow 0$ **to** N **do**
  inner_prod+=x[i]*y[i];

Concluding, after task and data partitioning, the design phase involves the detection of race conditions and dependencies between tasks. For some classes of algorithms, especially those that operate on irregular data structures asynchronously, this can be an extremely tedious and error-prone process.

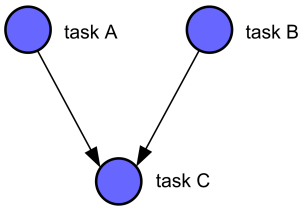
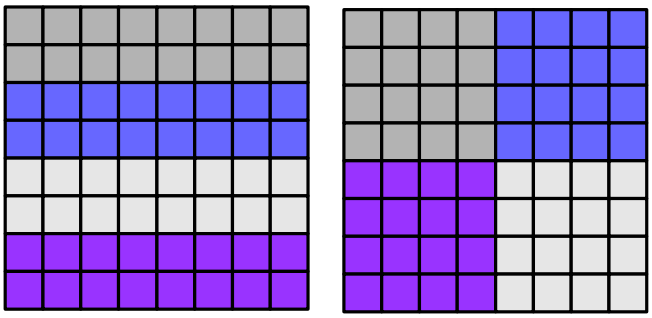


Figure 3.5: A task graph for inner product.

3.3 Mapping tasks to processes

Until now, we have worked with tasks, a computational entity that is independent of the underlying execution platform. At this point we need to consider how tasks will be assigned (mapped) to processes and executed by the actual execution engine. The goal of mapping is primarily to assign a problem-specific set of tasks to a platform-specific number of processes P and minimize overall parallel execution time. Driving forces for an efficient mapping approach are the maximization of parallelism and the minimization of interprocess synchronization/communication. There exist two major mapping strategies depending on the nature of the computation and the interactions between tasks: *static* and *dynamic* mapping.

Static mapping techniques distribute the tasks among processes prior to the execution of the algorithm. Such techniques are used when task generation, size and interaction are known or can be effectively predicted statically. Even when task sizes are known the problem of obtaining an optimal mapping is NP-complete for non-uniform tasks. However, for several practical cases good and easy to implement heuristics exist.

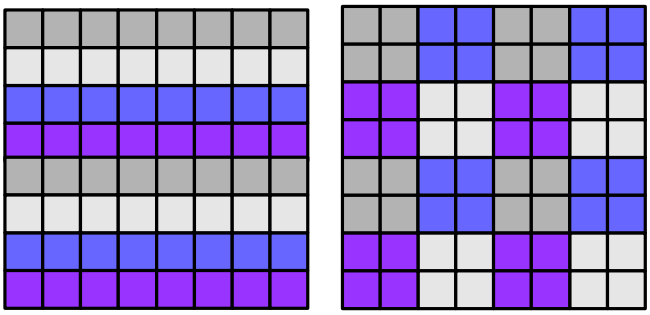


(a) 1-dimensional (b) 2-dimensional

Figure 3.6: Sequential array mappings

Revisiting the algorithms that operate on regular data structures like matrices or grids, the mapping can be performed again in a data-centric approach similar to the one followed to distribute the work into tasks. *Sequential mappings* group contiguous tasks into the same process as shown in Figure 3.6. *Cyclic mappings* assign contiguous tasks into different processes as shown in Figure 3.7. Sequential mappings are straightforward and easier to implement, but fail to achieve load imbalance in algorithms that operate with different density on the various parts of the array. In these cases cyclic mappings are preferable.

Dynamic mapping techniques map tasks to processes during the execution of the algorithm. If tasks are generated dynamically, then they must be mapped dynamically too. If task sizes are unknown then a static mapping can potentially lead to severe load imbalance, dynamic



(a) 1-dimensional (b) 2-dimensional

Figure 3.7: Cyclic array mappings

mappings are usually more effective in this case too. However, dynamic mapping has to pay the cost of online decision taking, interaction between the processes, and is generally more complex to implement, especially when data are distributed between tasks (and processes). Dynamic mapping is also referred to as *dynamic load balancing* (as its primary goal is to distribute load evenly between processes) or *task scheduling*. Dynamic mapping techniques are usually classified as *centralized* or *distributed*.

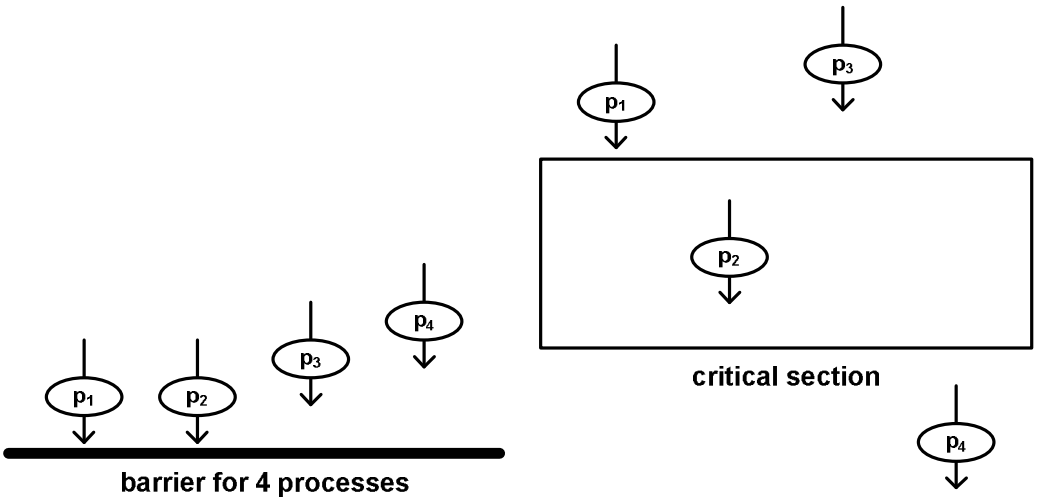
In a centralized mapping scheme the orchestration of the mapping is carried out by a special process that possesses an overall view of the tasks and their execution status. This *master* or *scheduling* process interacts with *slave* or *worker* processes and assigned chunks of work to them. Alternatively, unassigned work can be maintained in a central data structure. Whenever a worker process has no work, it communicates with the scheduling process or accesses the central data structure to obtain further work.

A characteristic and widely applied example of dynamic scheduling is that of the work carried out at the iterations of a *parallel loop*, i.e. a loop whose iterations can be executed in parallel without any kind of interaction. The calculation of prime numbers in the set $1 \dots N$ shown in Algorithm 2 is such an example (for the sake of this example we can ignore the update of the *primer_counter* variable). Prime numbers are not uniformly distributed in the set under consideration, and, furthermore, the calculations to decide whether numbers a and b are can be significantly different. Therefore each iteration of the loop in Algorithm 2 can take different amounts of time. A naive mapping (such as the one we used in Section 3.2) can lead to load-imbalance and severe performance degradation due to processor idle times. A possible solution to this problem would be to maintain a central pool of loop indices and, whenever a process is idle, it can pick an index, delete it from the pool and perform the relevant calculations. This method of scheduling independent iterations of a loop among parallel processes is called *self scheduling*. There exist several interesting variations of this approach that distribute chunks of iterations to processes in order to increase load balance and keep scheduling costs low.

Centralized mapping schemes are easier to implement and work well for a large number of realistic cases. However, they suffer from scalability since there exist single points of congestion, either these being the communication path to the master process, or the shared data structure that keeps information on the work that waits to be scheduled. Moreover, shared data structures are not easy to implement in distributed memory platforms. For these reasons, several classes of applications benefit from distributed scheduling schemes, where each process maintains its own work queue and interacts with other processes in the cases when it becomes idle or overloaded.

3.4 Communication and synchronization

The final step in the design of a parallel program is to insert synchronization and/or communication primitives between processes that will ensure the semantic equivalence of the serial with the parallel program. In the previous step of task mapping several tasks have been allocated to the same process, so their interactions remain within the same execution context. In this case task interaction requires no or minor extra design overhead: it suffices to ensure that the initial task dependencies are respected by the actual execution of the process. Interaction of tasks that have been mapped to different processes require extra attention and special synchronization and/or communication operations need to be inserted in this design phase.



(a) barrier (b) mutual exclusion

Figure 3.8: Synchronization primitives

Figure 3.8 shows two widely used mechanisms for process synchronization. A *barrier* is used to gather all processes together before proceeding with the rest of the execution. Barriers are typically associated with a specific number of processes (say p), so the first $p-1$ processes that reach the barrier will wait until the p-th process reaches that point of execution as well. After that, all processes continue with their execution. *Mutual exclusion* is used to control the concurrency of processes in a code segment called the *critical section*, i.e. access of a common variable as discussed in Algorithm 2. Figure 3.8(b) implies the most typical cases of mutual exclusion where only one process is allowed to enter a critical section. Other alternatives of mutual exclusion may require a specific number of processes entering the critical section, allow some combination of processes entering the critical section (e.g. readers-writers synchronization), or enforce some ordering in the entrance of the section (e.g. enter in the order dictated by the process id's).

All modern languages and tools for parallel programming provide implementations of basic and sometimes more advanced synchronization operations. Programmers can also build upon the provided primitives to develop synchronization constructs that are suitable for the needs of their applications. However, we need to consider that synchronization is one of the major sources of performance degradation in parallel programs, thus we need to avoid unnecessary synchronization operations or select the most efficient of the available ones, even at the design phase.

The second widely applied form of process interaction is communication. Communication is required when processes need to exchange data that are not shared among them. Communication is typically implemented with *messages*, where a sender process sends data to a receiver process. Several alternatives exist here as well, including point-to-point and collective communication (e.g. broadcast or multicast), synchronous or asynchronous communication and others. Again here, the designer needs to consider that communication overheads are responsible for low performance and poor scalability for many classes of parallel applications, so special attention needs to be paid to reduce both the volume of data and the number of messages that need to be exchanged.

Chapter 4

Parallel Programming: Implementation

In this chapter we review concepts and technologies associated with the implementation phase of a parallel program. The design phase has provided a conceptual distribution of a serial algorithm (its computations and data) to a finite set of processes and has pointed out points in the algorithm, where processes need to interact. We now need to translate this design to a working parallel program. The first step is to select the programming model which affects the view of data by the processes. In the next section we discuss the two major choices: shared address space and message passing. The second step is employ parallel programming constructs that express parallelism. This is discussed in Section 4.2. Finally, in Section 4.3 we review existing technologies that are used to implement parallel programs.

4.1 Parallel programming models

There exist two major programming models depending on whether the processes have a common view of the address space or not. When processes can have a common view and access data with typical memory access operations, the programming model is called *shared address space*. When each process has its own, local address space, the model is called *message passing* (because processes need to communicate with a messaging mechanism in order to exchange data). Note that we can observe a clear correspondence of parallel programming models, with the parallel execution platforms discussed in Chapter 1. This is natural, since, as in other features of programming languages supported or influencing hardware design, parallel programming models are constructed to adapt to the underlying execution platforms. Thus, shared address space is inspired by shared memory platforms, while message passing is inspired by distributed memory platforms. Of course, any coupling of programming models and platforms can be implemented with very interesting characteristics.

4.1.1 Shared address space

In the shared address space parallel programming model, processes have a common view of memory (see Figure 4.1). Shared data can be accessed by regular memory operations (e.g. load/store). This model can support all the sharing attributes for data discussed in Chapter 3,

i.e. shared, distributed or replicated. Especially the support for shared data structures is what makes this programming model desirable, since it greatly simplifies the implementation and drastically reduces the coding effort. However, the ability to share data at the existence of data races may lead to programs with subtle and non-repeatable bugs that are sometimes almost impossible to locate. This is what can make programming for shared address space less time-consuming during implementation, but extremely time consuming during debugging.

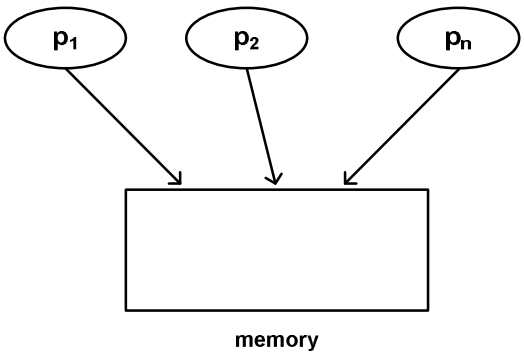


Figure 4.1: Shared addres space

Shared address space programming models can be easily implemented for shared memory platforms. Actually, data sharing mechanisms are supported by operating systems to enable inter-process communication. However, implementing shared address space programming over a distributed memory platform is far from being straightforward. It requires a special software layer (traditionally called distributed shared memory - DSM), hardware support, or both. Additional software layers such as DSM greatly degrade the performance of parallel programs, while hardware extensions are always extremely costly without easily providing the expected performance benefits. For these reasons, shared address space programming models are considered suitable for shared memory platforms and thus carry their major disadvantage: they have limited scalability.

4.1.2 Message passing

To enable parallel programs scale to the number of processes offered by distributed memory systems, the programming model needs to adapt to the constraints imposed by the underlying execution platform. To this direction, the message passing programming model assumes that there is no data sharing between the processes, and thus each process has access only to its local data (see Figure 4.2). Only distributed and replicated data structures can be supported. Process interaction can be realized only by message exchange (hence the name of the model).

The message passing model can be implemented both for shared memory and distributed memory platforms. In the first case, messages are transferred over the interconnection network, while in the second cases messages are exchanged through the physical memory. Programming with the message passing model is tedious and requires a lot of effort to distribute data and orchestrate explicitly the process interaction. On the other hand, the absence of data sharing removes the risk of subtle bugs, making debugging more straightforward, yet in any case much more complicated than sequential programming.

The selection of the programming model is critical for the implementation phase. The programmer needs to consider two important factors when taking this decision: *performance* and

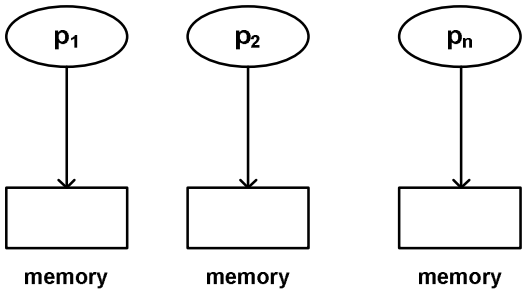


Figure 4.2: Message passing

productivity. Regarding performance, there is no clear superiority of any of the models in shared memory platforms. However, in distributed memory platforms, message passing has been traditionally the only choice. Thus, if we target for a scalable implementation we need to resort to message passing. On the other hand, implementing a program in the shared address space model can be as easy as inserting a couple of keywords in original serial program. If this program does not have peculiar or subtle data races, then the programming task is accomplished with minor overhead. On the other hand, message passing leads to fragmented code and extra programming overhead to distribute data and collect results, being definitely a more burdensome process. However, to make the selection process a little more interesting, bugs are much easier to locate for code written in the message passing model.

4.2 Parallel programming constructs

In this section we discuss the most dominant parallel programming constructs, that is extensions of languages or library support that enable a programmer to "speak a parallel language" and include in a program notions used during the design phase. Key entities described by parallel programming constructs are threads, tasks, task graphs, synchronization operations, parallel loops etc. Typically, a parallel construct may impose a *programming style*, so we use these terms interchangeably.

4.2.1 SPMD

Single Program Multiple data (SPMD) is a parallel programming construct where multiple autonomous threads simultaneously execute the same program at independent points. SPMD is a very common style of parallel programming. In this style the programmer makes use of the process' id to alter its execution path, either differentiating its role as shown in Figure 4.3 for the implementation of a parallel program with one master and many workers, or directing it to operate on different data segments as shown in Figure 4.4 for data parallel matrix-vector multiplication.

SPMD is a straightforward extension of serial, imperative programming. The programmer has full control of the execution path followed by the parallel program. SPMD parallel constructs require minimal additional support by the language or run-time system. Typical operations required by this style may include creation and destroy of threads, request for the thread's id and total number of executing threads in the parallel program, and some synchronization mechanisms like barriers, locks or semaphores. Such operations are usually provided by the

```
if (id == 0) {
/* master code */
}
else {
/* worker code */
}
```

Figure 4.3: SPMD code segment for master and worker roles.

```
work_chunk = N / N_procs;  // N_procs: total number of processes
mystart = id * work_chunk;
myend = min(mystart + work_chunk, N);

for (i=mystart; i<myend; i++)
   for (j=0; j<N; j++)
      y[i]+= A[i][j]*x[j];
```

Figure 4.4: SPMD code segment for data parallel matrix-vector multiplication.

operating system, thus an SPMD API can be implemented by providing to the programmer access to the aforementioned mechanisms. POSIX Threads (Pthreads) and MPI follow the SPMD philosophy, and OpenMP provides a wide set of SPMD-style mechanisms (see Section 4.3).

4.2.2 Fork / Join

Fork and *Join* are the two classical operating system calls to create a new child process (fork) and synchronize it with its parent at the end of its execution (join). In the context of parallel programming, fork and join are used to create and synchronize tasks. Figure 4.5 shows the pseudocode of the parallel calculation of the Fibonacci numbers creating tasks in a recursive way.

```
int fib (int n)
{
   if (n < 2) return n;
   else {
      int x, y;
      x = fork fib(n-1);
      y = fork fib(n-2);
      join;
      return (x+y);
   }
}
```

Figure 4.5: Parallel calculation of Fibonacci numbers using the fork/join construct.

Parallel programming using fork and join provides great flexibility during the execution of an application. New tasks can be created dynamically at request in order to cover the needs of the program. This style is particularly useful for the parallelization of algorithms on irregular data structures (graphs, lists, trees, etc) and especially in the cases where parallelism can be expressed in a recursive way (as in the calculation of Fibonacci numbers in Figure 4.5). Decomposing

the program in a large number of parallel tasks favours portability and efficiency: it allows the execution to adapt to underlying platforms with a different number of processors, and at the same time enables larger opportunities for load balancing. The approach where potential parallelism exceeds hardware parallelism is called *parallel slack*. Parallel slack is an important property to attain high performance through parallelism in applications implemented in the fork/join style. Obviously, there is a correlation between the number of tasks and the amount of work performed by each task: increasing the number of tasks will result in tasks that do less work. There is a point after which reducing the amount of work each task performs, i.e., increasing parallel slack, stops being beneficial for performance.

In an analogy to serial programming languages, the fork/join style resembles more declarative programming: the programmer does not describe *how* the program will be parallelized (which tasks will be assigned to which thread, etc) but *what* parallelism exists in the form of tasks. Obviously, we need run-time language support to orchestrate the execution of tasks into the available threads. A task is similar to an OS process: it represents a computation that can be, for some part, executed independently. This part is large for algorithms that are embarrassingly parallel and small for algorithms with many dependencies. A run-time system needs to manage tasks the same way an OS needs to manage processes. It needs to create them, terminate them, manage their memory, and schedule them to processors. *Task scheduling* is one of the most significant responsibilities of the run-time system in parallel languages that support fork/join constructs. OpenMP and Cilk support fork/join parallel constructs. As for now, this programming style is coupled with the shared address space programming model.

4.2.3 Task graphs

As we discussed in Chapter 3, a task graph is a key data structure when designing a parallel program: it explicitly describes the decomposition of the algorithm into tasks and additionally provides the interaction between the tasks. Clearly, it is a good choice to support parallel programming constructs that are capable of expressing task graphs. Figure 4.6 shows the pseudocode that implements the task graph of Figure 4.7. One needs to explicitly define a starting node (s in this example) and all the nodes of the task graph together with the functions each node will execute. In the sequel, interactions (edges) between nodes are also defined. After the definition of the graph, we can start its parallel execution.

As in the paradigm of fork/join, the run-time system is responsible for the efficient scheduling of tasks to the available threads. The task graph construct enables the programmer to express more complex task interactions. On the other hand, once its execution has started, the graph cannot change, restricting in this way the applicability of this model to applications whose behaviour does not change dynamically. Intel's Thread Building Blocks support this parallel programming construct.

4.2.4 Parallel *for*

The largest part of serial programs is consumed in iterative constructs like for or while loops. Parallelizing such constructs is expected to lead to significant performance optimizations. In particular, parallelization of for-loops has attracted vivid research and technological interest during the last decades. This is due to the fact that for-loops are ubiquitous in time-consuming serial programs (especially in algorithms with regular computations on matrices and computational

```
graph g;
source_node s;  //source node

//each node executes a different function body
node a( g, body_A() );
node b( g, body_B() );
node c( g, body_C() );
node d( g, body_D() );
node e( g, body_E() );
node f( g, body_F() );
node h( g, body_H() );
node i( g, body_I() );
node j( g, body_J() );

//create edges
make_edge( s, a );
make_edge( s, b );
make_edge( s, c );
make_edge( a, d );
make_edge( b, e );
make_edge( c, f );
make_edge( d, h );
make_edge( e, h );
make_edge( f, i );
make_edge( h, j );
make_edge( i, j );

s.start(); // start parallel execution of task graph
g.wait_for_all();  //wait for all to complete
```

Figure 4.6: Pseudocode for the creation and execution of a task graph.

grids) and are rather easy to handle either at compiler or at run-time. In addition, the parallelization of for-loops can be applied directly on serial code, bypassing to a large extend most of the effort-consuming steps in design and implementation. For these reasons languages and libraries for parallel computing (e.g. OpenMP and Cilk) support the *parallel for* construct that translate a serial loop to a parallel one, i.e. all iterations of the loop are considered independent and distributed to the executing threads in a proper way. Figure 4.8 shows the pseudocode for the parallel matrix-vector multiplication using parallel-for.

In its basic functionality, the parallel-for construct can be considered as a syntactic sugar that simplifies the SPMD style of Figure 4.4, performing statically a source-to-source translation. However, parallel-for constructs can become more powerful by supporting dynamic scheduling of iterations to threads to achieve load balancing, or handling each iteration of the loop as a separate task and schedule it using a more sophisticated task scheduler (utilizing in this way the available parallel slack).

However, one needs to be extremely cautious with the use of the parallel-for construct. It is up to the programmer to decide whether a for-loop is parallel, i.e. all its iterations can be legally executed independently. The research work on parallelizing compilers has provided a wide theory to aid towards proving that a loop is parallel. In typical cases, however, the programmer is able to decide whether a loop is parallel or not by simple inspection. If, nevertheless, the

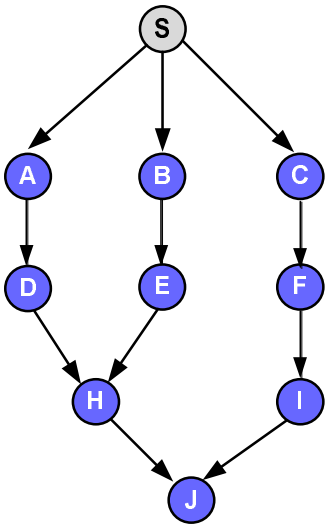


Figure 4.7: Task graph

```
forall (i=0; i<N; i++)
   for (j=0; j<N; j++)
      y[i]+= A[i][j]*x[j];
```

Figure 4.8: Code segment for data parallel matrix-vector multiplication using the *parallel for* construct.

programmer is uncertain then he/she should act conservatively and leave the for-loop executed serially in order to preserve the original semantics of the application.

4.3 Languages, libraries and tools

4.3.1 POSIX Threads (Pthreads)

POSIX Threads, usually referred to as Pthreads, is a POSIX standard for threads that defines an API for creating and manipulating threads. Implementations of the API are available on many Unix-like POSIX-conformant operating systems. Programming with Pthreads is very similar to system programming, using low-level system calls provided by the operating system. Pthreads programs follow the shared address space programming model, making use of the fork/join constructs. The entity that is manipulated is the thread. Assigning tasks to threads is the programmer's responsibility. The library provides functions to create, join and synchronize threads. Pthreads provide a powerful, low-level mechanism for parallel processing in shared memory platforms, and they grant full control to the programmer. On the other hand, programming the Pthreads is quite counter productive.

4.3.2 OpenMP

OpenMP (Open Multi-Processing) is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most processor architectures and operating systems. It consists of a set of compiler directives, library routines, and environment variables

that influence run-time behavior. OpenMP follows the shared address space programming model and supports the SPMD, parallel for, and fork/join parallel constructs. OpenMP is a very popular API that greatly simplifies parallel programming for shared memory systems. OpenMP directives can be easily inserted in serial programs to inject parallelism in a very productive way. The OpenMP 3.0 standard released in 2008 provides support for irregular parallelism with the creation and manipulation of tasks.

4.3.3 Cilk

The Cilk language is an extension of C for supporting shared-memory parallel programming following the fork-join model. The language provides two basic keywords for expressing parallelism within a program, spawn and sync; the former is used to indicate procedures that can be executed in parallel, and the latter provides barrier synchronization for previously spawned procedures. At the low level, the Cilk runtime generates and manipulates parallel tasks, employing task stealing as a key mechanism for dynamically balancing the load across processors. A common parallelism pattern found in many Cilk programs is that of recursive parallelization, where a problem is recursively subdivided into smaller, potentially parallel sub-problems. Under certain occasions, this feature can facilitate the development of cache-oblivious parallel algorithms. Cilk has been developed since 1994 at the MIT Laboratory for Computer Science. Its commercial version supports both C and C++ and is distributed by Intel under the name Intel Cilk Plus.

4.3.4 Threading Building Blocks

Threading Building Blocks (TBBs) is a C++ template library developed by Intel for writing multithreaded programs. Much like the serial algorithms found in C++ Standard Template Library (STL), TBBs provide a rich set of algorithmic skeletons for expressing parallelism at different forms and levels. Typical constructs are the parallel_for, parallel_reduce and parallel_scan algorithms. Additionally, the library supports more advanced structures (e.g. classes for expressing pipeline parallelism and flow graphs), concurrent data structures, as well as a low-level tasking interface. At its heart, the TBBs runtime system manipulates tasks, and adopts Cilk's major concepts for dealing with load balancing and locality, i.e. task stealing and recursion. Unlike Cilk, the use of templates and function objects make it possible to write parallel programs without special compiler support. For example, one can transform an STL for_each loop into a parallel one by simply replacing it with the parallel_for construct.

4.3.5 Java threads

The performance available from current implementations of Java is not as good as that of programming languages more typically used in high-performance computing. However, the ubiquity and portability of Java make it an important platform. Java supports the creation of different threads in a shared address space model. The language also provides a large number of mechanisms to orchestrate synchronized access to shared data.

4.3.6 Message passing interface (MPI)

The Message Passing Interface (MPI) is a standardized and portable library designed by a group of researchers from academia and industry to function on a wide variety of parallel computers.

As evident by its name, MPI follows the message-passing programming model, and is extensively used for parallel programming. In fact, the vast majority of large scale enterprise simulations running in supercomputers are implemented with MPI. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in Fortran, C and C++. Several well-tested and efficient implementations of MPI include some that are free and in the public domain.

MPI programs follow the SPMD style. MPI processes are generated in various processing nodes of a parallel platform, start their autonomous execution on distributed data and communicate explicitly with messages. MPI supports a large number of communication primitives (point-to-point and collective), and a number of routines to manage processes, organize them in logical groups and topologies etc. Originally in its first standard, MPI supported only static generation of processes at the beginning of the program execution. To support more irregular and dynamic parallelism the MPI 2.0 standard included dynamic process management. In addition, the MPI 2.0 standard supports one-sided communication and parallel I/O for concurrent, efficient access to shared files.

4.3.7 PGAS languages

Partitioned global address space (PGAS) parallel languages assume a global memory address space that is logically partitioned and a portion of it is local to each processor. The novelty of PGAS is that the portions of the shared memory space may have an affinity for a particular thread, thereby exploiting locality of reference. The PGAS model is the basis of Unified Parallel C, Co-array Fortran, Titanium, Fortress, Chapel and X10. The goal here is to keep the best of both worlds: the scalability of the message-passing model and the productivity of the shared address space model.

Chapter 5

Programming for Shared Memory

In this chapter we discuss issues related to the implementation of parallel programs on shared memory platforms. As mentioned earlier, shared memory architectures provide a common view of the address space to the processors and thus easily support shared address space as a programming model. Common access to shared data is a powerful programming feature that greatly simplifies parallel programming. However, the programmer needs to keep in mind that in several cases this simplicity comes at the cost of performance. To develop efficient programs in shared-memory architectures, the programmer needs to understand several hardware and architectural features that affect performance. We briefly review some hardware concepts in the next paragraph and continue with a discussion on performance issues affecting data sharing and synchronization.

5.1 Hardware concepts

In Figure 1.3(a) we showed a simple organization of a shared memory platform. Based on the performance improvements of caches on the execution of serial programs, computer architects have included local caches in each processor of a shared memory system as well. This design choice would not affect program execution, if serial programs were to be scheduled on the system only. However, when parallel execution is concerned, then in the existence of caches several copies of data may exist in main memory and in the local caches of the processors. Without any additional support, processors would have an inconsistent view of shared data, a fact that would greatly limit the power of such platforms, that being the ability to provide a common view of memory space. *Cache coherence* is intended to manage data conflicts and maintain consistency between cache and memory.

Cache coherence defines the behavior of reads and writes to the same memory location. The coherence of caches is obtained if the following conditions are met:

1. A read made by a processor P to a location X that follows a write by the same processor P to X, with no writes of X by another processor occurring between the write and the read instructions made by P, X must always return the value written by P. This condition is related with the program order preservation, and this must be achieved even in monoprocessed architectures.

2. A read made by a processor P1 to location X that follows a write by another processor

P2 to X must return the written value made by P2 if no other writes to X made by any processor occur between the two accesses. This condition defines the concept of coherent view of memory. If processors can read the same old value after the write made by P2, we can say that the memory is incoherent.

3. Writes to the same location must be sequenced. In other words, if location X received two different values A and B, in this order, by any two processors, the processors can never read location X as B and then read it as A. The location X must be seen with values A and B in that order.

Caches in modern multiprocessor systems are augmented with special hardware that keeps the cache blocks coherent. This is done by the implementation of a *cache coherence protocol* which maintains a Finite State Machine for each cache block. Bus traffic is snooped and cache blocks' states change according to the protocol. Figure 5.1 shows the state transitions and the consequent bus transactions in the MESI cache coherence protocol. The responsibility of any cache coherence protocol is to ensure that all the processors' caches share data from system memory properly and do not use stale data that has been modified in another processor's cache. In other words, the protocol makes it possible for all processors to work like they are all connected directly to a single, globally shared memory module, while actually working with caches and multiple cached copies of data. This abstraction of shared memory provided by the protocol eases parallel programming significantly, since the parallel threads of an application can refer directly to memory locations (just as in sequential programs), but may hide serious performance pitfalls.

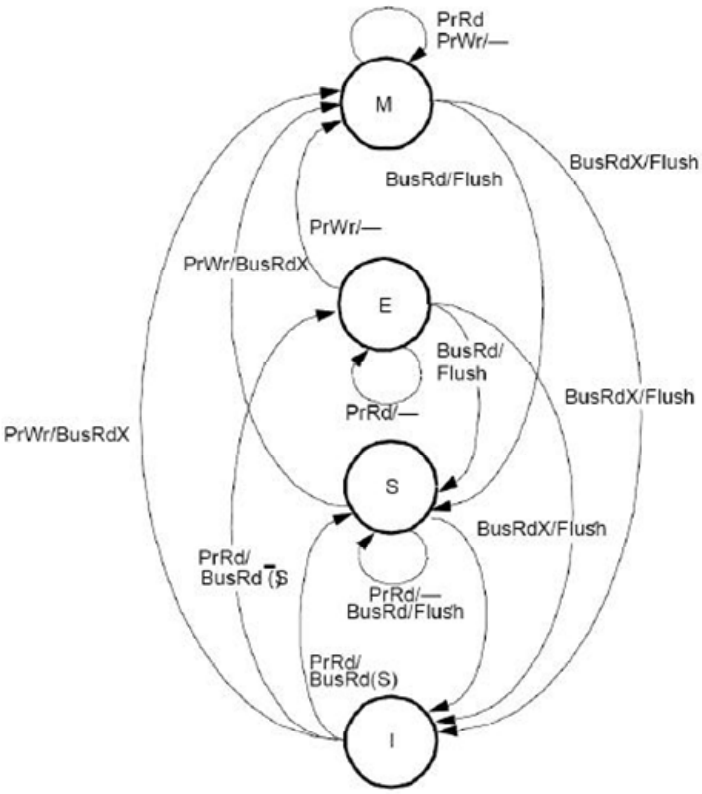


Figure 5.1: State transitions and bus transactions in the MESI cache coherence protocol

5.2 Data sharing

Excessive read-write sharing between processors on a centralized data structure can introduce a large amount of traffic on the bus. This is because on every write operation a processor has to invalidate all existing copies of the cache line holding the data structure, and on subsequent reads from other processors the modified cache line has to be transferred to their caches. This "ping-pong" effect usually introduces large performance penalties, not only because of the actual data transfers but also due to the large amount of protocol control messages being exchanged. The cost associated with each such read or write operation can be many times larger than an ordinary read or write to private data, and increases with the number of sharers, the distance (in terms of processor topology) of the requestors, the frequency of read-write operations, etc. Typical examples of variables being heavily shared across processors under a read-write pattern are reduction and synchronization variables.

Another common issue is when threads on different processors modify variables that happen to reside on the same cache line. This problem is known as *false sharing*, because it occurs unintentionally (i.e. threads do not actually access the same variable), but has all the performance drawbacks of true readwrite sharing discussed above. While the programmer cannot usually do many things to avoid or eliminate true sharing at the application level, he can do much more to remedy false sharing, as we will discuss in following sections.

Interesting implications in data sharing occur in modern multicore platforms that have a typical organization as shown in Figure 5.2. Note that this organization may couple with both UMA and NUMA memory organizations that were discussed in Chapter 1. A multicore system consists of one or more processors (often called packages) that are based on a multicore architecture, thus incorporating more than one processing cores in each package. Typically, there may also exist some form of cache sharing between the cores within one package (intra package cache sharing is not common among multicore platforms). Especially in the existence of a NUMA technology to access main memory, the placement of threads in such platform organization may significantly affect performance.

Consider the scenario where two threads are placed on the same package and additionally share some level of cache memory. Depending on the memory access patterns this sharing can be *constructive*, because for example the two threads read the same data structures that fit in the shared cache and actually one loads data the will be used by the other, or *destructive*, because the one thread's memory accesses evict from the cache the other thread's data (due to conflict or capacity misses) dramatically increasing the overall miss rate. Now consider the scenario where two threads are places on different packages and share some read-only data structures. The execution may end up in an excessive cache-to-cache data transfer between the two packages, which can potentially create hotspots in the memory bus and degrade performance.

Overall, data sharing in a shared memory architecture is well supported providing a powerful programming mechanism. However, architectural details may impose subtle overheads to the execution that are not easily traced by the programmer. One needs to keep in mind that, as in several other aspects of computer science, programming convenience may come at the cost of performance overhead. As multicore architectures will become more complex, with larger number of cores, more sophisticated data access channels and deeper cache hierarchies, the programmer may need to drastically reduce the amount of data that need to be shared and resort to programming techniques that are enforced by distributed-memory platforms like data distri-

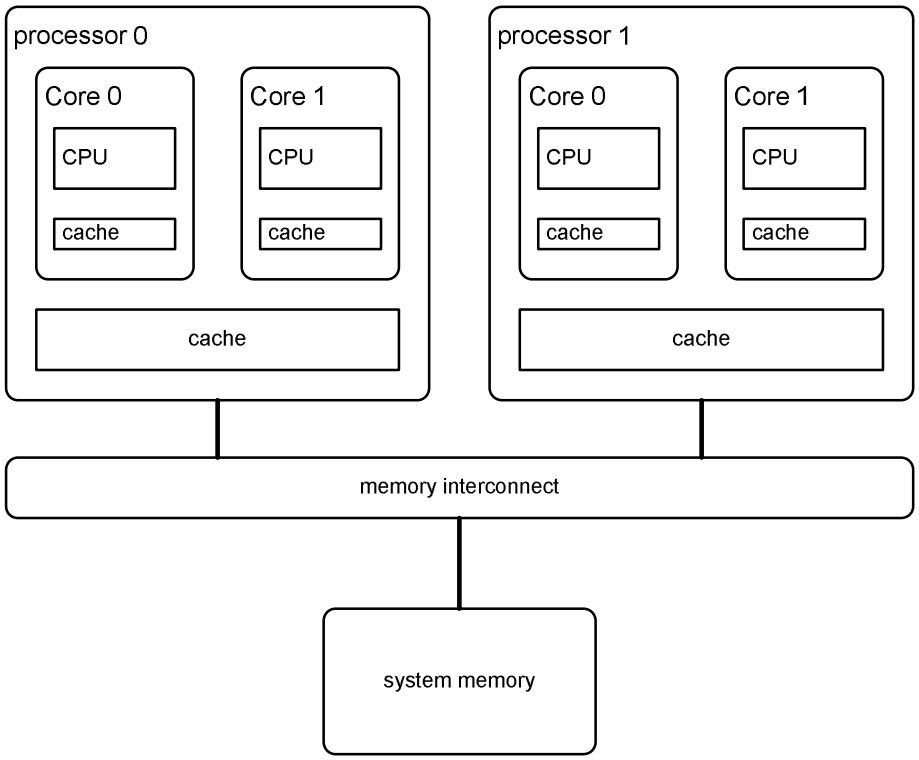


Figure 5.2: A typical organization of modern multicore platforms

bution and replication. In the next paragraph we discuss another important aspect with data sharing, that of synchronization, which may also impose significant overhead to the execution of an application.

5.3 Synchronization

Parallel applications use locks to synchronize entry to program regions where access should be atomic, usually to protect some shared resource from concurrent updates, such as a shared variable or an output file. Such regions are known as critical sections. While a thread is inside a critical section no other thread can enter, implying that all threads requesting access should wait. As a result the execution of critical sections is serialized. There are many parameters of lock usage and implementation that can affect performance of a parallel application. Below we mention the most important of them:

- *critical section extent*: Since critical sections constitute a serialization point in parallel execution, they should be as small as possible to reduce the amount of time other threads sit idle waiting to enter the critical section and to therefore allow the application to scale efficiently.

- *lock contention*: If there is high demand among threads for accessing the resource protected by a critical section, there could be simultaneously many threads contending for the corresponding lock. In the best case, the waiting time for a thread could scale linearly with the total number of competing threads. In practice, however, things can be even worse, as the

period a thread spends waiting can introduce significant coherence traffic due to read-write sharing on the lock variable.

- *locking overhead*: The operations to acquire and release a lock entail by themselves some measurable overhead. In general, there are two main classes of lock implementations with different characteristics with respect to cost and scalability: user-level spinning and OS-level blocking. In the first case, threads poll continuously the value of a user-level lock variable until it appears to be free. Spin-based locks are efficient and thus preferable for small thread counts (e.g. less than 10) and short critical sections. Because they operate entirely at user space, the acquire and release operations have low latency and are easier to implement. However, in critical sections which incur long waiting times (i.e., large extent and/or high contention) they can introduce notable coherence traffic. The same happens also for large thread counts. In these cases spin-based locks do not scale well, and OS-based ones may perform better. A thread that uses an OS-based lock goes to sleep in kernel mode if it fails to acquire the lock. This action, as well as the action of waking up a sleeping thread comes at the cost of executing a system call. It is harder to implement, it penalizes short critical sections (e.g., critical sections with execution time almost equal or less than the invocation time of acquire/release operations) but it is much more efficient and scalable for long-term waiting and large thread counts, since waiters do not consume resources (CPU, memory).

- *lock granularity*: The granularity is a measure of the amount of data the lock is protecting. In many cases, the programmer can have the freedom of choice on how to implement synchronized access on a data structure between coarse-grain schemes and fine-grain ones. In the former, a small number of locks is utilized to protect large segments in the data structure. This approach is easy in its conception and implementation, but entails a lot of performance drawbacks, i.e. large critical sections and increased likelihood for contention. Conversely, finegrain schemes use many locks to protect small pieces of data. They are usually more difficult and error-prone to program, but yield smaller critical sections with reduced contention. However, such small critical sections, even completely uncontended, can introduce overhead in two ways: first, because of the additional bus traffic due to the large number of lock variables, and second, because of the increased locking overhead associated with each memory operation on the protected data structure (relative to a coarser-grain scheme). Finding the best granularity level is not trivial and requires considering all the aforementioned factors.

5.4 Memory bandwidth saturation

Memory access can still become a serious bottleneck even when threads of a parallel application work mostly on private data, without incurring notable interprocessor traffic. This is particularly true for memory-intensive parallel applications with large working sets (e.g. streaming applications), which may suffer from memory bandwidth saturation as more threads are introduced. In such cases, the application will not probably scale as expected, however efficiently parallelized it is, and the performance will be rather poor.

Each socket in the platform has a maximum bandwidth to memory which is shared by all processing elements it encompasses (cores, hardware threads, etc.). Depending on the architecture,

even multiple sockets might share access to main memory through a common bus. Given that even under perfect conditions (e.g. full software optimizations) the memory subsystem cannot fulfill a single thread's requests without having its core stalled, we can imagine the amount of pressure put on memory bus and how quickly it can be saturated when the number of cores increases. This is why the processor industry strives to provide improved bus speeds or alternate memory paths on each new processor generation (e.g. through Non-Uniform Memory Access designs), but unfortunately these enhancements have never been enough to make the memory subsystem keep pace with the increasing core counts.

Chapter 6

Programming for Distributed Memory

6.1 Hardware concepts

Distributed memory are the most generic parallel execution platforms. As shown in Figures 1.4 and 1.5 distributed memory systems are built by connecting single or multicore nodes through an interconnection network. Modern distributed memory systems are typically based on multicore nodes (Figure 1.5). This architectural approach has two significant effects on the implementation and execution of parallel applications. First, the system does not support a global, common view of the program's address space, and thus the programmer needs to implement parallel programs in a fragmented way, using the message-passing model (e.g. with MPI). Each parallel process has its own, local address space and thus data need to be distributed among the physical memories of participating nodes[1]. Second, despite the great advancements in interconnect technology, data transfers between the system nodes are time-consuming and thus, communication is the major bottleneck in large classes of applications. Similar to the well known CPU-memory gap, there is also a CPU-interconnect gap. In the following paragraphs we further discuss these issues together with the implications created by resource sharing in distributed memory systems with multicore nodes.

6.2 Data distribution

There exist two options when partitioning data for distributed memory: data distribution and data replication. Replicating read-only data eases programming but may create heavy and possibly unnecessary initialization traffic, and increase the total working set during execution, with a negative effect on cache utilization and memory traffic. A good compromise would be to replicate "small" read-only data structures and distribute larger ones, ensuring that each node maintains its read set locally. Imposing communication traffic for read-only data during application execution should be avoided, unless absolutely necessary (e.g. the data structure does not fit in main memory).

[1] Current parallel language trends try to break this paradigm by implementing partitioned, global address space (PGAS).

More critical decisions need to be taken for data that are both read and written during application execution. If the read and write operations are performed by different processes, this creates a need for inter-process communication. Typically, data are allocated to the writer's main memory (*computer-owns rule*), so we need to insert communication primitives that implement data sending from the writer towards the reader. The key goal in this case is to allocate write sets in a way that inter-process communication is minimized.

A simple example of how data distribution policies affect the communication is demonstrated in Figure 6.1. In this case a two dimensional matrix A[L][2L] is distributed among four processes by row (on the left) or by column (on the right). The matrix is used to store in-place computations where each matrix element is iteratively updated using the values of its four neighbours. This is a typical computation in computational science to solve partial differential equations. This form of computations creates the "nearest neighbour" communication pattern. In the case where data are distributed by rows this creates data exchange upwards and downwards, while in the case where data are distributed by columns this creates data exchange towards left and right. Communication data are designated in grey. Clearly, the first approach is less communication efficient, since it leads to double the amount of data that need to be exchanged (12L vs. 6L).

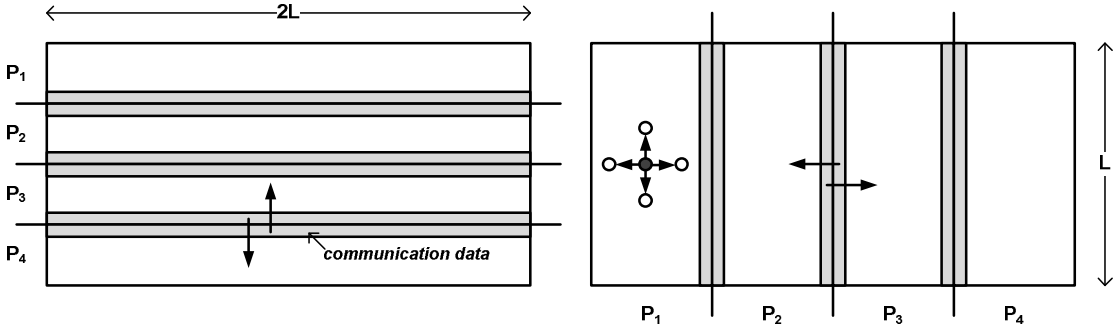


Figure 6.1: Difference in communication volume for two data distribution schemes.

6.3 Communication

Minimizing the communication overhead should be one of the primary concerns when designing and implementing a parallel application for distributed memory. This effort affects all steps in the development process, from the highest algorithmic level, where one can select between algorithms that reduce or even eliminate communication, to the level of implementation and system tuning. The most important techniques to mitigate the communication overhead are as follows:

- *Bulk communication*: Recall that the communication time T_{comm} for a message with size m is $T_{comm} = t_s + mt_w$ where t_s is the startup time and t_w the per-word transfer time. If we are able to coalesce a number of messages into a single message, then we are able to drastically reduce the effect of the startup communication latency. This is especially critical in commodity interconnection networks, where the startup latencies are quite high.

- *Point-to-point vs. collective*: Collective communication is a neat approach that can greatly facilitate communication between parallel processes. However, this mode of communication should be used sparingly and only when absolutely necessary, since it is very time consuming

and not scalable. In modern supercomputing platforms with thousands or even millions of cores, broadcasting information even with the most efficient logarithmic path, may lead to significant performance overheads.

- *Communication to computation overlapping*: The idea behind this approach is to perform computation and communication in parallel. Figure 6.2 shows the standard mode of communication, where the process communicates and computes in a serialized manner. To overlap computation and communication we can initiate communication before computation and finalize it after the completion of computation, as shown in Figure 6.3. However, two important issues need to be pointed out: first, in the majority of the cases this overlapping is not straightforward since the two operations may be interdependent, i.e. the computations need the provided by the communication operation. In this case the algorithm needs some redesign to enable this overlapping. Second, this offloading of communication implies that the operations involved in the communication function are offloaded to some other piece of hardware. The typical case is that a sophisticated Network Interface Card (NIC) with a special network processor can undertake the communication operations. In an alternative scenario, communication can be offloaded to a different core in a multicore platform.

```
while (1){
    communicate();
    compute();
}
```

Figure 6.2: Standard mix of computation and communication.

```
while (1){
    initialize_communication();
    compute();
    finalize_communication();
}
```

Figure 6.3: Overlapping computation with communication

6.4 Resource sharing

Modern distributed memory systems follow the hybrid design, where multiple multicore nodes are interconnected to build a scalable, large scale system (see Figure 1.5). In this architecture, the memory hierarchy and the I/O bus are shared among the cores of each node. Recall from previous discussions that CPU technology is advancing more rapidly than memory and interconnect technology, so these "slow" resources are being shared by a number of "fast" processing cores. If the algorithm is not CPU intensive, but is rather memory of communication intensive, then this sharing is expected to lead to reduced scalability when all cores a utilized. This issue is expected to be more severe in the future, as CPU/memory and CPU/interconnect gaps increase and the number of cores included within one node increases as well.

Figure 6.4 demonstrates a frequent performance behaviour of parallel applications when executed in hybrid platforms. In the chart we assume that the system constitutes of P nodes with

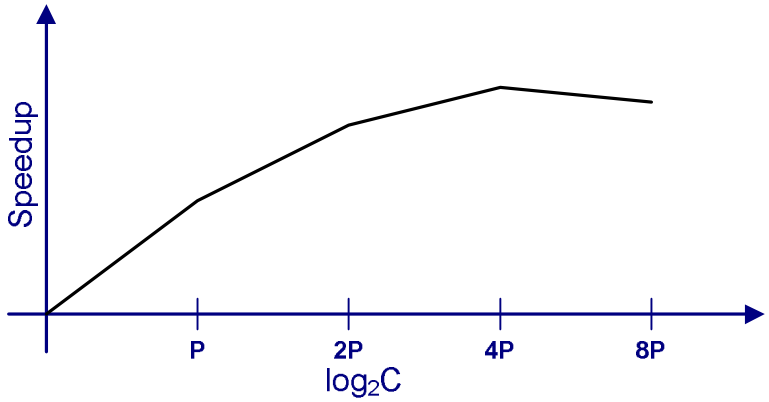


Figure 6.4: Reduced scalability due to resource sharing: as more cores within one node participate in the execution, the scaling of the applications drops.

eight cores each. In the x-axis (logarithmic scale) we represent the number of cores (C) participating in the execution, while in the y-axis we represent the achieved speedup. In the first P cores, only one core per node is used. At this point additional processes are allocated cyclically to gradually fill the whole system at $8P$. We may observe that as more cores are activated in each node, the scalability of the application drops, resulting in even performance degradation due to resource contention. Depending on the memory intensity and the communication needs of the application, the scalability of the algorithm may be even worse.

Resource sharing is a significant issue in modern HPC platforms. Poor resource utilization is the main reason why extremely powerful supercomputers may fail to deliver they high capacity to user applications. In several cases, the combination of algorithm and underlying execution platform, may pose hard limits to the upper bounds of the achieved performance. Current research trends propose the drastic redesign of applications and system software in order to effectively utilize the computing resources offered by existing and future systems. This redesign may affect diverse applications from hand-held computers up to exascale supercomputers.