Chapter 1

Parallel Programming Models and Paradigms

Luís Moura e Silva[†] and Rajkumar Buyya[‡]

[†] Departamento de Engenharia Informática
Universidade de Coimbra, Polo II
3030 Coimbra - Portugal

[‡] School of Computer Science and Software Engineering
Monash University
Melbourne, Australia

Email: *luis@dei.uc.pt, rajkumar@ieee.org*

1.1 Introduction

In the 1980s it was believed computer performance was best improved by creating faster and more efficient processors. This idea was challenged by parallel processing, which in essence means linking together two or more computers to jointly solve a computational problem. Since the early 1990s there has been an increasing trend to move away from expensive and specialized proprietary parallel supercomputers (vector-supercomputers and massively parallel processors) towards networks of computers (PCs/Workstations/SMPs). Among the driving forces that have enabled this transition has been the rapid improvement in the availability of commodity high-performance components for PCs/workstations and networks. These technologies are making a network/cluster of computers an appealing vehicle for cost-effective parallel processing and this is consequently leading to low-cost commodity super-computing.

Scalable computing clusters, ranging from a cluster of (homogeneous or heterogeneous) PCs or workstations, to SMPs, are rapidly becoming the standard platforms for high-performance and large-scale computing. The main attractiveness of such systems is that they are built using affordable, low-cost, commodity hardware (such as Pentium PCs), fast LAN such as Myrinet, and standard software components such as UNIX, MPI, and PVM parallel programming environments. These systems are scalable, i.e., they can be tuned to available budget and computational needs and allow efficient execution of both demanding sequential and parallel applications. Some examples of such systems are Berkeley NOW, HPVM, Beowulf, Solaris-MC, which have been discussed in Volume 1 of this book [8].

Clusters use intelligent mechanisms for dynamic and network-wide resource sharing, which respond to resource requirements and availability. These mechanisms support scalability of cluster performance and allow a flexible use of workstations, since the cluster or network-wide available resources are expected to be larger than the available resources at any one node/workstation of the cluster. These intelligent mechanisms also allow clusters to support multiuser, time-sharing parallel execution environments, where it is necessary to share resources and at the same time distribute the workload dynamically to utilize the global resources efficiently [4].

The idea of exploiting this significant computational capability available in networks of workstations (NOWs) has gained an enthusiastic acceptance within the high-performance computing community, and the current tendency favors this sort of commodity supercomputing. This is mainly motivated by the fact that most of the scientific community has the desire to minimize economic risk and rely on consumer based off-the-shelf technology. Cluster computing has been recognized as the wave of the future to solve large scientific and commercial problems.

We have presented some of the main motivations for the widespread use of clusters in high-performance parallel computing. In the next section, we discuss a generic architecture of a cluster computer and the rest of the chapter focuses on levels of parallelism, programming environments or models, possible strategies for writing parallel programs, and the two main approaches to parallelism (implicit and explicit). Within these two approaches, we briefly summarize the whole spectrum of choices to exploit parallel processing: through the use of parallelizing compilers, parallel languages, message-passing libraries, distributed shared memory, object-oriented programming, and programming skeletons. However, the main focus of the chapter is about the identification and description of the main parallel programming paradigms that are found in existing applications. At the end of the chapter, we present some examples of parallel libraries, tools, and environments that provide higher-level support for parallel programming through the use of skeletons or templates. This approach presents some interesting advantages, for example, the reuse of code, higher flexibility, and the increased productivity of the parallel program developer.

1.2 A Cluster Computer and its Architecture

A cluster is a type of parallel or distributed processing system, which consists of a collection of interconnected stand-alone computers working together as a single, integrated computing resource.

A computer node can be a single or multiprocessor system (PCs, workstations, or SMPs) with memory, I/O facilities, and an operating system. A cluster generally refers to two or more computers (nodes) connected together. The nodes can exist in a single cabinet or be physically separated and connected via a LAN. An inter-connected (LAN-based) cluster of computers can appear as a single system to users and applications. Such a system can provide a cost-effective way to gain features and benefits (fast and reliable services) that have historically been found only on more expensive proprietary shared memory systems. The typical architecture of a cluster is shown in Figure 1.1.

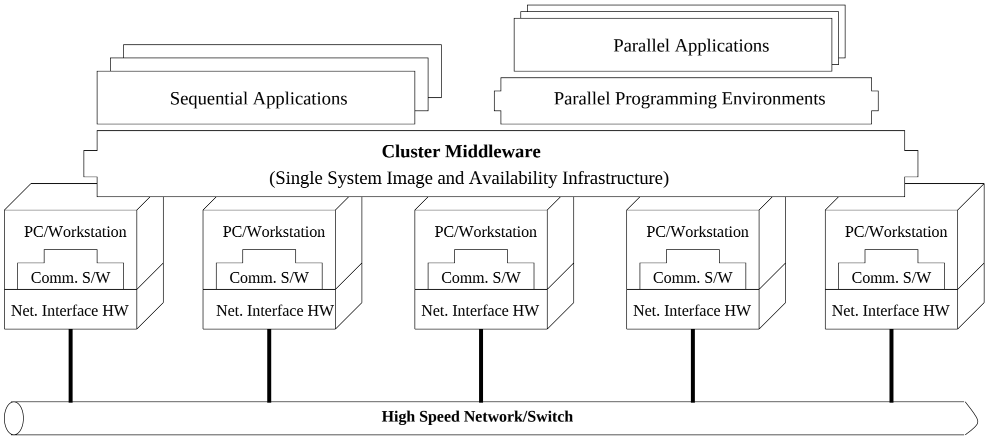

Figure 1.1 Cluster computer architecture.

The following are some prominent components of cluster computers:

- Multiple High Performance Computers (PCs, Workstations, or SMPs)
- State-of-the-art Operating Systems (Layered or Micro-kernel based)
- High Performance Networks/Switches (such as Gigabit Ethernet and Myrinet)
- Network Interface Cards (NICs)
- Fast Communication Protocols and Services (such as Active and Fast Messages)
- Cluster Middleware (Single System Image (SSI) and System Availability Infrastructure)
  - Hardware (such as Digital (DEC) Memory Channel, hardware DSM, and SMP techniques)
  - Operating System Kernel or Gluing Layer (such as Solaris MC and GLUnix)

  - Applications and Subsystems
    * Applications (such as system management tools and electronic forms)
    * Run-time Systems (such as software DSM and parallel file-system)
    * Resource Management and Scheduling software (such as LSF (Load Sharing Facility) and CODINE (COmputing in DIstributed Networked Environments))

- Parallel Programming Environments and Tools (such as compilers, PVM (Parallel Virtual Machine), and MPI (Message Passing Interface))

- Applications
  - Sequential
  - Parallel or Distributed

The network interface hardware acts as a communication processor and is responsible for transmitting and receiving packets of data between cluster nodes via a network/switch.

Communication software offers a means of fast and reliable data communication among cluster nodes and to the outside world. Often, clusters with a special network/switch like Myrinet use communication protocols such as active messages for fast communication among its nodes. They potentially bypass the operating system and thus remove the critical communication overheads providing direct user-level access to the network interface.

The cluster nodes can work collectively as an integrated computing resource, or they can operate as individual computers. The cluster middleware is responsible for offering an illusion of a unified system image (single system image) and availability out of a collection on independent but interconnected computers.

Programming environments can offer portable, efficient, and easy-to-use tools for development of applications. They include message passing libraries, debuggers, and profilers. It should not be forgotten that clusters could be used for the execution of sequential or parallel applications.

1.3 Parallel Applications and Their Development

The class of applications that a cluster can typically cope with would be considered demanding sequential applications and grand challenge/supercomputing applications. Grand Challenge Applications (GCAs) are fundamental problems in science and engineering with broad economic and scientific impact [18]. They are generally considered intractable without the use of state-of-the-art parallel computers. The scale of their resource requirements, such as processing time, memory, and communication needs distinguishes GCAs.

A typical example of a grand challenge problem is the simulation of some phenomena that cannot be measured through experiments. GCAs include massive crystallographic and microtomographic structural problems, protein dynamics and biocatalysis, relativistic quantum chemistry of actinides, virtual materials design and processing, global climate modeling, and discrete event simulation.

Although the technology of clusters is currently being deployed, the development of parallel applications is really a complex task. First of all, it is largely dependent on the availability of adequate software tools and environments. Second, parallel software developers must contend with problems not encountered during sequential programming, namely: non-determinism, communication, synchronization, data partitioning and distribution, load-balancing, fault-tolerance, heterogeneity, shared or distributed memory, deadlocks, and race conditions. All these issues present some new important challenges.

Currently, only some specialized programmers have the knowledge to use parallel and distributed systems for executing production codes. This programming technology is still somehow distant from the average sequential programmer, who does not feel very enthusiastic about moving into a different programming style with increased difficulties, though they are aware of the potential performance gains. Parallel computing can only be widely successful if parallel software is able to accomplish some expectations of the users, such as:

- provide architecture/processor type transparency;
- provide network/communication transparency;
- be easy-to-use and reliable;
- provide support for fault-tolerance;
- accommodate heterogeneity;
- assure portability;
- provide support for traditional high-level languages;
- be capable of delivering increased performance;
- and finally, the *holy-grail* is to provide parallelism transparency.

This last expectation is still at least one decade away, but most of the others can be achieved today. The internal details of the underlying architecture should be hidden from the user and the programming environment should provide high-level support for parallelism. Otherwise, if the programming interface is difficult to use, it makes the writing of parallel applications highly unproductive and painful for most programmers. There are basically two main approaches for parallel programming:

1. the first one is based on *implicit parallelism*. This approach has been followed by parallel languages and parallelizing compilers. The user does not specify, and thus cannot control, the scheduling of calculations and/or the placement of data;

2. the second one relies on *explicit parallelism*. In this approach, the programmer is responsible for most of the parallelization effort such as task decomposition, mapping tasks to processors, and the communication structure. This

approach is based on the assumption that the user is often the best judge of how parallelism can be exploited for a particular application.

It is also observed that the use of explicit parallelism will obtain a better efficiency than parallel languages or compilers that use implicit parallelism.

1.3.1 Strategies for Developing Parallel Applications

Undoubtedly, the main software issue is to decide between either porting existing sequential applications or developing new parallel applications from scratch. There are three strategies for creating parallel applications, as shown in Figure 1.2.

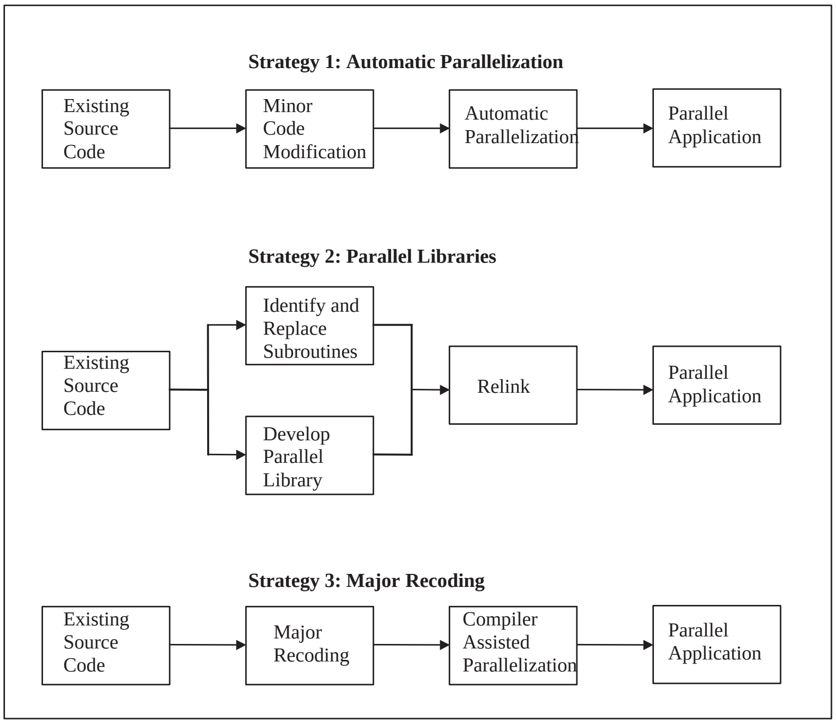

Figure 1.2 Porting strategies for parallel applications.

The first strategy is based on automatic parallelization, the second is based on the use of parallel libraries, while the third strategy—major recoding—resembles *from-scratch* application development.

The goal of automatic parallelization is to relieve the programmer from the parallelizing tasks. Such a compiler would accept *dusty-deck* codes and produce efficient parallel object code without any (or, at least, very little) additional work by the programmer. However, this is still very hard to achieve and is well beyond the reach of current compiler technology.

Another possible approach for porting parallel code is the use of parallel libraries. This approach has been more successful than the previous one. The basic idea is to encapsulate some of the parallel code that is common to several applications into a parallel library that can be implemented in a very efficient way. Such a library can then be reused by several codes. Parallel libraries can take two forms:

1. they encapsulate the control structure of a class of applications;

2. they provide a parallel implementation of some mathematical routines that are heavily used in the kernel of some production codes.

The third strategy, which involves writing a parallel application from the very beginning, gives more freedom to the programmer who can choose the language and the programming model. However, it may make the task very difficult since little of the code can be reused. Compiler assistance techniques can be of great help, although with a limited applicability. Usually the tasks that can be effectively provided by a compiler are data distribution and placement.

1.4 Code Granularity and Levels of Parallelism

In modern computers, parallelism appears at various levels both in hardware and software: signal, circuit, component, and system levels. That is, at the very lowest level, signals travel in parallel along parallel data paths. At a slightly higher level, multiple functional units operate in parallel for faster performance, popularly known as *instruction level parallelism*. For instance, a PC processor such as Pentium Pro has the capability to process three instructions simultaneously. Many computers overlap CPU and I/O activities; for instance, a disk access for one user while executing instruction of another user. Some computers use a memory interleaving technique – several banks of memory can be accessed in parallel for faster accesses to memory. At a still higher level, SMP systems have multiple CPUs that work in parallel. At an even higher level of parallelism, one can connect several computers together and make them work as a single machine, popularly known as cluster computing.

The first two levels (signal and circuit level) of parallelism is performed by a hardware implicitly technique called hardware parallelism. The remaining two levels (component and system) of parallelism is mostly expressed implicitly/explicitly by using various software techniques, popularly known as software parallelism.

Levels of parallelism can also be based on the lumps of code (grain size) that can be a potential candidate for parallelism. Table 1.1 lists categories of code granularity for parallelism. All approaches of creating parallelism based on code granularity have a common goal to boost processor efficiency by hiding latency of a lengthy operation such as a memory/disk access. To conceal latency, there must be another activity ready to run whenever a lengthy operation occurs. The idea is to execute concurrently two or more single-threaded applications, such as compiling, text formatting, database searching, and device simulation, or parallelized applications having multiple tasks simultaneously.

Table 1.1 Code Granularity and Parallelism

Grain Size	Code Item	Parallelised by
Very Fine	Instruction	Processor
Fine	Loop/Instruction block	Compiler
Medium	Standard One Page Function	Programmer
Large	Program-Separate heavyweight process	Programmer

Parallelism in an application can be detected at several levels. They are:

- very-fine grain (multiple instruction issue)
- fine-grain (data-level)
- medium-grain (or control-level)
- large-grain (or task-level)

The different levels of parallelism are depicted in Figure 1.3. Among the four levels of parallelism, the first two levels are supported transparently either by the hardware or parallelizing compilers. The programmer mostly handles the last two levels of parallelism. The three important models used in developing applications are shared-memory model, distributed memory model (message passing model), and distributed-shared memory model. These models are discussed in Chapter 2.

1.5 Parallel Programming Models and Tools

This section presents a brief overview on the area of parallel programming and describes the main approaches and models, including parallelizing compilers, parallel languages, message-passing, virtual shared memory, object-oriented programming, and programming skeletons.

1.5.1 Parallelizing Compilers

There has been some research in parallelizing compilers and parallel languages but their functionality is still very limited. Parallelizing compilers are still limited to applications that exhibit *regular* parallelism, such as computations in loops. Parallelizing/vectorizing compilers have proven to be relatively successful for some applications on shared-memory multiprocessors and vector processors with shared memory, but are largely unproven for distributed-memory machines. The difficulties are due to the non uniform access time of memory in the latter systems. The currently existing compiler technology for automatic parallelization is thus limited in scope and only rarely provides adequate speedup.

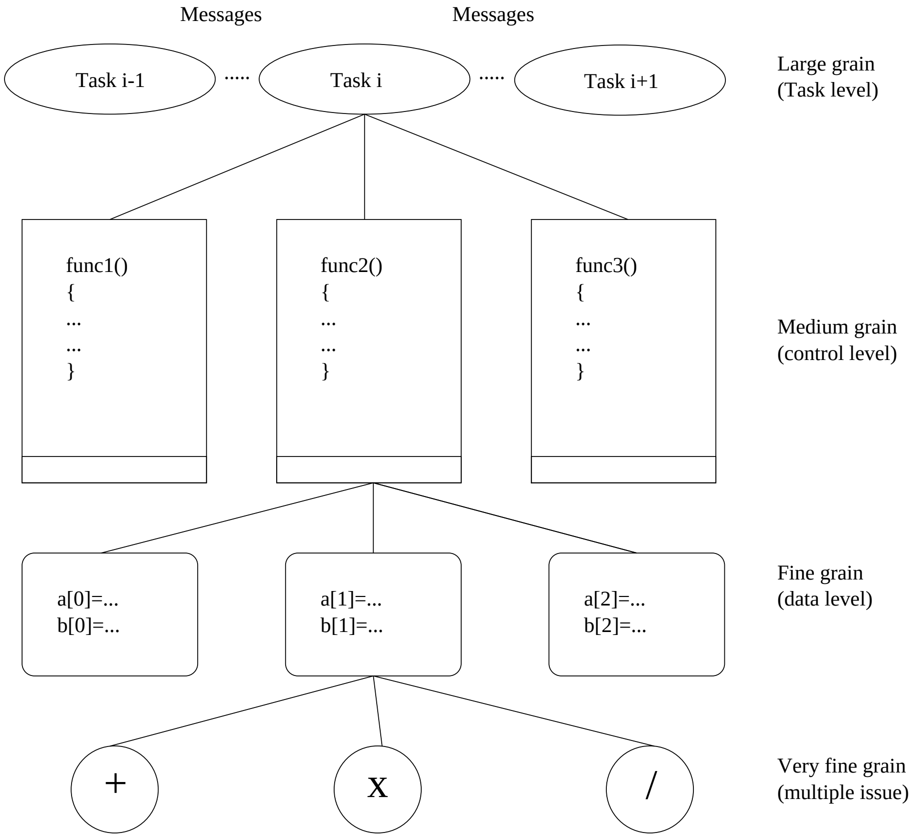

Figure 1.3 Detecting parallelism.

1.5.2 Parallel Languages

Some parallel languages, like SISAL [11] and PCN [13] have found little favor with application programmers. This is because users are not willing to learn a completely new language for parallel programming. They really would prefer to use their traditional high-level languages (like C and Fortran) and try to recycle their already available sequential software. For these programmers, the extensions to existing languages or run-time libraries are a viable alternative.

1.5.3 High Performance Fortran

The High Performance Fortran (HPF) initiative [20] seems to be a promising solution to solve the *dusty-deck* problem of Fortran codes. However, it only supports applications that follow the SPMD paradigm and have a very regular structure. Other applications that are missing these characteristics and present a more asynchronous structure are not as successful with the current versions of HPF. Current and future research will address these issuess.

1.5.4 Message Passing

Message passing libraries allow efficient parallel programs to be written for distributed memory systems. These libraries provide routines to initiate and configure the messaging environment as well as sending and receiving packets of data. Currently, the two most popular high-level message-passing systems for scientific and engineering application are the PVM (Parallel Virtual Machine) from Oak Ridge National Laboratory and MPI (Message Passing Interface) defined by the MPI Forum.

Currently, there are several implementations of MPI, including versions for networks of workstations, clusters of personal computers, distributed-memory multiprocessors, and shared-memory machines. Almost every hardware vendor is supporting MPI. This gives the user a comfortable feeling since an MPI program can be executed on almost all of the existing computing platforms without the need to rewrite the program from scratch. The goal of portability, architecture, and network transparency has been achieved with these low-level communication libraries like MPI and PVM. Both communication libraries provide an interface for C and Fortran, and additional support of graphical tools.

However, these message-passing systems are still stigmatized as low-level because most tasks of the parallelization are still left to the application programmer. When writing parallel applications using message passing, the programmer still has to develop a significant amount of software to manage some of the tasks of the parallelization, such as: the communication and synchronization between processes, data partitioning and distribution, mapping of processes onto processors, and input/output of data structures. If the application programmer has no special support for these tasks, it then becomes difficult to widely exploit parallel computing. The easy-to-use goal is not accomplished with a bare message-passing system, and hence requires additional support.

Other ways to provide alternate-programming models are based on Virtual Shared Memory (VSM) and parallel object-oriented programming. Another way is to provide a set of programming skeletons in the form of run-time libraries that already support some of the tasks of parallelization and can be implemented on top of portable message-passing systems like PVM or MPI.

1.5.5 Virtual Shared Memory

VSM implements a shared-memory programming model in a distributed-memory environment. Linda is an example of this style of programming [1]. It is based on the notion of generative communication model and on a virtual shared associative memory, called tuple space, that is accessible to all the processes by using *in* and *out* operations.

Distributed Shared Memory (DSM) is the extension of the well-accepted shared-memory programming model on systems without physically shared memory [21]. The shared data space is flat and accessed through normal read and write operations. In contrast to message passing, in a DSM system a process that wants to fetch some data value does not need to know its location; the system will find and fetch it automatically. In most of the DSM systems, shared data may be replicated to enhance the parallelism and the efficiency of the applications.

While scalable parallel machines are mostly based on distributed memory, many users may find it easier to write parallel programs using a shared-memory programming model. This makes DSM a very promising model, provided it can be implemented efficiently.

1.5.6 Parallel Object-Oriented Programming

The idea behind parallel object-oriented programming is to provide suitable abstractions and software engineering methods for structured application design. As in the traditional object model, objects are defined as abstract data types, which encapsulate their internal state through well-defined interfaces and thus represent passive data containers. If we treat this model as a collection of shared objects, we can find an interesting resemblance with the shared data model.

The object-oriented programming model is by now well established as the state-of-the-art software engineering methodology for sequential programming, and recent developments are also emerging to establish object-orientation in the area of parallel programming. The current lack of acceptance of this model among the scientific community can be explained by the fact that computational scientists still prefer to write their programs using traditional languages like Fortran. This is the main difficulty that has been faced by the object-oriented environments, though it is considered as a promising technique for parallel programming. Some interesting object-oriented environments such as CC++ and Mentat have been presented in the literature [24].

1.5.7 Programming Skeletons

Another alternative to the use of message-passing is to provide a set of high-level abstractions which provides support for the mostly used parallel paradigms. A programming paradigm is a class of algorithms that solve different problems but have the same control structure [19]. Programming paradigms usually encapsulate information about useful data and communication patterns, and an interesting idea is to provide such abstractions in the form of programming templates or skeletons. A skeleton corresponds to the instantiation of a specific parallel programming paradigm, and it encapsulates the control and communication primitives of the application into a single abstraction.

After the recognition of parallelizable parts and an identification of the appropriate algorithm, a lot of developing time is wasted on programming routines closely related to the paradigm and not the application itself. With the aid of a good set of efficiently programmed interaction routines and skeletons, the development time can be reduced significantly.

The skeleton hides from the user the specific details of the implementation and allows the user to specify the computation in terms of an interface tailored to the paradigm. This leads to a style of skeleton oriented programming (SOP) which has been identified as a very promising solution for parallel computing [6].

Skeletons are more general programming methods since they can be implemented on top of message-passing, object-oriented, shared-memory or distributed memory systems, and provide increased support for parallel programming.

To summarize, there are basically two ways of looking at an explicit parallel programming system. In the first one, the system provides some primitives to be used by the programmer. The structuring and the implementation of most of the parallel control and communication is the responsibility of the programmer. The alternative is to provide some enhanced support for those control structures that are common to a parallel programming paradigm. The main task of the programmer would be to provide those few routines unique to the application, such as computation and data generation. Numerous parallel programming environments are available, and many of them do attempt to exploit the characteristics of parallel paradigms.

1.6 Methodical Design of Parallel Algorithms

There is no simple recipe for designing parallel algorithms. However, it can benefit from a methodological approach that maximizes the range of options, that provides mechanisms for evaluating alternatives, and that reduces the cost of backtracking from wrong choices. The design methodology allows the programmer to focus on machine-independent issues such as concurrency in the early stage of design process, and machine-specific aspects of design are delayed until late in the design process. As suggested by Ian Foster, this methodology organizes the design process into four distinct stages [12]:

- partitioning
- communication
- agglomeration
- mapping

The first two stages seek to develop concurrent and scalable algorithms, and the last two stages focus on locality and performance-related issues as summarized below:

1.6.1 Partitioning

It refers to decomposing of the computational activities and the data on which it operates into several small tasks. The decomposition of the data associated with a problem is known as *domain/data decomposition*, and the decomposition of computation into disjoint tasks is known as *functional decomposition*. Various paradigms underlying the partitioning process are discussed in the next section.

1.6.2 Communication

It focuses on the flow of information and coordination among the tasks that are created during the partitioning stage. The nature of the problem and the decomposition method determine the communication pattern among these cooperative tasks of a parallel program. The four popular communication patterns commonly used in parallel programs are: local/global, structured/unstructured, static/dynamic, and synchronous/asynchronous.

1.6.3 Agglomeration

In this stage, the tasks and communication structure defined in the first two stages are evaluated in terms of performance requirements and implementation costs. If required, tasks are grouped into larger tasks to improve performance or to reduce development costs. Also, individual communications may be bundled into a *super* communication. This will help in reducing communication costs by increasing computation and communication granularity, gaining flexibility in terms of scalability and mapping decisions, and reducing software-engineering costs.

1.6.4 Mapping

It is concerned with assigning each task to a processor such that it maximizes utilization of system resources (such as CPU) while minimizing the communication costs. Mapping decisions can be taken statically (at compile-time/before program execution) or dynamically at runtime by load-balancing methods as discussed in Volume 1 of this book [8] and Chapter 17.

Several grand challenging applications have been built using the above methodology (refer to Part III, Algorithms and Applications, for further details on the development of real-life applications using the above methodology).

1.7 Parallel Programming Paradigms

It has been widely recognized that parallel applications can be classified into some well defined programming paradigms. A few programming paradigms are used repeatedly to develop many parallel programs. Each paradigm is a class of algorithms that have the same control structure [19].

Experience to date suggests that there are a relatively small number of paradigms underlying most parallel programs [7]. The choice of paradigm is determined by the available parallel computing resources and by the type of parallelism inherent in the problem. The computing resources may define the level of granularity that can be efficiently supported on the system. The type of parallelism reflects the structure of either the application or the data and both types may exist in different parts of the same application. Parallelism arising from the structure of the application is named as functional parallelism. In this case, different parts of the program can perform different tasks in a concurrent and cooperative manner. But parallelism may also be found in the structure of the data. This type of parallelism allows the execution of parallel processes with identical operation but on different parts of the data.

1.7.1 Choice of Paradigms

Most of the typical distributed computing applications are based on the very popular client/server paradigm. In this paradigm, the processes usually communicate through Remote Procedure Calls (RPCs), but there is no inherent parallelism in this sort of applications. They are instead used to support distributed services, and thus we do not consider this paradigm in the parallel computing area.

In the world of parallel computing there are several authors which present a paradigm classification. Not all of them propose exactly the same one, but we can create a superset of the paradigms detected in parallel applications.

For instance, in [22], a theoretical classification of parallel programs is presented and broken into three classes of parallelism:

1. processor farms, which are based on replication of independent jobs;
2. geometric decomposition, based on the parallelisation of data structures; and
3. algorithmic parallelism, which results in the use of data flow.

Another classification was presented in [19]. The author studied several parallel applications and identified the following set of paradigms:

1. pipelining and ring-based applications;
2. divide and conquer;
3. master/slave; and
4. cellular automata applications, which are based on data parallelism.

The author of [23] also proposed a very appropriate classification. The problems were divided into a few decomposition techniques, namely:

1. Geometric decomposition: the problem domain is broken up into smaller domains and each process executes the algorithm on each part of it.

2. Iterative decomposition: some applications are based on loop execution where each iteration can be done in an independent way. This approach is implemented through a central queue of runnable tasks, and thus corresponds to the task-farming paradigm.

3. Recursive decomposition: this strategy starts by breaking the original problem into several subproblems and solving these in a parallel way. It clearly corresponds to a divide and conquer approach.

4. Speculative decomposition: some problems can use a speculative decomposition approach: N solution techniques are tried simultaneously, and (N-1) of them are thrown away as soon as the first one returns a plausible answer. In some cases this could result optimistically in a shorter overall execution time.

5. Functional decomposition: the application is broken down into many distinct phases, where each phase executes a different algorithm within the same problem. The most used topology is the process pipelining.

In [15], a somewhat different classification was presented based on the temporal structure of the problems. The applications were thus divided into:

1. synchronous problems, which correspond to regular computations on regular data domains;
2. loosely synchronous problems, that are typified by iterative calculations on geometrically irregular data domains;
3. asynchronous problems, which are characterized by functional parallelism that is irregular in space and time; and
4. embarrassingly parallel applications, which correspond to the independent execution of disconnected components of the same program.

Synchronous and loosely synchronous problems present a somehow different synchronization structure, but both rely on data decomposition techniques. According to an extensive analysis of 84 real applications presented in [14], it was estimated that these two classes of problems dominated scientific and engineering applications being used in 76 percent of the applications. Asynchronous problems, which are for instance represented by event-driven simulations, represented 10 percent of the studied problems. Finally, embarrassingly parallel applications that correspond to the master/slave model, accounted for 14 percent of the applications.

The most systematic definition of paradigms and application templates was presented in [6]. It describes a generic tuple of factors which fully characterizes a parallel algorithm including: process properties (structure, topology and execution), interaction properties, and data properties (partitioning and placement). That classification included most of the paradigms referred so far, albeit described in deeper detail.

To summarize, the following paradigms are popularly used in parallel programming:

- Task-Farming (or Master/Slave)
- Single Program Multiple Data (SPMD)

- Data Pipelining
- Divide and Conquer
- Speculative Parallelism

1.7.2 Task-Farming (or Master/Slave)

The task-farming paradigm consists of two entities: master and multiple slaves. The master is responsible for decomposing the problem into small tasks (and distributes these tasks among a farm of slave processes), as well as for gathering the partial results in order to produce the final result of the computation. The slave processes execute in a very simple cycle: get a message with the task, process the task, and send the result to the master. Usually, the communication takes place only between the master and the slaves.

Task-farming may either use static load-balancing or dynamic load-balancing. In the first case, the distribution of tasks is all performed at the beginning of the computation, which allows the master to participate in the computation after each slave has been allocated a fraction of the work. The allocation of tasks can be done once or in a cyclic way. Figure 1.4 presents a schematic representation of this first approach.

The other way is to use a dynamically load-balanced master/slave paradigm, which can be more suitable when the number of tasks exceeds the number of available processors, or when the number of tasks is unknown at the start of the application, or when the execution times are not predictable, or when we are dealing with unbalanced problems. An important feature of dynamic load-balancing is the ability of the application to adapt itself to changing conditions of the system, not just the load of the processors, but also a possible reconfiguration of the system resources. Due to this characteristic, this paradigm can respond quite well to the failure of some processors, which simplifies the creation of robust applications that are capable of surviving the loss of slaves or even the master.

At an extreme, this paradigm can also enclose some applications that are based on a trivial decomposition approach: the sequential algorithm is executed simultaneously on different processors but with different data inputs. In such applications there are no dependencies between different runs so there is no need for communication or coordination between the processes.

This paradigm can achieve high computational speedups and an interesting degree of scalability. However, for a large number of processors the centralized control of the master process can become a bottleneck to the applications. It is, however, possible to enhance the scalability of the paradigm by extending the single master to a set of masters, each of them controlling a different group of process slaves.

1.7.3 Single-Program Multiple-Data (SPMD)

The SPMD paradigm is the most commonly used paradigm. Each process executes basically the same piece of code but on a different part of the data. This involves

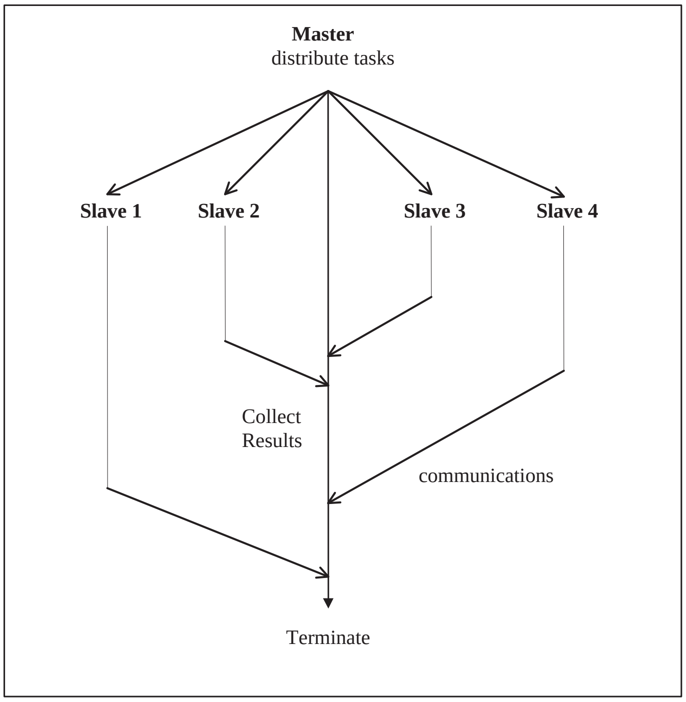

Figure 1.4 A static master/slave structure.

the splitting of application data among the available processors. This type of parallelism is also referred to as geometric parallelism, domain decomposition, or data parallelism. Figure 1.5 presents a schematic representation of this paradigm.

Many physical problems have an underlying regular geometric structure, with spatially limited interactions. This homogeneity allows the data to be distributed uniformly across the processors, where each one will be responsible for a defined spatial area. Processors communicate with neighbouring processors and the communication load will be proportional to the size of the boundary of the element, while the computation load will be proportional to the volume of the element. It may also be required to perform some global synchronization periodically among all the processes. The communication pattern is usually highly structured and extremely predictable. The data may initially be self-generated by each process or may be read from the disk during the initialization stage.

SPMD applications can be very efficient if the data is well distributed by the processes and the system is homogeneous. If the processes present different workloads or capabilities, then the paradigm requires the support of some load-balancing scheme able to adapt the data distribution layout during run-time execution.

This paradigm is highly sensitive to the loss of some process. Usually, the loss of a single process is enough to cause a deadlock in the calculation in which none of the processes can advance beyond a global synchronization point.

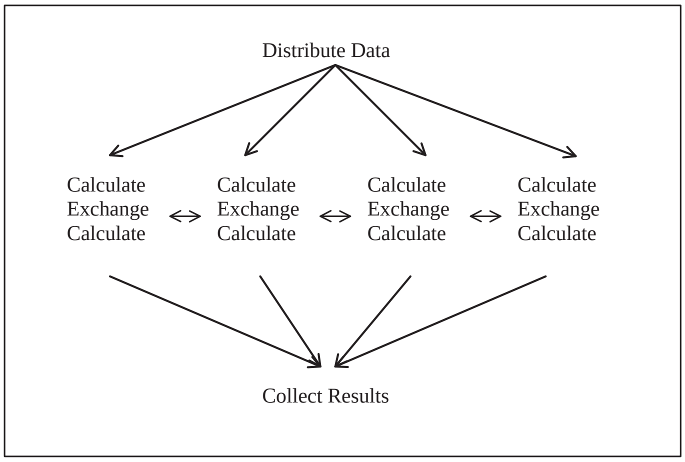

Figure 1.5 Basic structure of a SPMD program.

1.7.4 Data Pipelining

This is a more fine-grained parallelism, which is based on a functional decomposition approach: the tasks of the algorithm, which are capable of concurrent operation, are identified and each processor executes a small part of the total algorithm. The pipeline is one of the simplest and most popular functional decomposition paradigms. Figure 1.6 presents the structure of this model.

Processes are organized in a pipeline – each process corresponds to a stage of the pipeline and is responsible for a particular task. The communication pattern can be very simple since the data flows between the adjacent stages of the pipeline. For this reason, this type of parallelism is also sometimes referred to as data flow parallelism. The communication may be completely asynchronous. The efficiency of this paradigm is directly dependent on the ability to balance the load across the stages of the pipeline. The robustness of this paradigm against reconfigurations of the system can be achieved by providing multiple independent paths across the stages. This paradigm is often used in data reduction or image processing applications.

1.7.5 Divide and Conquer

The divide and conquer approach is well known in sequential algorithm development. A problem is divided up into two or more subproblems. Each of these subproblems is solved independently and their results are combined to give the final result. Often, the smaller problems are just smaller instances of the original

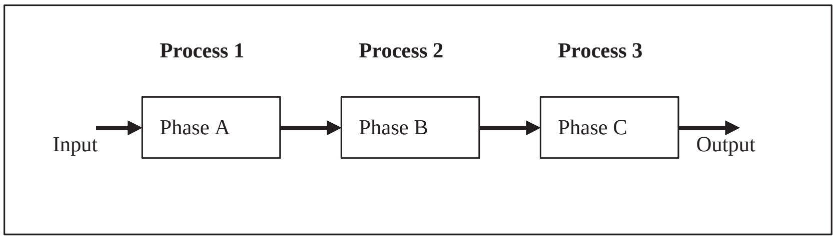

Figure 1.6 Data pipeline structure.

problem, giving rise to a recursive solution. Processing may be required to divide the original problem or to combine the results of the subproblems. In parallel divide and conquer, the subproblems can be solved at the same time, given sufficient parallelism. The splitting and recombining process also makes use of some parallelism, but these operations require some process communication. However, because the subproblems are independent, no communication is necessary between processes working on different subproblems.

We can identify three generic computational operations for divide and conquer: split, compute, and join. The application is organized in a sort of virtual tree: some of the processes create subtasks and have to combine the results of those to produce an aggregate result. The tasks are actually computed by the compute processes at the leaf nodes of the virtual tree. Figure 1.7 presents this execution.

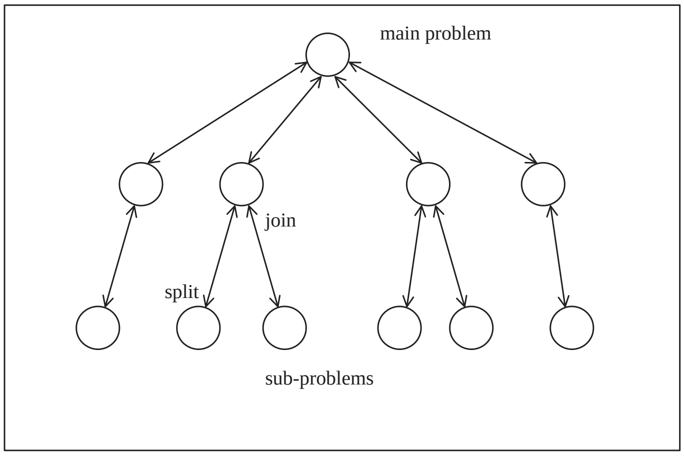

Figure 1.7 Divide and conquer as a virtual tree.

The task-farming paradigm can be seen as a slightly modified, degenerated form of divide and conquer; i.e., where problem decomposition is performed before tasks are submitted, the split and join operations is only done by the master process and all the other processes are only responsible for the computation.

In the divide and conquer model, tasks may be generated during runtime and may be added to a single job queue on the manager processor or distributed through several job queues across the system.

The programming paradigms can be mainly characterized by two factors: decomposition and distribution of the parallelism. For instance, in geometric parallelism both the decomposition and distribution are static. The same happens with the functional decomposition and distribution of data pipelining. In task farming, the work is statically decomposed but dynamically distributed. Finally, in the divide and conquer paradigm both decomposition and distribution are dynamic.

1.7.6 Speculative Parallelism

This paradigm is employed when it is quite difficult to obtain parallelism through any one of the previous paradigms. Some problems have complex data dependencies, which reduces the possibilities of exploiting the parallel execution. In these cases, an appropriate solution is to execute the problem in small parts but use some speculation or optimistic execution to facilitate the parallelism.

In some asynchronous problems, like discrete-event simulation [17], the system will attempt the *look-ahead execution* of related activities in an optimistic assumption that such concurrent executions do not violate the consistency of the problem execution. Sometimes they do, and in such cases it is necessary to rollback to some previous consistent state of the application.

Another use of this paradigm is to employ different algorithms for the same problem; the first one to give the final solution is the one that is chosen.

1.7.7 Hybrid Models

The boundaries between the paradigms can sometimes be fuzzy and, in some applications, there could be the need to mix elements of different paradigms. Hybrid methods that include more than one basic paradigm are usually observed in some large-scale parallel applications. These are situations where it makes sense to mix data and task parallelism simultaneously or in different parts of the same program.

1.8 Programming Skeletons and Templates

The term skeleton has been identified by two important characteristics [10]:

- it provides only an underlying structure that can be hidden from the user;
- it is incomplete and can be parameterized, not just by the number of processors, but also by other factors, such as granularity, topology and data distribution.

Hiding the underlying structure from the user by presenting a simple interface results in programs that are easier to understand and maintain, as well as less prone to error. In particular, the programmer can now focus on the computational task rather than the control and coordination of the parallelism.

Exploiting the observation that parallel applications follow some well-identified structures, much of the parallel software specific to the paradigm can be potentially reusable. Such software can be encapsulated in parallel libraries to promote the reuse of code, reduce the burden on the parallel programmer, and to facilitate the task of recycling existing sequential programs. This guideline was followed by the PUL project [9], the TINA system [7], and the ARNIA package [16].

A project developed at the Edinburgh Parallel Computing Centre [9] involved the writing of a package of parallel utilities (PUL) on top of MPI that gives programming support for the most common programming paradigms as well as parallel input/output. Apart from the libraries for global and parallel I/O, the collection of the PUL utilities includes a library for task-farming applications (PUL-TF), another that supports regular domain decomposition applications (PUL-RD), and another one that can be used to program irregular mesh-based problems (PUL-SM). This set of PUL utilities hides the hard details of the parallel implementation from the application programmer and provides a portable programming interface that can be used on several computing platforms. To ensure programming flexibility, the application can make simultaneous use of different PUL libraries and have direct access to the MPI communication routines.

The ARNIA package [16] includes a library for master/slave applications, another for the domain decomposition paradigm, a special library for distributed computing applications based on the client/server model, and a fourth library that supports a global shared memory emulation. ARNIA allows the combined use of its building libraries for those applications that present mixed paradigms or distinct computational phases.

In [7], a skeleton generator was presented, called TINA, that supports the reusability and portability of parallel program components and provides a complete programming environment.

Another graphical programming environment, named TRACS (see Chapter 7), provides a graphical toolkit to design distributed/parallel applications based on reusable components, such as farms, grids, and pipes.

Porting and rewriting application programs requires a support environment that encourages code reuse, portability among different platforms, and scalability across similar systems of different size. This approach, based on skeletal frameworks, is a viable solution for parallel programming. It can significantly increase programmer productivity because programmers will be able to develop parts of programs simply by filling in the templates. The development of software templates has been increasingly receiving the attention of academic research and is seen as one of the key directions for parallel software.

The most important advantages of this approach for parallel programming are summarized below.

1.8.1 Programmability

A set of ready-to-use solutions for parallelization will considerably increase the productivity of the programmers: the idea is to hide the lower level details of the system, to promote the reuse of code, and relieve the burden of the application programmer. This approach will increase the programmability of the parallel systems since the programmer will have more time to spend in optimizing the application itself, rather than on low-level details of the underlying programming system.

1.8.2 Reusability

Reusability is a hot-topic in software engineering. The provision of skeletons or templates to the application programmer increases the potential for reuse by allowing the same parallel structure to be used in different applications. This avoids the replication of efforts involved in developing and optimizing the code specific to the parallel template. In [3] it was reported that a percentage of code reuse rose from 30 percent up to 90 percent when using skeleton-oriented programming. In the Chameleon system [2], 60 percent of the code was reusable, while in [5] it was reported that an average fraction of 80 percent of the code was reused with the ROPE library.

1.8.3 Portability

Providing portability of the parallel applications is a problem of paramount importance. It allows applications developed on one platform to run on another platform without the need for redevelopment.

1.8.4 Efficiency

There could be some conflicting trade-off between optimal performance and portability/programmability. Both portability and efficiency of parallel programming systems play an important role in the success of parallel computing.

1.9 Conclusions

This chapter presented a brief overview about the motivations for using clusters in parallel computing, presented the main models of execution (parallelizing compilers, message-passing libraries, virtual shared-memory, object-oriented programming), and described the mostly used parallel programming paradigms that can be found in existing applications. At the end of the chapter we have underlined the most important advantages of using programming skeletons and environments for higher-level parallel programming.

In these past years there has been a considerable effort in developing software for exploiting the computational power of parallel, distributed, and cluster-based systems. Many advances have been achieved in parallel software but there is still considerable work to do in the next decade in order to effectively exploit the computational power of cluster for parallel high performance computing.

Acknowledgment

We thank Dan Hyde (Bucknell University, USA) for his comments and suggestions on this chapter.

1.10 Bibliography

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and Friends. *IEEE Computer,* pages 26-34, August 1986.

[2] G.A. Alverson and D. Notkin. Program Structuring for Effective Parallel Portability. *IEEE Transactions on Parallel and Distributed Systems,* vol. 4 (9), pages 1041-1069, September 1993.

[3] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti and M. Vanneschi. Summarising an Experiment in Parallel Programming Language Design. *Lecture Notes in Computer Science,* 919, High-Performance Computing and Networking, HPCN'95, Milano, Italy, pages 7-13, 1995.

[4] A. Barak and O. La'adan. Performance of the MOSIX Parallel System for a Cluster of PC's. In *Proceedings of HPCN - Europe conference*, 1997.

[5] J.C. Browne, T. Lee, and J.Werth. Experimental Evaluation of a Reusability-Oriented Parallel Programming Environment. *IEEE Transactions on Software Engineering,* vol. 16 (2), pages 111-120, February 1990.

[6] H. Burkhart, C.F. Korn, S. Gutzwiller, P. Ohnacker, and S.Waser. em BACS: Basel Algorithm Classification Scheme. *Technical Report 93-03*, Univ. Basel, Switzerland, 1993.

[7] H. Burkhart and S. Gutzwiller. Steps Towards Reusability and Portability in Parallel Programming. In *Programming Environments for Massively Parallel Distributed Systems,* Monte Verita, Switzerland pages 147-157, April 1994.

[8] R. Buyya (editor). *High Performance Cluster Computing: Systems and Architectures.* Volume 1, Prentice Hall PTR, NJ, 1999.

[9] L. Clarke, R. Fletcher, S. Trewin, A. Bruce, G. Smith, and S. Chapple. Reuse, Portability and Parallel Libraries. In *Proceedings of IFIP WG10.3 Working Conference on Programming Environments for Massively Parallel and Distributed Systems,* Monte Verita, Switzerland, April 1994.

[10] M.Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations.* MIT Press, Cambridge, MA, 1989.

[11] J. Feo, D. Cann, and R. Oldehoeft. A Report on the SISAL Language Project. *Journal of Parallel and Distributed Computing,* vol 10, pages 349-366, 1990.

[12] I. Foster. *Designing and Building Parallel Programs.* Addison Wesley, 1996, available at `http://www.mcs.anl.gov/dbpp`

[13] I. Foster and S. Tuecke. Parallel Programming with PCN. *Technical Report ANL-91/32,* Argonne National Laboratory, Argonne, December 1991.

[14] G. Fox. What Have We Learnt from Using Real Parallel Machines to Solve Real Problems. In *Proceedings 3rd Conf. Hypercube Concurrent Computers and Applications,* 1988.

[15] G. Fox. Parallel Computing Comes of Age: Supercomputer Level Parallel Computations at Caltech. *Concurrency: Practice and Experience,* vol. 1 (1), pages 63-103, September 1989.

[16] M. Fruscione, P. Flocchini, E. Giudici, S. Punzi, and P. Stofella. Parallel Computational Frames: An Approach to Parallel Application Development based on Message Passing Systems. In *Programming Environments for Massively Parallel Distributed Systems,* Monte Verita, Italy, pages 117-126, 1994.

[17] R. M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM,* vol.33 (10), pages 30-53, October 1990.

[18] *Grand Challenging Applications.* `http://www.mcs.anl.gov/Projects/grand-challenges/`

[19] P. B. Hansen. Model Programs for Computational Science: A Programming Methodology for Multicomputers. *Concurrency: Practice and Experience,* vol. 5 (5), pages 407-423, 1993.

[20] D. Loveman. High-Performance Fortran. *IEEE Parallel and Distributed Technology,* vol. 1 (1), February 1993.

[21] B. Nitzberg and V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer,* vol. 24 (8), pages 52-60, 1991.

[22] D. Pritchard. Mathematical Models of Distributed Computation. In *Proceedings of OUG-7, Parallel Programming on Transputer Based Machines,* IOS Press, pages 25-36, 1988.

[23] G. Wilson. *Parallel Programming for Scientists and Engineers.* MIT Press, Cambridge, MA, 1995.

[24] G. Wilson and P. Lu. *Parallel Programming using C++.* MIT Press, Cambridge, MA, 1996.