Chapter 1

Software Technologies

Ian Foster & Ken Kennedy

1.1 Introduction

While parallel computing is defined by hardware technology, it is software that renders a parallel computer usable. Parallel software is the topic of both this overview chapter and the ten more comprehensive chapters that comprise Part IV of this book.

The concerns of the parallel programmer are those of any programmer: algorithm design, convenience of expression, efficiency of execution, ease of debugging, component reuse, and lifecycle issues. Hence, we should not be surprised to find that the software technologies required to support parallel program development are familiar in terms of their basic function. In particular, the parallel programmer, like any programmer, requires: languages and/or application programming interfaces (APIs) that allow for the succinct expression of complex algorithms, hiding unimportant details while providing control over performance-critical issues; associated tools (e.g., performance profilers) that allow them to diagnose and correct errors and performance problems; and convenient formulations of efficient algorithms for solving key problems, ideally packaged so that they can easily be integrated into an application program.

However, despite these commonalities, the particular characteristics of parallel computers and of parallel computing introduce additional concerns that tend to complicate both parallel programming and the development of parallel programming tools. In particular, we must be concerned with the following three challenges

1. *Concurrency and communication.* Parallel programs may involve the creation, coordination, and management of potentially thousands of independent threads of control. Interactions between concurrent threads of control may result in nondeterminism. These issues introduce unique con-

cerns that have profound implications for every aspect of the program development process.

2. *Need for high performance.* In sequential programming, ease of expression may be as important or even more important than program performance. In contrast, the motivation for using parallel computation is almost always a desire for high performance. This requirement places stringent constraints on the programming models and tools that can reasonably be used for parallel programming.

3. *Diversity of architecture.* The considerable diversity seen in parallel computer architectures makes the development of standard tools and portable programs more difficult than is the case in sequential computing, where we find remarkable uniformity in basic architecture.

The role of parallel software is thus to satisfy the requirements listed at the beginning of this section, while simultaneously addressing in some fashion the three challenges of concurrency and communication, performance demands, and architectural diversity. This is a difficult task, and so in practice we find a variety of approaches to parallel software, each making different tradeoffs between these requirements.

In the rest of this chapter, we provide an overview of the major software and algorithmic technologies that we can call upon when developing parallel programs. We structure the presentation in terms of the three key questions that we believe will be asked by any parallel programmer:

- *How do I select the parallel programming technology (library or language) to use when writing a program?* We introduce the programming models, APIs, and languages that are commonly used for parallel program development, and provide guidance concerning when these different models, APIs, and languages may be appropriate.

- *How do I achieve correct and efficient execution?* Here, we discuss issues relating to nondeterminism and performance modeling.

- *How do I reuse existing parallel algorithms and code?* Here, we provide a roadmap to the parallel algorithms described in this book and describe several techniques used to achieve code reuse in parallel algorithms.

In each case, we provide pointers to the chapters in which these issues are discussed at greater length.

> *This chapter does not say anything about AD or parallel file systems.*

1.2 Selecting a Parallel Program Technology

As was explained in Chapter **??**, a parallel computer is a collection of processing and memory elements, plus a communication network used to route requests and

information among these elements. The task of the parallel programmer is to coordinate the operation of these diverse elements so as to achieve efficient and correct execution on the problem of interest.

The performance of a parallel program is determined by how effectively it maximizes *concurrency* (the number of operations that can be performed simultaneously) while minimizing the amount of *communication* required to access "nonlocal" data, transfer intermediate results, and synchronize the operation of different threads of control. Communication costs are frequently sensitive to *data distribution*, the mapping of application data structures to memory elements: a good data distribution can reduce the number of memory accesses that require expensive communication operations. If work is not distributed evenly among processors, *load imbalances* may occur, reducing concurrency and performance.

When evaluating the correctness of a parallel program, the programmer may need to take into account the possibility of *race conditions*, which occur when the executions of two or more distinct threads of control are sufficiently unconstrained that the result of a computation can vary nondeterministically, depending simply on the speed at which different threads proceed.

The programmer, when faced with the task of writing an efficient and correct parallel program, can call upon a variety of parallel languages, compilers, and libraries, each of which implements a distinct programming model with different tradeoffs between ease of use, generality, and achievable performance.

In the rest of this section, we first review some of the principal programming models implemented by commonly used languages and libraries. Then, we examine each of these languages and libraries in turn and discuss their advantages and disadvantages.

1.2.1 Parallel Programming Models

We first make some general comments concerning the programming models that underly the various languages and libraries that will be discussed subsequently.

Thirty years of research have led to the definition and exploration of a large number of parallel programming models [2]. Few of these models have survived, but much experience has been gained in what is useful in practical settings.

Data parallelism vs. task parallelism. Parallel programs may be categorized according to whether they emphasize concurrent execution of the same task on different data elements (*data* parallelism) or the concurrent execution of different tasks on the same or different data (*task* parallelism). For example, a simulation of galaxy formation might require that essentially the same operation be performed on each of a large number of data items (stars); in this case, a data parallel algorithm is obtained naturally by performing this operation on multiple items simultaneously. In contrast, in a simulation of a complex physical system comprising multiple processes (e.g., a multidisciplinary optimization of an aircraft might couple airflow, structures, and engine simulations) the different components can be executed concurrently, hence obtaining task parallelism.

Most programs for scalable parallel computers are data parallel in nature,

for the simple reason that the amount of concurrency that can obtained from data parallelism tends to be larger than can be achieved via task parallelism. Nevertheless, task parallelism can have an important role to play as a software engineering technique: it often makes sense to execute distinct components on disjoint sets of processors (or even on different computers) for modularity reasons. It is increasingly common for parallel programs to be structured as a task-parallel composition of data-parallel components.

Explicit vs. implicit parallelism. Parallel programming systems can be categorized according to whether they support an explicitly or implicitly parallel programming model. An *explicitly* parallel system requires that the programmer specify directly the activities of the multiple concurrent "threads of control" that form a parallel computation. In contrast, an *implicitly* parallel system allows the programmer to provide a higher-level specification of program behavior in which parallelism is not represented directly. It is then the responsibility of the compiler or library to implement this parallelism efficiently and correctly.

Implicitly parallel systems can simplify programming by eliminating the need for the programmer to coordinate the execution of multiple processes. For example, in the implicitly parallel, primarily data-parallel language High Performance Fortran, the programmer writes what is essentially sequential Fortran 90 code, augmented with some directives. Race conditions cannot occur and the HPF program need not be rewritten to take advantage of different parallel architectures.

Explicitly parallel systems provide the programmer with more control over program behavior and hence can often be used to achieve higher performance. For example, an MPI implementation of an adaptive mesh refinement algorithm may incorporate sophisticated techniques for computing mesh distributions, for structuring communications among subdomains, and for redistributing data when load imbalances occur. These strategies are beyond the capabilities of today's HPF compilers.

A parallel programming style that is becoming increasingly popular is to encapsulate the complexities of parallel algorithm design within libraries (e.g., an adaptive mesh refinement library, as just discussed). An application program can then consist of just a sequence of calls to such library functions, as illustrated in Figure **??** below. In this way, many of the advantages of an implicitly parallel approach can be obtained within an explicitly parallel framework.

Shared memory vs. distributed memory. Explicitly parallel programming systems can be categorized according to whether they support a shared or distributed memory programming model. In a *shared memory* model, the programmer's task is to specify the activities of a set of processes that communicate by reading and writing shared memory. In a *distributed memory* model, processes only have local memory and must use some other mechanism (e.g., message passing or remote procedure call) to exchange information.

Shared memory models have the significant advantage that the programmer need not be concerned with data distribution issues. On the other hand, high-

Table 1.1: Major parallel programming technologies discussed in this book

Name	Model	.	...

```
   MPI Version of
Some Simple Application
```

Figure 1.1: An MPI formulation of the XX problem

performance implementations may be difficult on computers that lack hardware support for shared memory, and race conditions tend to arise more easily.

Distributed memory models have the advantage that programmers have explicit control over data distribution and communication; thisc control facilitates high-performance programming on large distributed memory parallel computers.

1.2.2 Parallel Programming Technologies

Table 1.1 lists and categorizs the major programming technologies discussed in this book. We provide here a brief summary of each and provide pointers to the chapters where they are covered in more detail. In the next subsection, we discuss the situations in which each is to be preferred.

We should include cross-references to application chapters as case studies for various of these approaches.

Message Passing Interface

The Message Passing Interface (MPI) is a specification for a set of functions for managing the movement of data among sets of communicating processes. Official MPI bindings are defined for C, Fortran, and C++; bindings for various other languages have been produced as well. MPI defines functions for point-to-point communication between two processes, for collective operations among processes, for parallel I/O, and for process management. In addition, MPI's support for *communicators* facilitates the creation of modular programs and reusable libraries. As illustrated in Figure 1.1, MPI programs are commonly implemented in terms of a Single Program Multiple Data (SPMD) model, in which all processes execute essentially the same logic. Chapter **??** provides more details on MPI, while the parallel I/O aspects of MPI are discussed in Chapter **??**.

Parallel Virtual Machine (PVM) represents another popular instantiation of the message passing model which however has been largely superseded by MPI.

Analysis. MPI is today *the* technology for constructing scalable parallel

```
   P-Threads Version of
Some Simple Application
```

Figure 1.2: A P-threads formulation of the XX problem

programs: no other technology can beat it for portability and scalability. In addition, a significant body of MPI-based libraries has emerged that provide high-performance implementations of commonly used algorithms. Nevertheless, other technologies may be appropriate if our goal is a modestly parallel version of an existing program (in which case OpenMP may be appropriate), we are using Fortran 90 (HPF), or our application is a task-parallel composition designed to execute in a distributed environment(CORBA, RMI).

Parallelizing Compilers

On many small (primarily shared memory) parallel computers, parallelizing compilers are provided that can extract a certain amount of parallelism from sequential code. The performance gains that can be expected from this technology are application dependent but are generally small. Programmer-supplied information (typically communicated via directives) can improve things in some situations. However, in that case you should be considering OpenMP.

Analysis. Parallelizing compilers are certainly worth trying when available if only a small degree of parallelism is required. Otherwise OpenMP or MPI are better solutions.

P-threads

As noted above, in the shared memory programming model, multiple threads of control operate in a single memory space. The POSIX standard threads package (P-threads) represents a particularly low level but widely available implementation of this model. As illustrated in Figure 1.2, the P-threads library provides functions for creating and destroying threads and for coordinating thread activities via constructs designed to ensure exclusive access to selected memory locations (locks and condition variables). Chapter **??** provides a more detailed discussion of P-threads.

Analysis. We do *not* recommend the use of P-threads for parallel program development. The unstructured nature of P-threads constructs makes the development of correct and maintainable programs difficult. In addition, P-threads programs are not scalable to large numbers of processors.

```
   OpenMP Version of
Some Simple Application
```

Figure 1.3: An OpenMP formulation of the XX problem

OpenMP

An alternative approach to shared-memory programming is to use more structured constructs such as parallel loops to represent opportunities for parallel execution. This approach is taken in the increasingly popular OpenMP, a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs. As illustrated in Figure 1.3, OpenMP extensions focus on the exploitation of parallelism within loops. This parallelism may be fine-grained (as in the example) or coarse grained (if loops call computationally expensive functions). A desirable feature of OpenMP is that it preserves sequential semantics: ignore the structured comments and a sequential program is obtained. This simplifies program development, debugging, and maintenance. Chapter **??** provides a more detailed discussion of OpenMP.

We should say something about determinism or lack of it.

Analysis. We recommend the use of OpenMP when the goal is to achieve modest parallelism on a shared memory computer. In this environment, the simplicity of the OpenMP model and the fact that a parallel program can be obtained via the incremental addition of directives to a sequential program are significant advantages. On the other hand, the lack of support for user management of data distribution means that scalable implementations of OpenMP are unlikely to appear in the foreseeable future.

High Performance Fortran

High Performance Fortran (HPF), like OpenMP, extends a sequential base language (in this case Fortran 90) with a combination of directives, library functions, and (in the case of HPF) some new language constructs to provide a data-parallel, implicitly parallel programming model. HPF differs from OpenMP in its focus on support for user management of data distribution, so as to support high-performance execution on scalable computers of all kinds, particularly in distributed memory environments. Figure 1.4 illustrates how structured comments are used to express the number of processors that a program is to run on and to control the distribution of data. Chapter **??** provides more details on HPF.

```
       program hpf_finite_difference
!HPF$  PROCESSORS pr(4)                ! Running on 4 processors
       real X(100,100), New(100,100)   ! Data arrays
!HPF$  ALIGN New(:,:)  WITH X(:,:)     ! Arrays decomposed in
!HPF$  DISTRIBUTE X(BLOCK,*) ONTO pr   ! one dimension.

       New(2:99,2:99) = (X(1:98, 2:99) + X(3:100, 2:99) +
      $                       X(2:99, 1:98) + X(2:99, 3:100))/4
       diffmax = MAXVAL(ABS(New-X))
```

Figure 1.4: An HPF formulation of the finite difference problem. Notice that only three directives have been added to what is otherwise a pure Fortran 90 program: `PROCESSORS`, `DISTRIBUTE`, and `ALIGN` directives. These directives partition each of the two arrays by row, hence allocating 25 rows to each of 4 processors.

```
   POOMA (or HPC++?) Version of
Some Simple Application
```

Figure 1.5: A POOMA formulation of the XX problem

Analysis. When HPF works well, it is a wonderful tool: complex parallel algorithms can be expressed succinctly as Fortran 90 code. However, the class of algorithms that can be expressed effectively in HPF remains relatively small (although it continues to grow), and HPF compilers are not available for all computers. Hence, HPF remains a niche technology for now at least.

POOMA and HPC++

An alternative approach to the implementation of implicit data parallelism is to use libraries that encapsulate data parallel operations. This is essentially the approach taken in POOMA and HPC++.

> *More details are needed here. Standard numerical libraries (e.g., PETSc) could be discussed here also, although I think it is better to focus on it as a reuse technology.*

CORBA and Java

> *Details to be added.*

```
   CORBA Version of
Some Simple Application
```

Figure 1.6: A CORBA formulation of the XX problem

Hybrids

A variety of hybrids approaches are possible and in some cases are proving effective and popular. For example, it is increasingly common to see applications developed as a distributed memory (MPI) framework with shared memory parallelism (e.g., OpenMP) used within each "process." The primary motivation is a desire to write programs whose structure mirrors that of contemporary parallel computers consisting of multiple shared memory computers connected via a network. The technique can have advantages: for example, a multidimensional problem can be decomposed across processes in one dimension and within a process in a second.

Other hybrids that have been discussed in a research context include MPI and P-Threads, MPI and HPF, CORBA and HPF.

1.2.3 Summary

In the preceding discussion of parallel programming models and technologies we have made a number of points concerning the pros and cons of different approaches. Table 1.2 brings these various issues together in the form of a set of rules for selecting parallel programming models.

Perhaps we can create some sort of decision tree?

1.3 Achieving Correct and Efficient Execution

The problem of achieving *correct* and *efficient* parallel programs is made difficult by the issues noted in the introduction to this chapter: nondeterminism, concurrency, and complex parallel computer architectures. These problems can be overcome by a combination of good programming practice and appropriate tools. Tools such as debuggers, profilers, and performance analyzers are discussed in Chapter **??**; we talk here about two issues of programming practice, namely dealing with nondeterminism and performance modeling.

Table 1.2: Decision rules for selecting parallel programming technologies

Use ...	If:
Compilers	goal is to extract moderate [O(4–10)] parallelism from existing code target platform has a good parallelizing compiler portability is not a major concern
OpenMP	goal is to extract moderate [O(10)] parallelism from existing code good quality OpenMP exists for target platform portability is not a major concern
MPI	you want to use MPI libraries scalability is important portability is important
HPF	writing in F90 program amenable to expression in array syntax good quality HPF is available on target platform
CORBA, RMI	program has task-parallel formulation interested in running in network-based system performance is not critical
Threads	scalability is not important program involves fine-grained operations on shared data program has significant load imbalances OpenMP is not available or suitable

1.3.1 Dealing with Nondeterminism

A nondeterministic computation is one in which the result computed depends on the order in which two or more unsynchronized threads of control happen to execute. Nondeterministic interactions can sometimes be desirable: for example, they can allow us to select the "first" solution computed by a set of worker processes that are executing subtasks of unknown size. However, the presence of nondeterminism also greatly complicates the task of verifying program correctness as, in principle, we need to trace every possible program execution before we can ensure that the program is correct. And in practice it can be difficult both to enumerate the set of possible executions and to reproduce a particular behavior. Hence, nondeterminism is to be avoided whenever possible. The following general techniques can be used to achieve this goal:

- When possible, use a parallel programming technology that does not permit race conditions to occur: e.g., HPF or OpenMP.
- If using a parallel programming technology that permits race conditions, adopt defensive programming practices to avoid unwanted nondeterminism. For example, in MPI, ensure that every "receive" call can match exactly one "send." Avoid the use of P-threads.
- When nondeterminism is required, encapsulate it within objects with well-defined semantics. For example, in a manager-worker structure, the man-

Table 1.3: Major parallel algorithms discussed in this book

Type	Description	Chapter	Page #
...			

ager may invoke a function "get next solution"; all nondeterminism is then encapsulated within this function.

1.3.2 Performance Modeling

In Chapter **??**, tools are described for measuring and analyzing the performance of a parallel program. In principle, a good performance tool should be able to relate observed performance to the constructs of whatever parallel programming technology was used to write the original program. It may also seek to suggest changes to the program that can improve performance. Tools available today do not typically achieve this ideal but they can provide useful information.

An important adjunct to any performance tool is the use of *analytic performance models* as a means of predicting likely performance and of explaining observed performance. As discussed for example in **??**, a good performance model relates parallel program performance (e.g., execution time) to key properties of the program and its target execution environment: for example, problem size, processor speed, and communication costs. Such a model can then be used for qualitative analysis of scalability. If the model is sufficiently accurate (and especially if it is calibrated with experimental data) it can also be used to explain observed performance.

Present a performance model for the example program presented earlier and use an example to show how this can be used to study scalability etc.

1.4 Reusing Parallel Algorithms and Code

The ability to reuse existing algorithms and code is critical to programmer productivity: without it, no programmer can build on prior experience and every programming project must start from scratch. Effective reuse requires both *cataloging* so that programmers can locate algorithms and techniques that meet their needs and *reuse technologies* that allow these algorithms and techniques to be encapsulated in a reusable fashion—whether as design patterns, functions, libraries, components, objects, or whatever.

This book is not intended to serve as a comprehensive catalog of parallel algorithms. Nevertheless, the various application chapters of Part I and the more detailed technology chapters of Part II do collectively present a broad spectrum of algorithms. We provide in Table 1.3 a reasonably complete listing and categorization of these algorithms. For more detailed discussions of parallel algorithm design see the excellent books by XX **??**, YY [], and ZZ [**?**].

Table 1.3 should be completed once the manuscript is further along.

The techologies and techniques used to achieve reuse are discussed in several chapters. We provide here a brief review of three major approaches.

Templates: Design Patterns for Parallel Software

In sequential programming, the concept of a *design pattern* has emerged as an approach to cataloging and communicating basic programming techniques [**?**]. For example, divide and conquer is a design pattern with relevance to a variety of problems. A specification of this pattern might specify the problem-independent structure and note where problem-specific logic must be supplied. This specification does not provide any executable code but provides a basic structure that can guide a programmer in developing an implementation.

The design pattern concept has considerable relevance to parallel programming as in practice there are only a fairly small number of basic parallel algorithm techniques. Here are three examples:

- Manager/worker: summary to be provided.
- Reduction/broadcast: summary to be provided.
- Domain decomposition: summary to be provided.

The design pattern concept turns up at various points in this book but is discussed in particular within Chapter **??**, where the concept of *templates* is introduced. A template is ... (need details).

Communicators and Data Structure Neutrality

The development of truely reusable parallel libraries is difficult (outside the somewhat constrained world of languages such as OpenMP and HPF) because of additional complexities associated with concurrency and data distribution:

- An unfortunate consequence of *concurrency* is that two processes or functions that execute correctly in isolation may not execute correctly when composed, because of race conditions.
- *Data distribution* issues can lead to both correctness and performance problems. If a function expects data to be distributed in one fashion and receives it in another, then either the function will execute incorrectly (in the worst case) or an expensive redistribution operation may be required.

A consequence of these complexities is that until recently there were relatively few examples of successful reusable parallel libraries. Those libraries that did exist (e.g., Scalapack) could only deal with a small number of data distributions and required that these data distributions be specified via cumbersome argument lists

Two recent advances have led to a new generation of libraries that can be composed and reused relatively easily, thanks to two techniques:

- MPI's *communicators* mechanism allows the programmer to encapsulate communications that are "internal" to a function, hence avoiding race conditions that might occur if communications intended for one function are intercepted by another. This mechanism makes it easier to construct components so that interactions occur only via well-defined interfaces.

- Improved software engineering techniques allow data distribution issues to be separated from other aspects of function logic. What are sometimes called *data-structure neutral* libraries allow an application to invoke an operation on a parallel data structure without regard to how the data structure is distributed: the distribution should impact performance but not correctness [**?**].

Contemporary examples of libraries that incorporate these two techniques are the PETSc collection of numerical solvers (see Chapter **??**, XX [], XX [], XX [], and XX [].

Common Component Architecture

> *Conclude with a discussion of the Common Component Architecture (CCA). See HPDC paper.*

1.5 Future Directions

We conclude this chapter with a discussion of four areas in which significant progress is required—and, we believe, will occur—in parallel software concepts and technologies.

> *The next item could be restructured as "higher-level languages" and "component architectures". "Better compilers" is not very exciting as it stands.*

Ease of use. One major goal for research and development in parallel computing must necessarily be to reduce the cost of writing and executing parallel programs, particularly for shared-memory multiprocessor systems.

[paragraph on how advances in compiler technology will help by automating extraction of parallelism from sequential programs.]

A second approach that appears to have considerable promise is to exploit parallelism within programs written in high-level languages such as Matlab, Mathematica, and Excel. [need a few comments about what this involves, references to relevant papers]

A third, related approach is PSEs... [this may be covered above]

Clusters and DSM. While shared-memory multiprocessors are becoming increasingly common, another parallel computing technology is also seeing widespread use, namely clusters constructed from PC nodes connected with commodity networks. Such clusters can be extremely cheap when compared with multiprocessors, but do not offer the same integrated operating system services or convenient shared memory programming model. Heterogeneity is another potential obstacle. However, numerous research and development activities are

working to overcome these problems.

At the operating system level, numerous activities based around MPI-IO, etc. Also work such as Fast Messages, etc., and Virtual Interface Architecture (VIA) focused on reducing communication costs to something more like MPPs.

Clusters today are almost invariably programmed with MPI. Yet experience with multiprocessors shows that shared memory parallelism can be more convenient for applications that involve irregular data structures and data access patterns. Hence, various groups are working to develop distributed shared memory (DSM) systems that will allow a cluster ... [material on DSM]

Grids. Emerging "Computational Grid" infrastructures support the coordinated use of network-connected computers, storage systems, and other resources, allowing them to be used as an integrated computational resource [1].

Grid concepts and technologies have significant implications for the practice of parallel computing. For example, while traditionally parallel computers have been used as "batch" engines for long-running, non-interactive jobs, in Grid environments a parallel computer may need to interact frequently with other systems, whether to acquire instrument data, enable interactive control, or access remote storage systems. These new modes of use are likely to require new runtime system and resource management techniques.

Grid infrastructures can also be used to create what might be termed "generalized clusters," enabling the dynamic discovery and assembly of collections of resources that can be used to solve a computational problem. Because so many computational resources are underutilized, this mode of use has the potential to deliver order-of-magnitude increases in available computation. However, the heterogeneous and dynamic nature of such generalized clusters introduces significant challenges for algorithms and software technologies.

Ultra-scale computers. The final area of future concern that we discuss relates to the software technologies required for tomorrow's extremely large-scale parallel computers—those capable of 10^{15} operations per second or more.

A variety of very different architectures have been proposed for such computers, ranging from scaled-up versions of today's commodity-based systems to systems based on processor-in-memory components and/or superconducting logic [3]. These different systems have in common a need to be able to exploit large amounts of parallelism—10^3 times more than today's largest computers—and to deal with deep memory hierarchies in which memory may be a factor of 10^3 further away (in terms of processor clock cycles) than in today's systems.

These scaling issues, which derive from trends in processor and memory technology, pose major challenges for parallel software technologies at every level.

Further Reading [Do we want this?]

An article by Skillicorn and Talia [2] provides an excellent survey of parallel programming paradigms and languages.

Kennedy compiler book?

Reed parallel I/O book?

MPI-2 book?

The book *The Grid: Blueprint for a Future Computing Infrastructure* provides a comprehensive review of the technologies that underly emerging Grid infrastructures and applications.

The book *Topics in Ultrascale Computing* reviews the hardware and software challenges that must be overcome to build the next generation of high-performance computers, and surveys the state of the art in relevant technologies.

Bibliography

[1] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[2] D. Skillicorn and D. Talia. Models and languages for parallel computation. *Computing Surveys*, 30(2):123–169, 1998.

[3] T. Sterling et al., editors. *Topics in Ultrascale Computing*. 2000.