NEAR EAST UNIVERSITY

Faculty of Engineering

DEPARTMENT OF COMPUTER ENGINEERING

PARALLEL PROCESSING AND DISTRIBUTED SYSTEMS

GRADUATION PROJECT
COM 400

Students : Seval Cömert (950134)
Koray Özdemir (950463)

Supervisor : Besime ERIN

Lefkoşa - 2000

ACKNOWLEDGEMENT

First of all, we want to thank to all instructors, who was help us about our lectures and social life during university period. Also we want to thank to our supervisor BESIME ERIN and for her interest which was help us to make easier this graduation project.

ABSTRACT

This graduation project is made of two main parts. They can be summarised as a Parallel Processing and a Distributed Processing. Parallel Processing has three chapters and the Distributed Processing has two chapters. The chapters in the project are written below:

CH 1 - Parallel computers and Computation,
CH 2 – Designing Parallel Algorithms,
CH 3 – Parallel Brunch-and-bound,
CH 4 – Distributed Processing,
CH 5 – Types of Distributed Systems,

While using the techniques in this project is possible to see how can make network systems and also show how can use network systems.

TABLE OF CONTENTS

INTRODUCTION

A first part of this project comprises that deal with the design of parallel programs. This part briefly introduces parallel computation and the importance of concurrency, scalability, locality, and modularity in parallel algorithm design.

The term parallel processing refers to a large class of methods that attempt to inrease computing speed by performing more than one computation concurrenty. A parallel procesor is a computer that implements some parallel processing technique. Like any type of computing, parallel processing can be viewed at various levels of complexity.

All modern computers involve some degree of parellelism. Parallelism offers new degrees of feedom into algorithms not found in sequential algorithms. The term *parallel processor* is designed to process more than one basic CPU instruction in parallel. Convential machines that can only execute one CPU instruction at a time are termed *sequantial processors*. (The trm uniprocessor is also used for such machines, principally to costract them with a particular class of parallel computers, namely, multiprocessors.)

The overhelming need for parallel algorithms arises from the fact that sequential algorithms are not efficient enough for many practically important problems.Parallel algorithms offer perhaps the greatest hope for major improvements. It is likely therefore that before the end of century all developments in both hardware and software will be concentrating on parallel processing.

Of what circulstances a parallel algorithm can be designed that will tremendously reduce the amount of time required to solve a problem in sequential envirenment. In the quest for answers to these kind of problems are centered around parallel algorithms and parallel architectures.

While many of these areas correspond to serial algorithms` development, a parallel algorithm designerneeds to know much more. Parallel algorithms, the languages they are writtenin, and the systems they run on are extremly more complex.

Distributed processing can loosely be described as the execution of co-operating processes which communicate by exchanging messages across an information network. This implies the IT infrastructure consists of distributed processors, enabling parallel execution of cesses and message exchanges. In this chapter, common models to support interaction ween processes executing in a distributed IT infrastructure are considered. The chapter focuses on the most widely used models of process interaction: the client/server model, the up model and the distributed object model.

Data exchanges between co-operating cesses can be implemented in two ways: using some common but passive resource or memory (a shared memory mechanism), or by supporting message exchanges between them.. Discussion is focused on two widely used message exchange mechanisms: message passing remote procedure calls **(RPC).** Some practical implementations will be examined to illustrate above approaches.

The term distributed processing will be used systems with multiple processor.proceessors can be interconnected in many ways for various reasons.In addition the term refers to a multiprocessor complex in one location, in its common usage,howewer the word distributed implies that the processor will in georaphically separete locations.occasionaly, this subject is applied to an operation using multiple minicomputers which wiil not connected at all

In fourth chapter will define the fundemental concepts and set the framework for distributed processing. Aim of the chapter will to answer 'what is dittributed processing' and why use distributig computing'questions.In addition to see the other tems how will be used with distributed processing:disributed fuction,distributed computing,networks,multicomputers,satellite processing backend processing, time sharing systems, and functionalyt modular systems.

A historical perspective on the evolotin of distributed systems revals a number of advantages and disadvantages. In practise both centralized and distributed approaches have a role to play of the IT infrastructure and information systems.A centralized system can be equally responsive to end user needs in the right circumtances and can offer superior, data integrity and systems management functionally. Aim of the fourth chapter to see advantages and disadvantages of distributed systems

There are several types of distributed processing in which the components are hooked together by telecommunications. Fifth chapter categorizes them and gives examples. İn this chapter types of distributed systems will describe.technolies for implementing application and information will also examined and will see many confirations of the future will be neither purely verttical nor purely horizontal.and there will be many application programs in the function distribution in addition will investigate reasons of the function and in this chapter will discuss function distribution which peripheral machines are not self-sufficient when isolated from their host by telecommunications.

Aim of the this chapter will compare the between types of distributed stystems and investigate each types and will see characteristics . to leran how will use these systems.

CHAPTER1

Parallel Computers and Computation

In this chapter, we review the role of parallelism in computing and introduce the parallel machine and programming models that will serve as the basis for subsequent discussion of algorithm design, performance analysis, and implementation.

After studying this chapter, you should be aware of the importance of concurrency, scalability, locality, and modularity in parallel program design. You should also be familiar with the idealized multicomputer model for which we shall design parallel algorithms, and the computation and communication abstractions that we shall use when describing parallel algorithms.

1.1 Parallelism and Computing

A *parallel computer* is a set of processors that are able to work cooperatively to solve a computational problem. This definition is broad enough to include parallel supercomputers that have hundreds or thousands of processors, networks of workstations, multiple-processor workstations, and embedded systems. Parallel computers are interesting because they offer the potential to concentrate computational resources whether processors, memory, or I/O bandwidth on important computational problems.

Parallelism has sometimes been viewed as a rare and exotic subarea of computing, interesting but of little relevance to the average programmer. A study of trends in applications, computer architecture, and networking shows that this view is no longer tenable. Parallelism is becoming ubiquitous, and parallel programming is becoming central to the programming enterprise.

1.1.1 Trends in Applications

As computers become ever faster, it can be tempting to suppose that they will eventually become ``fast enough'' and that appetite for increased computing power will be sated. However, history suggests that as a particular technology satisfies known applications, new applications will arise that are enabled by that technology and that will demand the development of new technology. As an amusing illustration of this phenomenon, a report prepared for the British government in the late 1940s concluded that Great Britain's computational requirements could be met by two or perhaps three computers. In those days, computers were used primarily for computing ballistics tables. The authors of the report did not consider other applications in science and engineering, let alone the commercial applications that would soon come to dominate computing. Similarly, the initial prospectus for Cray Research predicted a market for ten supercomputers; many hundreds have since been sold.

Traditionally, developments at the high end of computing have been motivated by numerical simulations of complex systems such as weather, climate, mechanical devices, electronic

circuits, manufacturing processes, and chemical reactions. However, the most significant forces driving the development of faster computers today are emerging commercial applications that require a computer to be able to process large amounts of data in sophisticated ways. These applications include video conferencing, collaborative work environments, computer-aided diagnosis in medicine, parallel databases used for decision support, and advanced graphics and virtual reality, particularly in the entertainment industry. For example, the integration of parallel computation, high-performance networking, and multimedia technologies is leading to the development of *video servers,* computers designed to serve hundreds or thousands of simultaneous requests for real-time video. Each video stream can involve both data transfer rates of many megabytes per second and large amounts of processing for encoding and decoding. In graphics, three-dimensional data sets are now approaching 10^9 volume elements (1024 on a side). At 200 operations per element, a display updated 30 times per second requires a computer capable of 6.4 * 10^12 operations per second. Although commercial applications may define the architecture of most future parall computers, traditional scientific applications will remain important users of parallel computing technology. Indeed, as nonlinear effects place limits on the insights offered by purely theoretical investigations and as experimentation becomes more costly or impractical, computational studies of complex systems are becoming ever more important. Computational costs typically increase as the fourth power or more of the ``resolution'' that determines accuracy, so these studies have a seemingly insatiable demand for more computer power. They are also often characterized by large memory and input/output requirements. For example, a ten-year simulation of the earth's climate using a state-of-the-art model may involve 10^16 floating-point operations ten days at an execution speed of 10^10 floating-point operations per second (10 gigaflops). This same simulation can easily generate a hundred gigabytes (10^11 bytes) or more of data. Yet as Table 1.1 shows, scientists can easily imagine refinements to these models that would increase these computational requirements 10,000 times.

TABLE 1.1

Current State	Needed Capability	Cost
100-km resolution	10-km resolution	10^2–10^3
Simple process representations	Improved process representations	2–10
Simple ocean	Fully coupled ocean	2–5
Simple atmospheric chemistry	Improved atmospheric chemistry	2–5
Limited biosphere	Comprehensive biosphere	about 2
Tens of years	Hundreds of years	10–10^2

Table 1.1: Various refinements proposed to climate models, and the increased computational requirements associated with these refinements. Altogether, these refinements could increase computational requirements by a factor of between 10^4 and 10^7.

In summary, the need for faster computers is driven by the demands of both data-intensive applications in commerce and computation-intensive applications in science and engineering. Increasingly, the requirements of these fields are merging, as scientific and engineering applications become more data intensive and commercial applications perform more sophisticated computations.

1.1.2 Trends in Computer Design

The performance of the fastest computers has grown exponentially from 1945 to the present, averaging a factor of 10 every five years. While the first computers performed a few tens of floating-point operations per second, the parallel computers of the mid-1990s achieve tens of billions of operations per second. Similar trends can be observed in the low-end computers of different eras: the calculators, personal computers, and workstations. There is little to suggest that this growth will not continue. However, the computer architectures used to sustain this growth are changing radically from sequential to parallel. The performance of a computer depends directly on the time required to perform a basic operation and the number of these basic operations that can be performed concurrently. The time to perform a basic operation is ultimately limited by the ``clock cycle'' of the processor, that is, the time required to perform the most primitive operation. However, clock cycle times are decreasing slowly and appear to be approaching physical limits such as the speed of light. We cannot depend on faster processors to provide increased computational performance. To circumvent these limitations, the designer may attempt to utilize internal concurrency in a chip, for example, by operating simultaneously on all 64 bits of two numbers that are to be multiplied. However, a fundamental result in Very Large Scale Integration (VLSI) complexity theory says that this strategy is expensive. This result states that for certain transitive computations (in which any output may depend on any input), the chip area A and the time T required to perform this computation are related so that AT^2must exceed some problem-dependent function of problem size. This result can be explained informally by assuming that a computation must move a certain amount of information from one side of a square chip to the other. The amount of information that can be moved in a time unit is limited by the cross section of the chip, sqrt(A). This gives a transfer rate of sqrt(AT), from which the AT^2 relation is obtained. To decrease the time required to move the information by a certain factor, the cross section must be increased by the same factor, and hence the total area must be increased by the square of that factor.

This AT^2 result means that not only is it difficult to build individual components that operate faster, it may not even be desirable to do so. It may be cheaper to use more, slower components. For example, if we have an area n^2A of silicon to use in a computer, we can either build n^2 components, each of size A and able to perform an operation in time T , or build a single component able to perform the same operation in time T/n . The multicomponent system is potentially n times faster.

Computer designers use a variety of techniques to overcome these limitations on single computer performance, including pipelining (different stages of several instructions execute concurrently) and multiple function units (several multipliers, adders, etc., are controlled by a single instruction stream). Increasingly, designers are incorporating multiple ``computers,'' each with its own processor, memory, and associated interconnection logic. This approach is facilitated by advances in VLSI technology that continue to decrease the number of components required to implement a computer. As the cost of a computer is (very approximately) proportional to the number of components that it contains, increased integration also increases the number of processors that can be included in a computer for a particular cost. The result is continued growth in processor counts.

1.1.3 Trends in Networking

Another important trend changing the face of computing is an enormous increase in the capabilities of the networks that connect computers. Not long ago, high-speed networks ran at 1.5 Mbits per second; by the end of the 1990s, bandwidths in excess of 1000 Mbits per second will be commonplace. Significant improvements in reliability are also expected. These trends make it feasible to develop applications that use physically distributed resources as if they were

part of the same computer. A typical application of this sort may utilize processors on multiple remote computers, access a selection of remote databases, perform rendering on one or more graphics computers, and provide real-time output and control on a workstation.

We emphasize that computing on networked computers (``distributed computing'') is not just a subfield of parallel computing. Distributed computing is deeply concerned with problems such as reliability, security, and heterogeneity that are generally regarded as tangential in parallel computing. (As Leslie Lamport has observed, ``A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.'') Yet the basic task of developing programs that can run on many computers at once is a parallel computing problem. In this respect, the previously distinct worlds of parallel and distributed computing are converging.

1.2 A Parallel Machine Model

The rapid penetration of computers into commerce, science, and education owed much to the early standardization on a single machine model, the von Neumann computer. A von Neumann computer comprises a central processing unit (CPU) connected to a storage unit (memory) (Figure 1.1). The CPU executes a stored program that specifies a sequence of read and write operations on the memory. This simple model has proved remarkably robust. Its persistence over more than forty years has allowed the study of such important topics as algorithms and programming languages to proceed to a large extent independently of developments in computer architecture. Consequently, programmers can be trained in the abstract art of ``programming'' rather than the craft of ``programming machine X'' and can design algorithms for an abstract von Neumann machine, confident that these algorithms will execute on most target computers with reasonable efficiency.

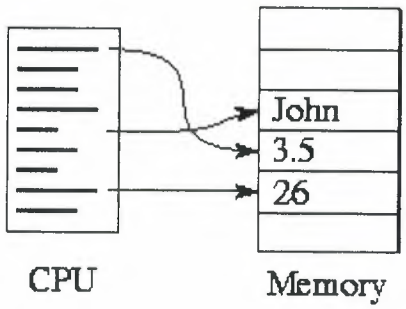

Figure 1.1: *The von Neumann computer. A central processing unit (CPU) executes a program that performs a sequence of read and write operations on an attached memory.*

Our study of parallel programming will be most rewarding if we can identify a parallel machine model that is as general and useful as the von Neumann sequential machine model. This machine model must be both simple and realistic: *simple* to facilitate understanding and programming, and *realistic* to ensure that programs developed for the model execute with reasonable efficiency on real computers.

1.2.1 The Multicomputer

A parallel machine model called the *multicomputer* fits these requirements. As illustrated in Figure1.2, a multicomputer comprises a number of von Neumann computers, or *nodes*, linked by an *interconnection network*. Each computer executes its own program. This program may access local memory and may send and receive messages over the network. Messages are used to

communicate with other computers or, equivalently, to read and write remote memories. In the idealized network, the cost of sending a message between two nodes is independent of both node location and other network traffic, but does depend on message length.

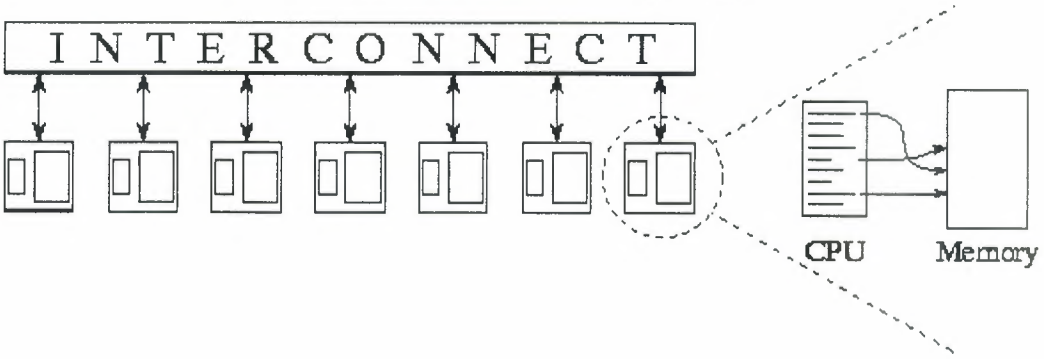

Figure 1.2: *The multicomputer, an idealized parallel computer model. Each node consists of a von Neumann machine: a CPU and memory. A node can communicate with other nodes by sending and receiving messages over an interconnection network.*

A defining attribute of the multicomputer model is that accesses to local (same-node) memory are less expensive than accesses to remote (different-node) memory. That is, read and write are less costly than send and receive. Hence, it is desirable that accesses to local data be more frequent than accesses to remote data. This property, called *locality*, is a third fundamental requirement for parallel software, in addition to concurrency and scalability. The importance of locality depends on the ratio of remote to local access costs. This ratio can vary from 10:1 to 1000:1 or greater, depending on the relative performance of the local computer, the network, and the mechanisms used to move data to and from the network.

1.2.2 Other Machine Models

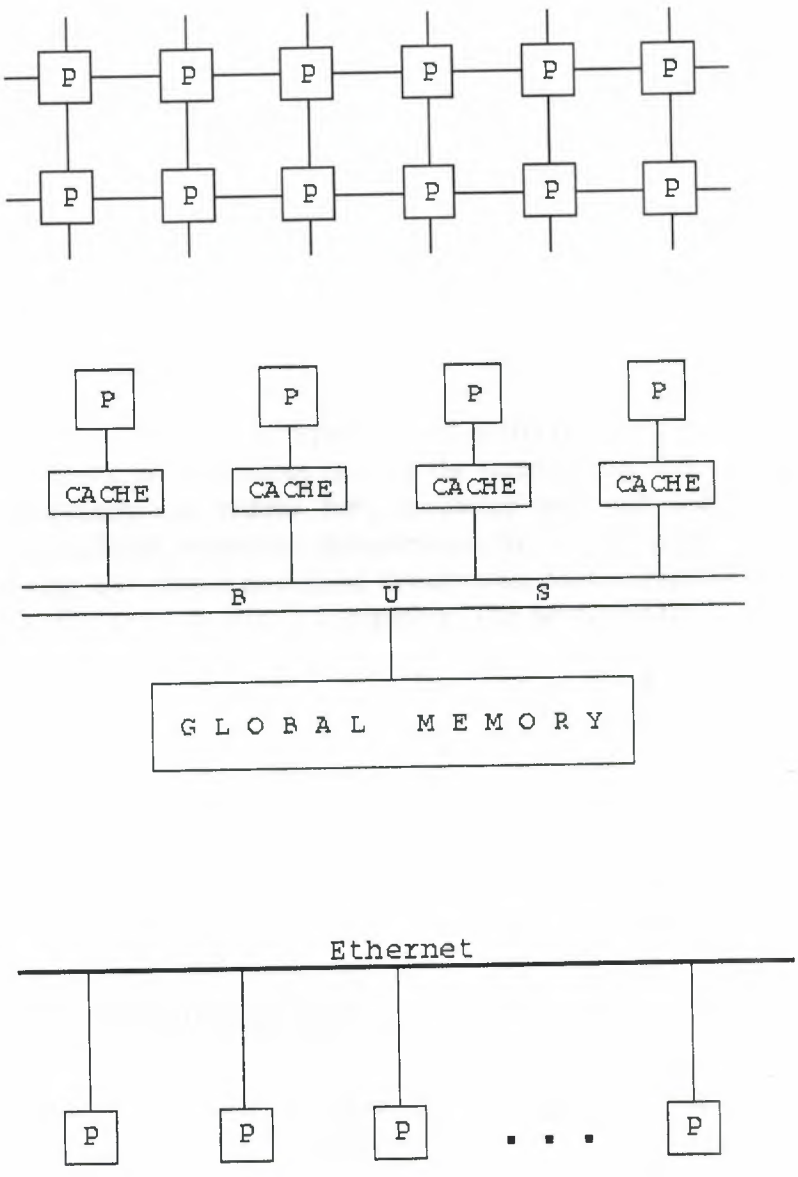

Figure 1.3: *Classes of parallel computer architecture. From top to bottom: a distributed-memory MIMD computer with a mesh interconnect, a shared-memory multiprocessor, and a local area network (in this case, an Ethernet). In each case, P denotes an independent processor.*

We review important parallel computer architectures (several are illustrated in Figure 1.3) and discuss briefly how these differ from the idealized multicomputer model.

The multicomputer is most similar to what is often called the *distributed-memory MIMD* (multiple instruction multiple data) computer. MIMD means that each processor can execute a separate stream of instructions on its own local data; distributed memory means that memory is distributed among the processors, rather than placed in a central location. The principal difference between a multicomputer and the distributed-memory MIMD computer is that in the latter, the cost of sending a message between two nodes may not be independent of node location

and other network traffic. Examples of this class of machine include the IBM SP, Intel Paragon, Thinking Machines CM5, Cray T3D, Meiko CS-2, and nCUBE.

Another important class of parallel computer is the *multiprocessor*, or shared-memory MIMD computer. In multiprocessors, all processors share access to a common memory, typically via a bus or a hierarchy of buses. In the idealized Parallel Random Access Machine (PRAM) model, often used in theoretical studies of parallel algorithms, any processor can access any memory element in the same amount of time. In practice, scaling this architecture usually introduces some form of memory hierarchy; in particular, the frequency with which the shared memory is accessed may be reduced by storing copies of frequently used data items in a *cache* associated with each processor. Access to this cache is much faster than access to the shared memory; hence, locality is usually important, and the differences between multicomputers and multiprocessors are really just questions of degree. Programs developed for multicomputers can also execute efficiently on multiprocessors, because shared memory permits an efficient implementation of message passing. Examples of this class of machine include the Silicon Graphics Challenge, Sequent Symmetry, and the many multiprocessor workstations.

A more specialized class of parallel computer is the *SIMD* (single instruction multiple data) computer. In SIMD machines, all processors execute the same instruction stream on a different piece of data. This approach can reduce both hardware and software complexity but is appropriate only for specialized problems characterized by a high degree of regularity, for example, image processing and certain numerical simulations. Multicomputer algorithms *cannot* in general be executed efficiently on SIMD computers. The MasPar MP is an example of this class of machine.

Two classes of computer system that are sometimes used as parallel computers are the local area network (LAN), in which computers in close physical proximity (e.g., the same building) are connected by a fast network, and the wide area network (WAN), in which geographically distributed computers are connected. Although systems of this sort introduce additional concerns such as reliability and security, they can be viewed for many purposes as multicomputers, albeit with high remote-access costs. Ethernet and asynchronous transfer mode (ATM) are commonly used network technologies.

1.3 A Parallel Programming Model

The von Neumann machine model assumes a processor able to execute sequences of instructions. An instruction can specify, in addition to various arithmetic operations, the address of a datum to be read or written in memory and/or the address of the next instruction to be executed. While it is possible to program a computer in terms of this basic model by writing machine language, this method is for most purposes prohibitively complex, because we must keep track of millions of memory locations and organize the execution of thousands of machine instructions. Hence, modular design techniques are applied, whereby complex programs are constructed from simple components, and components are structured in terms of higher-level abstractions such as data structures, iterative loops, and procedures. Abstractions such as procedures make the exploitation of modularity easier by allowing objects to be manipulated without concern for their internal structure. So do high-level languages such as Fortran, Pascal, C, and Ada, which allow designs expressed in terms of these abstractions to be translated automatically into executable code.

Parallel programming introduces additional sources of complexity: if we were to program at the lowest level, not only would the number of instructions executed increase, but we would also need to manage explicitly the execution of thousands of processors and coordinate millions of

interprocessor interactions. Hence, abstraction and modularity are at least as important as in sequential programming. In fact, we shall emphasize *modularity* as a fourth fundamental requirement for parallel software, in addition to concurrency, scalability, and locality.

1.3.1 Tasks and Channels

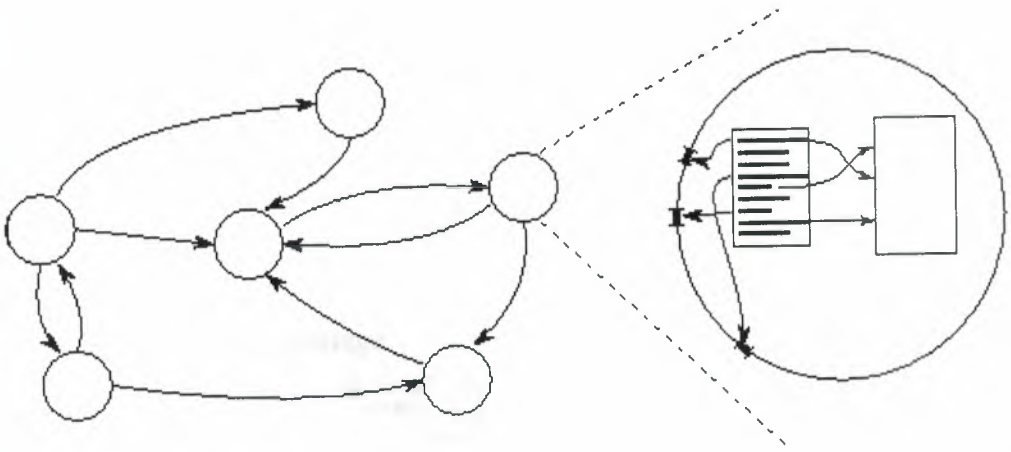

Figure 1.4: *A simple parallel programming model. The figure shows both the instantaneous state of a computation and a detailed picture of a single task. A computation consists of a set of tasks (represented by circles) connected by channels (arrows). A task encapsulates a program and local memory and defines a set of ports that define its interface to its environment. A channel is a message queue into which a sender can place messages and from which a receiver can remove messages, ``blocking'' if messages are not available.*

We consider next the question of which abstractions are appropriate and useful in a parallel programming model. Clearly, mechanisms are needed that allow explicit discussion about concurrency and locality and that facilitate development of scalable and modular programs. Also needed are abstractions that are simple to work with and that match the architectural model, the multicomputer. While numerous possible abstractions could be considered for this purpose, two fit these requirements particularly well: the *task* and *channel*. These are illustrated in Figure 1.5 and can be summarized as follows:

1. A parallel computation consists of one or more tasks. Tasks execute concurrently. The number of tasks can vary during program execution.
2. A task encapsulates a sequential program and local memory. (In effect, it is a virtual von Neumann machine.) In addition, a set of *inports* and *outports* define its interface to its environment.
3. A task can perform four basic actions in addition to reading and writing its local memory (Figure 1.5): send messages on its outports, receive messages on its inports, create new tasks, and terminate.
4. A send operation is asynchronous: it completes immediately. A receive operation is synchronous: it causes execution of the task to block until a message is available.
5. Outport/inport pairs can be connected by message queues called *channels*. Channels can be created and deleted, and references to channels (ports) can be included in messages, so connectivity can vary dynamically.

6. Tasks can be mapped to physical processors in various ways; the mapping employed does not affect the semantics of a program. In particular, multiple tasks can be mapped to a single processor. (We can also imagine a single task being mapped to multiple processors, but that possibility is not considered here.)

The task abstraction provides a mechanism for talking about locality: data contained in a task's local memory are ``close"; other data are ``remote." The channel abstraction provides a mechanism for indicating that computation in one task requires data in another task in order to proceed. (This is termed a *data dependency*). The following simple example illustrates some of these features.

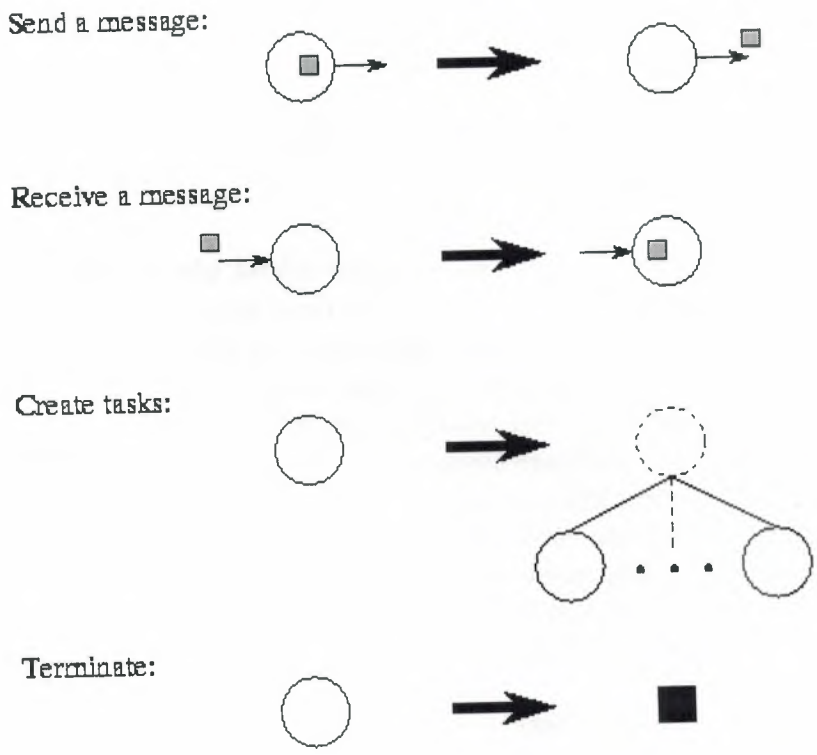

Figure 1.5: *The four basic task actions. In addition to reading and writing local memory, a task can send a message, receive a message, create new tasks (suspending until they terminate), and terminate.*

Example1. 1 Bridge Construction:

Consider the following real-world problem. A bridge is to be assembled from girders being constructed at a foundry. These two activities are organized by providing trucks to transport girders from the foundry to the bridge site. This situation is illustrated in Figure 1.6 (a), with the foundry and bridge represented as tasks and the stream of trucks as a channel. Notice that this approach allows assembly of the bridge and construction of girders to proceed in parallel without any explicit coordination: the foundry crew puts girders on trucks as they are produced, and the assembly crew adds girders to the bridge as and when they arrive.

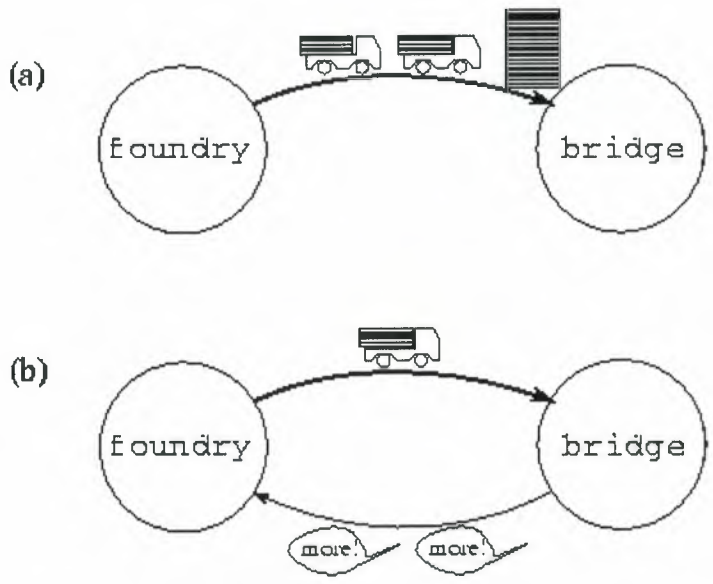

Figure 1.6: *Two solutions to the bridge construction problem. Both represent the foundry and the bridge assembly site as separate tasks,* `foundry` *and* `bridge`*. The first uses a single channel on which girders generated by* `foundry` *are transported as fast as they are generated. If* `foundry` *generates girders faster than they are consumed by* `bridge`*, then girders accumulate at the construction site. The second solution uses a second channel to pass flow control messages from* `bridge` *to* `foundry` *so as to avoid overflow.*

A disadvantage of this scheme is that the foundry may produce girders much faster than the assembly crew can use them. To prevent the bridge site from overflowing with girders, the assembly crew instead can explicitly request more girders when stocks run low. This refined approach is illustrated in Figure 1.6 (b), with the stream of requests represented as a second channel. The second channel can also be used to shut down the flow of girders when the bridge is complete.

We now examine some other properties of this task/channel programming model: performance, mapping independence, modularity, and determinism.

Performance. Sequential programming abstractions such as procedures and data structures are effective because they can be mapped simply and efficiently to the von Neumann computer. The task and channel have a similarly direct mapping to the multicomputer. A task represents a piece of code that can be executed sequentially, on a single processor. If two tasks that share a channel are mapped to different processors, the channel connection is implemented as interprocessor communication; if they are mapped to the same processor, some more efficient mechanism can be used.

Mapping Independence. Because tasks interact using the same mechanism (channels) regardless of task location, the result computed by a program does not depend on where tasks execute. Hence, algorithms can be designed and implemented without concern for the number of processors on which they will execute; in fact, algorithms are frequently designed that create many more tasks than processors. This is a straightforward way of achieving *scalability* : as the number of processors increases, the number of tasks per processor is reduced but the algorithm itself need not be modified. The creation of more tasks than processors can also serve to mask communication delays, by providing other computation that can be performed while communication is performed to access remote data.

Modularity. In modular program design, various components of a program are developed separately, as independent modules, and then combined to obtain a complete program. Interactions between modules are restricted to well-defined interfaces. Hence, module implementations can be changed without modifying other components, and the properties of a program can be determined from the specifications for its modules and the code that plugs these modules together. When successfully applied, modular design reduces program complexity and facilitates code reuse.

The task is a natural building block for modular design.A task encapsulates both data and the code that operates on those data; the ports on which it sends and receives messages constitute its interface. Hence, the advantages of modular design summarized in the previous paragraph are directly accessible in the task/channel model.

Strong similarities exist between the task/channel model and the popular object-oriented programming paradigm. Tasks, like objects, encapsulate data and the code that operates on those data. Distinguishing features of the task/channel model are its concurrency, its use of channels rather than method calls to specify interactions, and its lack of support for inheritance.

Determinism. An algorithm or program is deterministic if execution with a particular input always yields the same output. It is nondeterministic if multiple executions with the same input can give different outputs. Although nondeterminism is sometimes useful and must be supported, a parallel programming model that makes it easy to write deterministic programs is highly desirable. Deterministic programs tend to be easier to understand. Also, when checking for correctness, only one execution sequence of a parallel program needs to be considered, rather than all possible executions.

The "arms-length" interactions supported by the task/channel model makes determinism relatively easy to guarantee. As we shall see in Part II when we consider programming tools, it suffices to verify that each channel has a single sender and a single receiver and that a task receiving on a channel blocks until a receive operation is complete. These conditions can be relaxed when nondeterministic interactions are required.

In the bridge construction example, determinism means that the same bridge will be constructed regardless of the rates at which the foundry builds girders and the assembly crew puts girders together. If the assembly crew runs ahead of the foundry, it will block, waiting for girders to arrive. Hence, it simply suspends its operations until more girders are available, rather than attempting to continue construction with half-completed girders. Similarly, if the foundry produces girders faster than the assembly crew can use them, these girders simply accumulate until they are needed. Determinism would be guaranteed even if several bridges were constructed simultaneously: As long as girders destined for different bridges travel on distinct channels, they cannot be confused.

1.3.2 Other Programming Models

In subsequent chapters, the task/channel model will often be used to describe algorithms. However, this model is certainly not the only approach that can be taken to representing parallel computation. Many other models have been proposed, differing in their flexibility, task interaction mechanisms, task granularities, and support for locality, scalability, and modularity. Here, we review several alternatives.

Message passing. Message passing is probably the most widely used parallel programming model today. Message-passing programs, like task/channel programs, create multiple tasks, with each task encapsulating local data. Each task is identified by a unique name, and tasks interact by sending and receiving messages to and from named tasks. In this respect, message passing is really just a minor variation on the task/channel model, differing only in the mechanism used for data transfer. For example, rather than sending a message on ``channel `ch`,'' we may send a

message to ``task 17.",We explain that the definition of channels is a useful discipline even when designing message-passing programs, because it forces us to conceptualize the communication structure of a parallel program.

The message-passing model does not preclude the dynamic creation of tasks, the execution of multiple tasks per processor, or the execution of different programs by different tasks. However, in practice most message-passing systems create a fixed number of identical tasks at program startup and do not allow tasks to be created or destroyed during program execution. These systems are said to implement a *single program multiple data* (SPMD) programming model because each task executes the same program but operates on different data. As explained in subsequent chapters, the SPMD model is sufficient for a wide range of parallel programming problems but does hinder some parallel algorithm developments.

Data Parallelism. Another commonly used parallel programming model, data parallelism, calls for exploitation of the concurrency that derives from the application of the same operation to multiple elements of a data structure, for example, ``add 2 to all elements of this array," or ``increase the salary of all employees with 5 years service." A data-parallel program consists of a sequence of such operations. As each operation on each data element can be thought of as an independent task, the natural granularity of a data-parallel computation is small, and the concept of ``locality" does not arise naturally. Hence, data-parallel compilers often require the programmer to provide information about how data are to be distributed over processors, in other words, how data are to be partitioned into tasks. The compiler can then translate the data-parallel program into an SPMD formulation, thereby generating communication code automatically.We show that the algorithm design and analysis techniques developed for the task/channel model apply directly to data-parallel programs; in particular, they provide the concepts required to understand the locality and scalability of data-parallel programs.

Shared Memory. In the shared-memory programming model, tasks share a common address space, which they read and write asynchronously. Various mechanisms such as locks and semaphores may be used to control access to the shared memory. An advantage of this model from the programmer's point of view is that the notion of data ``ownership" is lacking, and hence there is no need to specify explicitly the communication of data from producers to consumers. This model can simplify program development. However, understanding and managing locality becomes more difficult, an important consideration (as noted earlier) on most shared-memory architectures. It can also be more difficult to write deterministic programs.

1.4 Parallel Algorithm Examples

We conclude this chapter by presenting two examples of parallel algorithms. We do not concern ourselves here with the process by which these algorithms are derived or with their efficiency. The goal is simply to introduce parallel algorithms and their description in terms of tasks and channels.

The first algorithm creates tasks dynamically during program execution, and the second uses a fixed number of tasks but has different tasks perform different functions.

1.4.1 Search

The first example illustrates the dynamic creation of tasks and channels during program execution. This problem can be structured as follows. Initially, a single task is created for the root of the tree. A task evaluates its node and then, if that node is not a solution, creates a new task for each `search` call (subtree). A channel created for each new task is used to return to the

new task's parent any solutions located in its subtree. Hence, new tasks and channels are created in a wavefront as the search progresses down the search tree (Figure1.7).

```
procedure search(A)
begin
if (solution(A)) then
score=oval(A)
report solution and score
else
foreach child A(I) of A
search(A(I))
endfor
endif
end.
```

Algorithm 1.1 : A recursive formulation of a simple search algorithm. When called to expand a search tree node, this procedure cheks to see whether the node is question represents a solution, If not algorithim makes recursive calls to the same procedure to expand each of the offspring nodes.

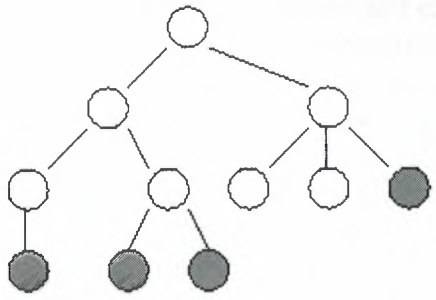

Figure 1.7: *Task structure for the search example. Each circle represents a node in the search tree and hence a call to the* `search` *procedure. A task is created for each node in the tree as it is explored. At any one time, some tasks are actively engaged in expanding the tree further (these are shaded in the figure); others have reached solution nodes and are terminating, or are waiting for their offspring to report back with solutions. The lines represent the channels used to return solutions.*

1.4.2 Parameter Study

In so-called embarrassingly parallel problems, a computation consists of a number of tasks that can execute more or less independently, without communication. These problems are usually easy to adapt for parallel execution. An example is a parameter study, in which the same computation must be performed using a range of different input parameters. The parameter values are read from an input file, and the results of the different computations are written to an output file.

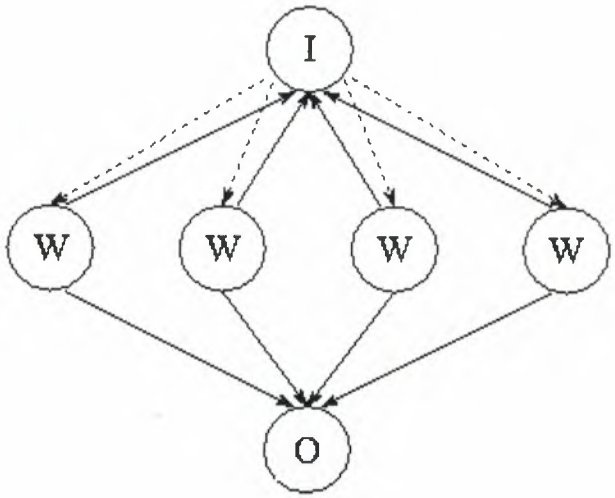

Figure 1.8: *Task structure for parameter study problem. Workers (W) request parameters from the input task (I) and send results to the output task (O). Note the many-to-one connections: this program is nondeterministic in that the input and output tasks receive data from workers in whatever order the data are generated. Reply channels, represented as dashed lines, are used to communicate parameters from the input task to workers.*

If the execution time per problem is constant and each processor has the same computational power, then it suffices to partition available problems into equal-sized sets and allocate one such set to each processor. In other situations, we may choose to use the task structure illustrated in Figure 1.8. The input and output tasks are responsible for reading and writing the input and output files, respectively. Each worker task (typically one per processor) repeatedly requests parameter values from the input task, computes using these values, and sends results to the output task. Because execution times vary, the input and output tasks cannot expect to receive messages from the various workers in any particular order. Instead, a many-to-one communication structure is used that allows them to receive messages from the various workers in arrival order.

The input task responds to a worker request by sending a parameter to that worker. Hence, a worker that has sent a request to the input task simply waits for the parameter to arrive on its reply channel. In some cases, efficiency can be improved by *prefetching* , that is, requesting the next parameter before it is needed. The worker can then perform computation while its request is being processed by the input task.

Because this program uses many-to-one communication structures, the order in which computations are performed is not necessarily determined. However, this nondeterminism affects only the allocation of problems to workers and the ordering of results in the output file, not the actual results computed.

CHAPTER 2

Designing Parallel Algorithms

Now that we have discussed what parallel algorithms look like, we are ready to examine how they can be designed. In this chapter, we show how a problem specification is translated into an algorithm that displays concurrency, scalability, and locality. Parallel algorithm design is not easily reduced to simple recipes. Rather, it requires the sort of integrative thought that is commonly referred to as ``creativity.'' However, it *can* benefit from a methodical approach that maximizes the range of options considered, that provides mechanisms for evaluating alternatives, and that reduces the cost of backtracking from bad choices. We describe such an approach and illustrate its application to a range of problems. Our goal is to suggest a framework within which parallel algorithm design can be explored. In the process, we hope you will develop intuition as to what constitutes a good parallel algorithm.
You should be able to partition computations, using both domain and functional decomposition techniques, and know how to recognize and implement both local and global, static and dynamic, structured and unstructured, and synchronous and asynchronous communication structures. You should also be able to use agglomeration as a means of reducing communication and implementation costs and should be familiar with a range of load-balancing strategies.

2.1 Methodical Design

Most programming problems have several parallel solutions. The best solution may differ from that suggested by existing sequential algorithms. The design methodology that we describe is intended to foster an exploratory approach to design in which machine-independent issues such as concurrency are considered early and machine-specific aspects of design are delayed until late in the design process. This methodology structures the design process as four distinct stages: partitioning, communication, agglomeration, and mapping. (The acronym PCAM may serve as a useful reminder of this structure.) In the first two stages, we focus on concurrency and scalability and seek to discover algorithms with these qualities. In the third and fourth stages, attention shifts to locality and other performance-related issues. The four stages are illustrated in Figure 2.1, and can be summarized as follows:

1. *Partitioning.* The computation that is to be performed and the data operated on by this computation are decomposed into small tasks. Practical issues such as the number of processors in the target computer are ignored, and attention is focused on recognizing opportunities for parallel execution.
2. *Communication.* The communication required to coordinate task execution is determined, and appropriate communication structures and algorithms are defined.
3. *Agglomeration.* The task and communication structures defined in the first two stages of a design are evaluated with respect to performance requirements and implementation

costs. If necessary, tasks are combined into larger tasks to improve performance or to reduce development costs.

4. *Mapping.* Each task is assigned to a processor in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs. Mapping can be specified statically or determined at runtime by load-balancing algorithms.

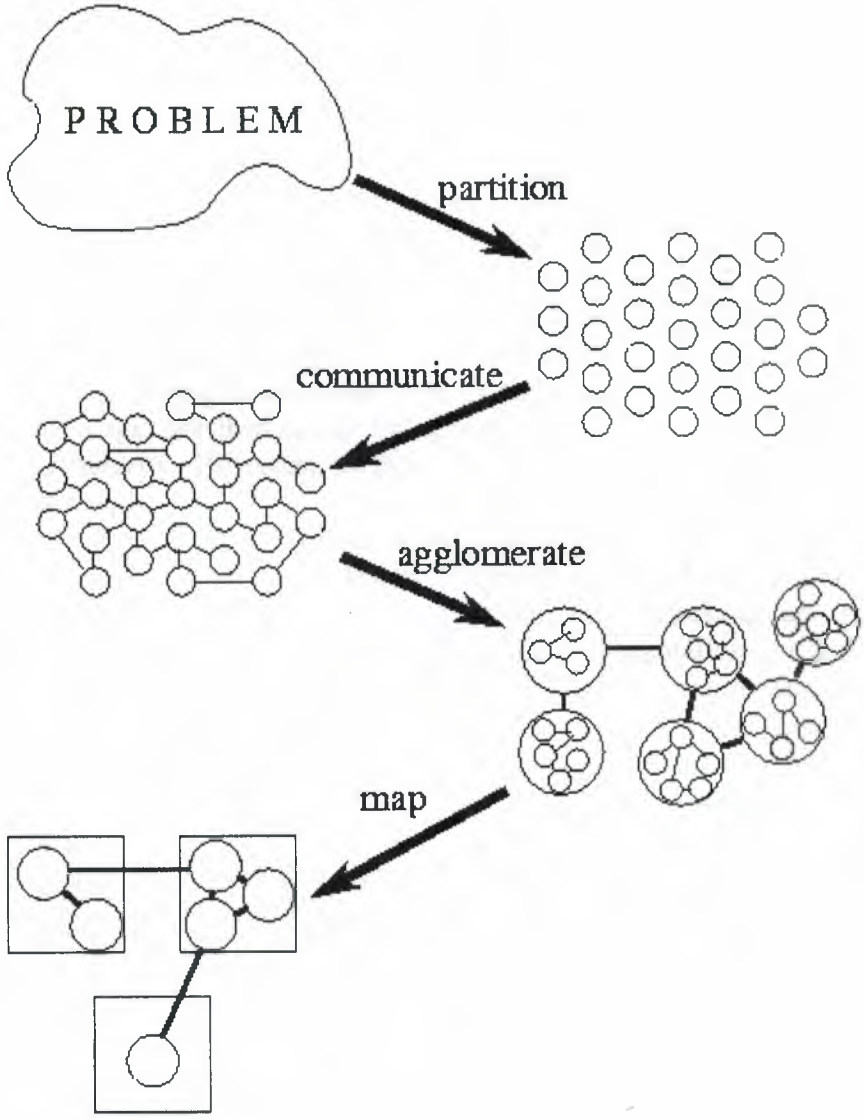

Figure 2.1: *PCAM: a design methodology for parallel programs. Starting with a problem specification, we develop a partition, determine communication requirements, agglomerate tasks, and finally map tasks to processors.*

The outcome of this design process can be a program that creates and destroys tasks dynamically, using load-balancing techniques to control the mapping of tasks to processors. Alternatively, it can be an SPMD program that creates exactly one task per processor. The same process of algorithm discovery applies in both cases, although if the goal is to produce an SPMD program, issues associated with mapping are subsumed into the agglomeration phase of the design.

Algorithm design is presented here as a sequential activity. In practice, however, it is a highly parallel process, with many concerns being considered simultaneously. Also, although we seek to avoid backtracking, evaluation of a partial or complete design may require changes to design decisions made in previous steps.

The following sections provide a detailed examination of the four stages of the design process. We present basic principles, use examples to illustrate the application of these principles, and include design checklists that can be used to evaluate designs as they are developed. In the final sections of this chapter, we use three case studies to illustrate the application of these design techniques to realistic problems.

2.2 Partitioning

The partitioning stage of a design is intended to expose opportunities for parallel execution. Hence, the focus is on defining a large number of small tasks in order to yield what is termed a *fine-grained* decomposition of a problem. Just as fine sand is more easily poured than a pile of bricks, a fine-grained decomposition provides the greatest flexibility in terms of potential parallel algorithms. In later design stages, evaluation of communication requirements, the target architecture, or software engineering issues may lead us to forego opportunities for parallel execution identified at this stage. We then revisit the original partition and agglomerate tasks to increase their size, or granularity. However, in this first stage we wish to avoid prejudging alternative partitioning strategies.

A good partition divides into small pieces both the *computation* associated with a problem and the *data* on which this computation operates. When designing a partition, programmers most commonly first focus on the data associated with a problem, then determine an appropriate partition for the data, and finally work out how to associate computation with data. This partitioning technique is termed *domain decomposition*. The alternative approach first decomposing the computation to be performed and then dealing with the data is termed *functional decomposition*. These are complementary techniques which may be applied to different components of a single problem or even applied to the same problem to obtain alternative parallel algorithms.

In this first stage of a design, we seek to avoid replicating computation and data; that is, we seek to define tasks that partition both computation and data into disjoint sets. Like granularity, this is an aspect of the design that we may revisit later. It can be worthwhile replicating either computation or data if doing so allows us to reduce communication requirements.

2.2.1 Domain Decomposition

In the domain decomposition approach to problem partitioning, we seek first to decompose the data associated with a problem. If possible, we divide these data into small pieces of approximately equal size. Next, we partition the computation that is to be performed, typically by associating each operation with the data on which it operates. This partitioning yields a number of tasks, each comprising some data and a set of operations on that data. An operation may require data from several tasks. In this case, communication is required to move data between tasks. This requirement is addressed in the next phase of the design process.

The data that are decomposed may be the input to the program, the output computed by the program, or intermediate values maintained by the program. Different partitions may be possible, based on different data structures. Good rules of thumb are to focus first on the largest data structure or on the data structure that is accessed most frequently. Different phases of the computation may operate on different data structures or demand different decompositions for the

same data structures. In this case, we treat each phase separately and then determine how the decompositions and parallel algorithms developed for each phase fit together.

Figure 2.2 illustrates domain decomposition in a simple problem involving a three-dimensional grid. (This grid could represent the state of the atmosphere in a weather model, or a three-dimensional space in an image-processing problem.) Computation is performed repeatedly on each grid point. Decompositions in the x, y, and/or z dimensions are possible. In the early stages of a design, we favor the most aggressive decomposition possible, which in this case defines one task for each grid point. Each task maintains as its state the various values associated with its grid point and is responsible for the computation required to update that state.

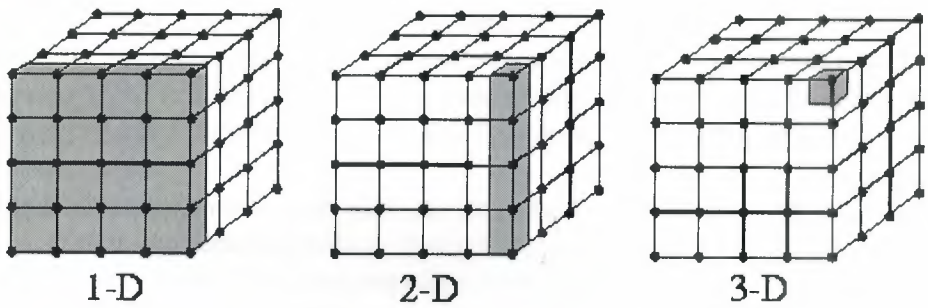

Figure 2.2: *Domain decompositions for a problem involving a three-dimensional grid. One-, two-, and three-dimensional decompositions are possible; in each case, data associated with a single task are shaded. A three-dimensional decomposition offers the greatest flexibility and is adopted in the early stages of a design.*

2.2.2 Functional Decomposition

Functional decomposition represents a different and complementary way of thinking about problems. In this approach, the initial focus is on the computation that is to be performed rather than on the data manipulated by the computation. If we are successful in dividing this computation into disjoint tasks, we proceed to examine the data requirements of these tasks. These data requirements may be disjoint, in which case the partition is complete. Alternatively, they may overlap significantly, in which case considerable communication will be required to avoid replication of data. This is often a sign that a domain decomposition approach should be considered instead.

While domain decomposition forms the foundation for most parallel algorithms, functional decomposition is valuable as a different way of thinking about problems. For this reason alone, it should be considered when exploring possible parallel algorithms. A focus on the computations that are to be performed can sometimes reveal structure in a problem, and hence opportunities for optimization, that would not be obvious from a study of data alone.

As an example of a problem for which functional decomposition is most appropriate, consider Algorithm 1.1. This explores a search tree looking for nodes that correspond to "solutions." The algorithm does not have any obvious data structure that can be decomposed. Initially, a single task is created for the root of the tree. A task evaluates its node and then, if that node is not a leaf, creates a new task for each `search` call (subtree). As illustrated in Figure 1.7, new tasks are created in a wavefront as the search tree is expanded.

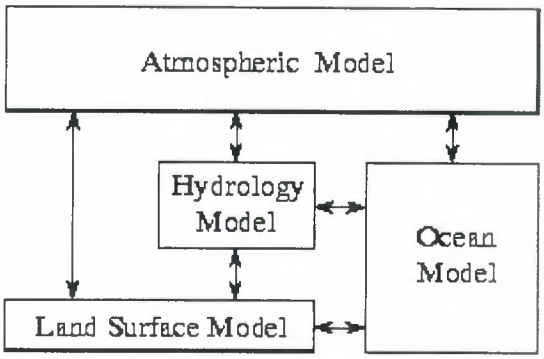

Figure 2.3: *Functional decomposition in a computer model of climate. Each model component can be thought of as a separate task, to be parallelized by domain decomposition. Arrows represent exchanges of data between components during computation: the atmosphere model generates wind velocity data that are used by the ocean model, the ocean model generates sea surface temperature data that are used by the atmosphere model, and so on.*

Functional decomposition also has an important role to play as a program structuring technique. A functional decomposition that partitions not only the computation that is to be performed but also the code that performs that computation is likely to reduce the complexity of the overall design. This is often the case in computer models of complex systems, which may be structured as collections of simpler models connected via interfaces. For example, a simulation of the earth's climate may comprise components representing the atmosphere, ocean, hydrology, ice, carbon dioxide sources, and so on. While each component may be most naturally parallelized using domain decomposition techniques, the parallel algorithm as a whole is simpler if the system is first decomposed using functional decomposition techniques, even though this process does not yield a large number of tasks (Figure 2.3).

2.2.3 Partitioning Design Checklist

The partitioning phase of a design should produce one or more possible decompositions of a problem. Before proceeding to evaluate communication requirements, we use the following checklist to ensure that the design has no obvious flaws. Generally, all these questions should be answered in the affirmative.

1. Does your partition define at least an order of magnitude more tasks than there are processors in your target computer? If not, you have little flexibility in subsequent design stages.
2. Does your partition avoid redundant computation and storage requirements? If not, the resulting algorithm may not be scalable to deal with large problems.
3. Are tasks of comparable size? If not, it may be hard to allocate each processor equal amounts of work.
4. Does the number of tasks scale with problem size? Ideally, an increase in problem size should increase the number of tasks rather than the size of individual tasks. If this is not the case, your parallel algorithm may not be able to solve larger problems when more processors are available.
5. Have you identified several alternative partitions? You can maximize flexibility in subsequent design stages by considering alternatives now. Remember to investigate both domain and functional decompositions.

Answers to these questions may suggest that, despite careful thought in this and subsequent design stages, we have a ``bad" design. In this situation it is risky simply to push ahead with implementation. We should use the performance evaluation techniques to determine whether the

design meets our performance goals despite its apparent deficiencies. We may also wish to revisit the problem specification. Particularly in science and engineering applications, where the problem to be solved may involve a simulation of a complex physical process, the approximations and numerical techniques used to develop the simulation can strongly influence the ease of parallel implementation. In some cases, optimal sequential and parallel solutions to the same problem may use quite different solution techniques. While detailed discussion of these issues is beyond the scope of this book.

2.3 Communication

The tasks generated by a partition are intended to execute concurrently but cannot, in general, execute independently. The computation to be performed in one task will typically require data associated with another task. Data must then be transferred between tasks so as to allow computation to proceed. This information flow is specified in the *communication* phase of a design.

We conceptualize a need for communication between two tasks as a channel linking the tasks, on which one task can send messages and from which the other can receive. Hence, the communication associated with an algorithm can be specified in two phases. First, we define a channel structure that links, either directly or indirectly, tasks that require data (consumers) with tasks that possess those data (producers). Second, we specify the messages that are to be sent and received on these channels. Depending on our eventual implementation technology, we may not actually create these channels when coding the algorithm. For example, in a data-parallel language, we simply specify data-parallel operations and data distributions. Nevertheless, thinking in terms of tasks and channels helps us to think quantitatively about locality issues and communication costs.

The definition of a channel involves an intellectual cost and the sending of a message involves a physical cost. Hence, we avoid introducing unnecessary channels and communication operations. In addition, we seek to optimize performance by distributing communication operations over many tasks and by organizing communication operations in a way that permits concurrent execution.

In domain decomposition problems, communication requirements can be difficult to determine. Recall that this strategy produces tasks by first partitioning data structures into disjoint subsets and then associating with each datum those operations that operate solely on that datum. This part of the design is usually simple. However, some operations that require data from several tasks usually remain. Communication is then required to manage the data transfer necessary for these tasks to proceed. Organizing this communication in an efficient manner can be challenging. Even simple decompositions can have complex communication structures.

In contrast, communication requirements in parallel algorithms obtained by functional decomposition are often straightforward: they correspond to the data flow between tasks. For example, in a climate model broken down by functional decomposition into atmosphere model, ocean model, and so on, the communication requirements will correspond to the interfaces between the component submodels: the atmosphere model will produce values that are used by the ocean model, and so on (Figure 2.3).

In the following discussion, we use a variety of examples to show how communication requirements are identified and how channel structures and communication operations are introduced to satisfy these requirements. For clarity in exposition, we categorize communication patterns along four loosely orthogonal axes: local/global, structured/unstructured, static/dynamic, and synchronous/asynchronous.

- In *local* communication, each task communicates with a small set of other tasks (its ``neighbors''); in contrast, *global* communication requires each task to communicate with many tasks.
- In *structured* communication, a task and its neighbors form a regular structure, such as a tree or grid; in contrast, *unstructured* communication networks may be arbitrary graphs.
- In *static* communication, the identity of communication partners does not change over time; in contrast, the identity of communication partners in *dynamic* communication structures may be determined by data computed at runtime and may be highly variable.
- In *synchronous* communication, producers and consumers execute in a coordinated fashion, with producer/consumer pairs cooperating in data transfer operations; in contrast, *asynchronous* communication may require that a consumer obtain data without the cooperation of the producer.

2.3.1 Local Communication

A local communication structure is obtained when an operation requires data from a small number of other tasks. It is then straightforward to define channels that link the task responsible for performing the operation (the consumer) with the tasks holding the required data (the producers) and to introduce appropriate send and receive operations in the producer and consumer tasks, respectively.

For illustrative purposes, we consider the communication requirements associated with a simple numerical computation, namely a Jacobi finite difference method. In this class of numerical method, a multidimensional grid is repeatedly updated by replacing the value at each point with some function of the values at a small, fixed number of neighboring points. The set of values required to update a single grid point is called that grid point's *stencil*.

2.3.2 Global Communication

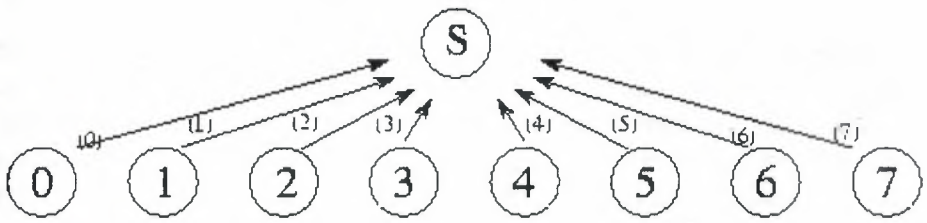

Figure 2.3: *A centralized summation algorithm that uses a central manager task (S) to sum N numbers distributed among N tasks. Here, N=8 , and each of the 8 channels is labeled with the number of the step in which they are used.*

A *global communication* operation is one in which many tasks must participate. When such operations are implemented, it may not be sufficient simply to identify individual producer/consumer pairs. Such an approach may result in too many communications or may restrict opportunities for concurrent execution. For example, consider the problem of performing a *parallel reduction* operation, that is, an operation that reduces *N* values distributed over *N* tasks using a commutative associative operator such as addition.

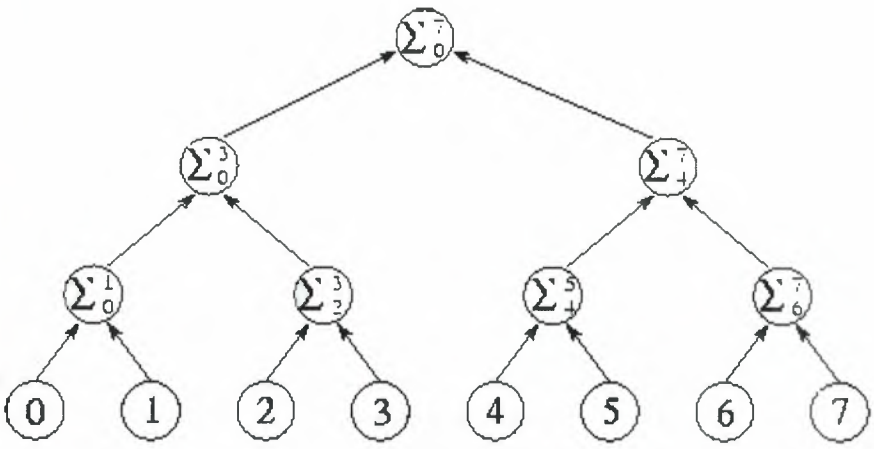

Figure 2.4: *Tree structure for divide-and-conquer summation algorithm with N=8 . The N numbers located in the tasks at the bottom of the diagram are communicated to the tasks in the row immediately above; these each perform an addition and then forward the result to the next level. The complete sum is available at the root of the tree after* **logN** *steps.*

In summary, we observe that in developing an efficient parallel summation algorithm, we have distributed the *N-1* communication and computation operations required to perform the summation and have modified the order in which these operations are performed so that they can proceed concurrently. The result is a regular communication structure in which each task communicates with a small set of neighbors.

2.3.3 Unstructured and Dynamic Communication

Communication patterns may be considerably more complex. For example, in finite element methods used in engineering calculations, the computational grid may be shaped to follow an irregular object or to provide high resolution in critical regions. Here, the channel structure representing the communication partners of each grid point is quite irregular and data-dependent and, furthermore, may change over time if the grid is refined as a simulation evolves.

Unstructured communication patterns do not generally cause conceptual difficulties in the early stages of a design. For example, it is straightforward to define a single task for each vertex in a finite element graph and to require communication for each edge. However, unstructured communication complicates the tasks of agglomeration and mapping. In particular, sophisticated algorithms can be required to determine an agglomeration strategy that both creates tasks of approximately equal size and minimizes communication requirements by creating the least number of intertask edges. If communication requirements are dynamic, these algorithms must be applied frequently during program execution, and the cost of these algorithms must be weighed against their benefits.

2.3.4 Asynchronous Communication

The examples considered in the preceding section have all featured synchronous communication, in which both producers and consumers are aware when communication operations are required, and producers explicitly send data to consumers. In *asynchronous*

communication, tasks that possess data (producers) are not able to determine when other tasks (consumers) may require data; hence, consumers must explicitly request data from producers.

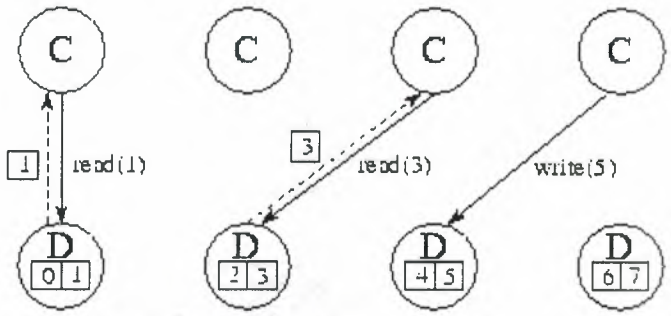

Figure 2.5: *Using separate ``data tasks'' to service read and write requests on a distributed data structure. In this figure, four computation tasks (C) generate read and write requests to eight data items distributed among four data tasks (D). Solid lines represent requests; dashed lines represent replies. One compute task and one data task could be placed on each of four processors so as to distribute computation and data equitably.*

This situation commonly occurs when a computation is structured as a set of tasks that must periodically read and/or write elements of a shared data structure. Let us assume that this data structure is too large or too frequently accessed to be encapsulated in a single task. Hence, a mechanism is needed that allows this data structure to be distributed while supporting asynchronous read and write operations on its components. Possible mechanisms include the following:

1. The data structure is distributed among the computational tasks. Each task both performs computation and generates requests for data located in other tasks. It also periodically interrupts its own computation and *polls* for pending requests.
2. The distributed data structure is encapsulated in a second set of tasks responsible only for responding to read and write requests (Figure 2.5).
3. On a computer that supports a shared-memory programming model, computational tasks can access shared data without any special arrangements. However, care must be taken to ensure that read and write operations on this shared data occur in the proper order.

Each strategy has advantages and disadvantages; in addition, the performance characteristics of each approach vary from machine to machine. The first strategy can result in convoluted, nonmodular programs because of the need to intersperse polling operations throughout application code. In addition, polling can be an expensive operation on some computers, in which case we must trade off the cost of frequent polling against the benefit of rapid response to remote requests. The second strategy is more modular: responsibility for the shared data structure is encapsulated in a separate set of tasks. However, this strategy makes it hard to exploit locality because, strictly speaking, there are no local data: all read and write operations require communication. Also, switching between the computation and data tasks can be expensive on some machines.

2.4 Agglomeration

In the first two stages of the design process, we partitioned the computation to be performed into a set of tasks and introduced communication to provide data required by these tasks. The resulting algorithm is still abstract in the sense that it is not specialized for efficient execution on

any particular parallel computer. In fact, it may be highly inefficient if, for example, it creates many more tasks than there are processors on the target computer and this computer is not designed for efficient execution of small tasks.

In the third stage, *agglomeration,* we move from the abstract toward the concrete. We revisit decisions made in the partitioning and communication phases with a view to obtaining an algorithm that will execute efficiently on some class of parallel computer. In particular, we consider whether it is useful to combine, or *agglomerate*, tasks identified by the partitioning phase, so as to provide a smaller number of tasks, each of greater size (Figure 2.6). We also determine whether it is worthwhile to *replicate* data and/or computation.

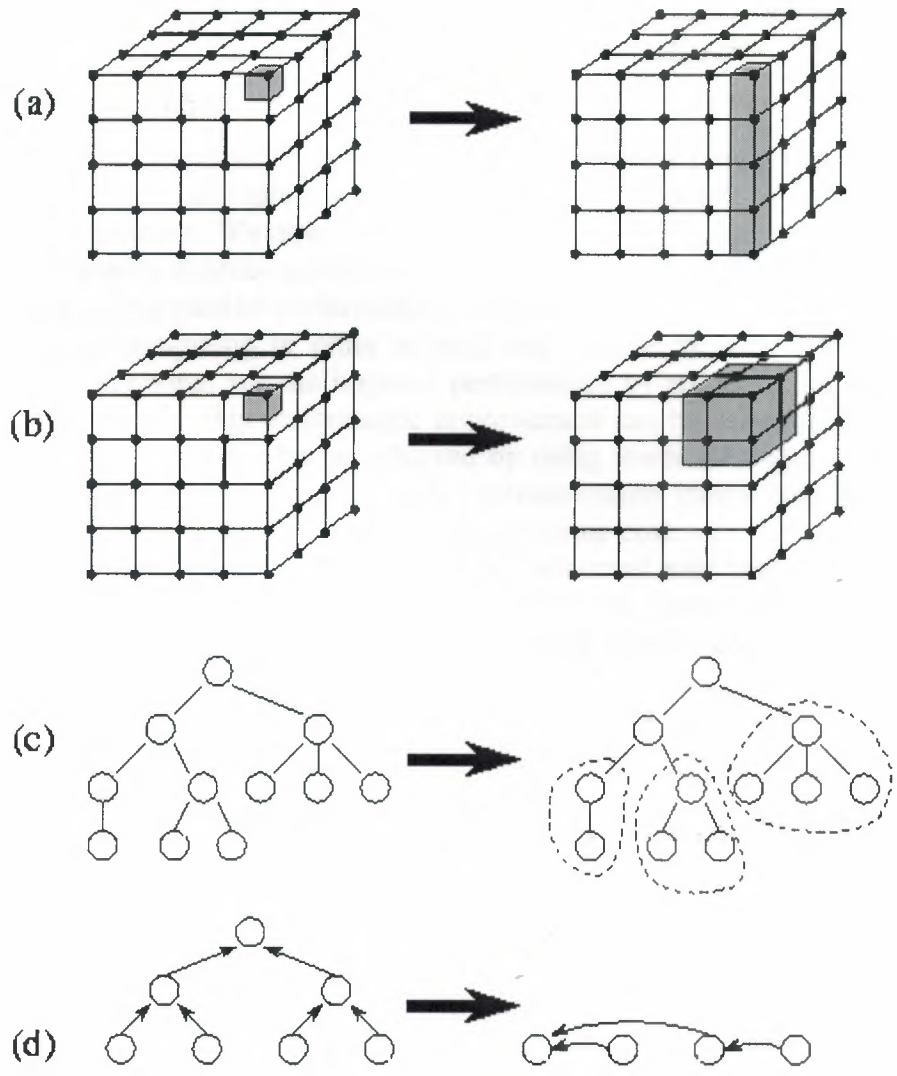

Figure 2.6: *Examples of agglomeration. In (a), the size of tasks is increased by reducing the dimension of the decomposition from three to two. In (b), adjacent tasks are combined to yield a three-dimensional decomposition of higher granularity. In (c), subtrees in a divide-and-conquer structure are coalesced. In (d), nodes in a tree algorithm are combined.*

The number of tasks yielded by the agglomeration phase, although reduced, may still be greater than the number of processors. In this case, our design remains somewhat abstract, since

issues relating to the mapping of tasks to processors remain unresolved. Alternatively, we may choose during the agglomeration phase to reduce the number of tasks to exactly one per processor. We might do this, for example, because our target parallel computer or program development environment demands an SPMD program. In this case, our design is already largely complete, since in defining P tasks that will execute on P processors, we have also addressed the mapping problem. In this section, we focus on general issues that arise when increasing task granularity. Specific issues relating to the generation of SPMD programs are discussed in Section 2.5.

Three sometimes-conflicting goals guide decisions concerning agglomeration and replication: reducing communication costs by increasing computation and communication *granularity*, retaining *flexibility* with respect to scalability and mapping decisions, and reducing *software engineering* costs. These goals are discussed in the next three subsections.

2.4.1 Increasing Granularity

In the partitioning phase of the design process, our efforts are focused on defining as many tasks as possible. This is a useful discipline because it forces us to consider a wide range of opportunities for parallel execution. We note, however, that defining a large number of fine-grained tasks does not necessarily produce an efficient parallel algorithm.

One critical issue influencing parallel performance is communication costs. On most parallel computers, we have to stop computing in order to send and receive messages. Because we typically would rather be computing, we can improve performance by reducing the amount of time spent communicating. Clearly, this performance improvement can be achieved by sending less data. Perhaps less obviously, it can also be achieved by using fewer messages, even if we send the same amount of data. This is because each communication incurs not only a cost proportional to the amount of data transferred but also a fixed startup cost.

In addition to communication costs, we may need to be concerned with task creation costs. For example, the performance of the fine-grained search algorithm illustrated in Figure 1.7, which creates one task for each search tree node, is sensitive to task creation costs.

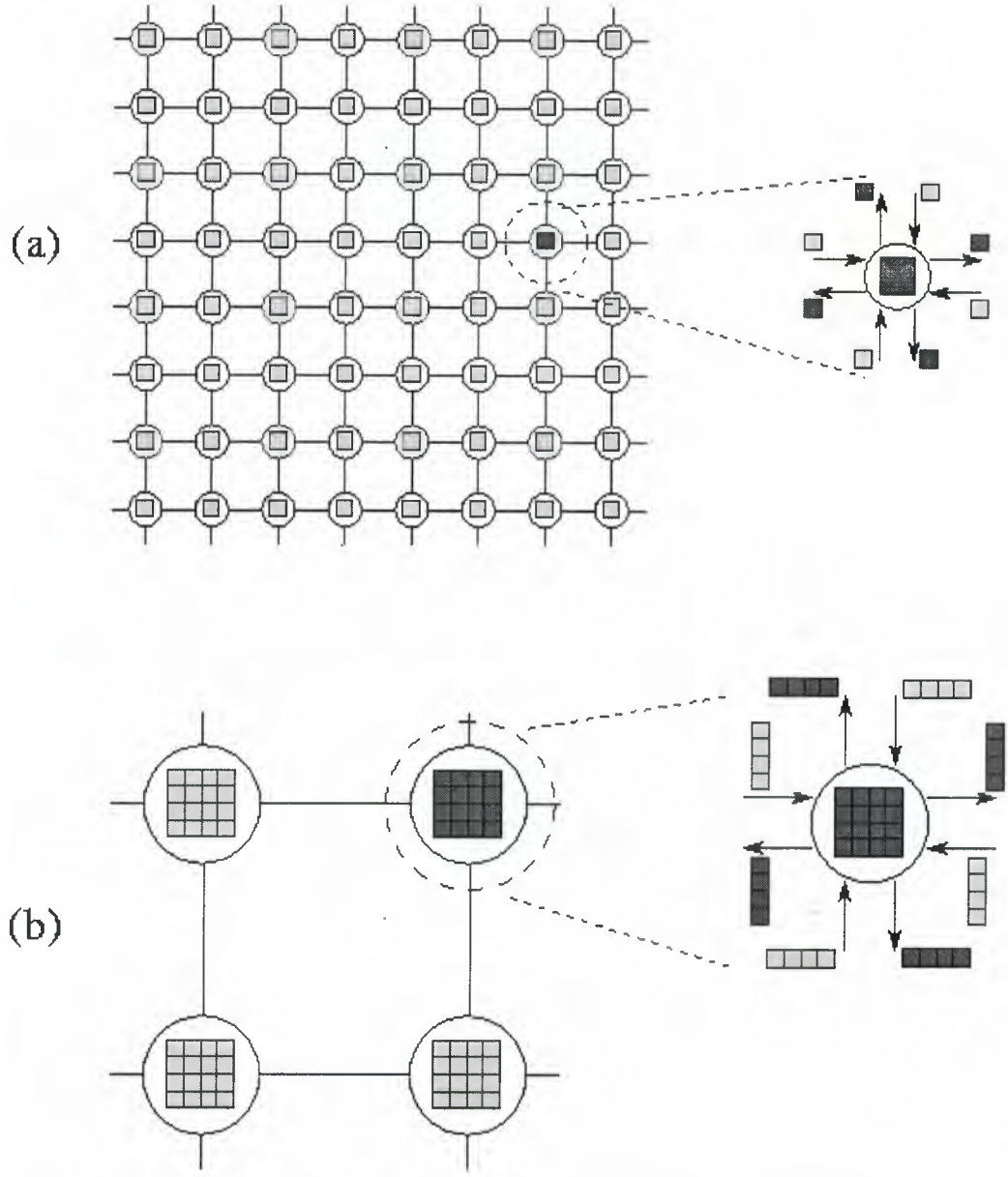

Figure 2.7: *Effect of increased granularity on communication costs in a two-dimensional finite difference problem with a five-point stencil. The figure shows fine- and coarse-grained two-dimensional partitions of this problem. In each case, a single task is exploded to show its outgoing messages (dark shading) and incoming messages (light shading). In (a), a computation on an* **8*8** *grid is partitioned into* **8*8=64** *tasks, each responsible for a single point, while in (b) the same computation is partioned into* **2*2=4** *tasks, each responsible for 16 points. In (a),* **64*4=256** *communications are required, 4 per task; these transfer a total of 256 data values. In (b), only* **4*4=16** *communications are required, and only* **16*4=64** *data values are transferred.*

Surface-to-Volume Effects

If the number of communication partners per task is small, we can often reduce both the number of communication operations and the total communication volume by increasing the granularity of our partition, that is, by agglomerating several tasks into one. This effect is illustrated in Figure 2.7. In this figure, the reduction in communication costs is due to a *surface-to-volume effect* . In other words, the communication requirements of a task are proportional to the surface of the subdomain on which it operates, while the computation requirements are

proportional to the subdomain's volume. In a two-dimensional problem, the ``surface" scales with the problem size while the ``volume" scales as the problem size squared. Hence, the amount of communication performed for a unit of computation (the *communication/computation ratio*) decreases as task size increases. This effect is often visible when a partition is obtained by using domain decomposition techniques.

A consequence of surface-to-volume effects is that higher-dimensional decompositions are typically the most efficient, other things being equal, because they reduce the surface area (communication) required for a given volume (computation). Hence, from the viewpoint of efficiency it is usually best to increase granularity by agglomerating tasks in all dimensions rather than reducing the dimension of the decomposition.

Replicating Computation

We can sometimes trade off replicated computation for reduced communication requirements and/or execution time. For an example, we consider a variant of the summation problem presented in Section 2.3.2, in which the sum must be replicated in each of the *N* tasks that contribute to the sum.

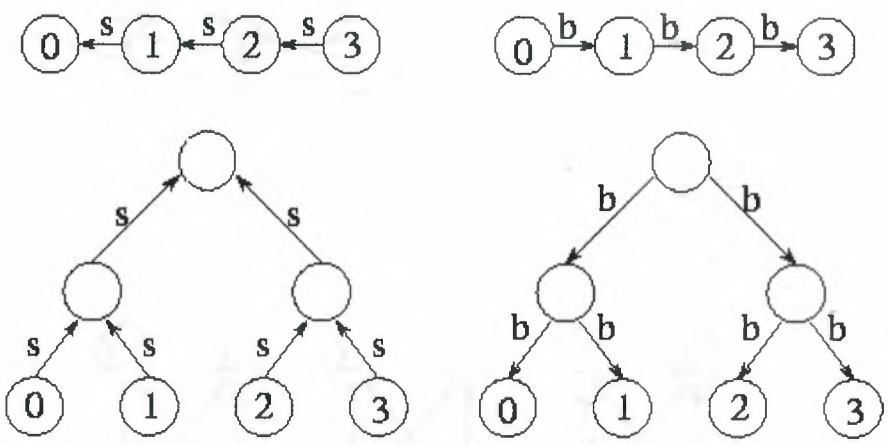

Figure 2.8: *Using an array (above) and a tree (below) to perform a summation and a broadcast. On the left are the communications performed for the summation (s); on the right, the communications performed for the broadcast (b). After* **2(N-1)** *or* **2logN** *steps, respectively, the sum of the* **N** *values is replicated in each of the* **N** *tasks.*

A simple approach to distributing the sum is first to use either a ring- or tree-based algorithm to compute the sum in a single task, and then to *broadcast* the sum to each of the *N* tasks. The broadcast can be performed using the same communication structure as the summation; hence, the complete operation can be performed in either *2(N-1)* or **N** steps, depending on which communication structure is used (Figure 2.8).

These algorithms are optimal in the sense that they do not perform any unnecessary computation or communication. However, there also exist alternative algorithms that execute in less elapsed time, although at the expense of unnecessary (replicated) computation and communication. The basic idea is to perform multiple summations concurrently, with each concurrent summation producing a value in a different task.

We first consider a variant of the array summation algorithm based on this idea. In this variant, tasks are connected in a *ring* rather than an array, and all *N* tasks execute the same algorithm so that *N* partial sums are in motion simultaneously. After *N-1* steps, the complete sum is replicated in every task. This strategy avoids the need for a subsequent broadcast operation, but at the expense of redundant additions **(N-1)^2** and **(N-1)^2** unnecessary communications. However, the summation and broadcast complete in *N-1* rather than *2(N-1)* steps. Hence, the strategy is faster if the processors would otherwise be idle waiting for the result of the summation.

The tree summation algorithm can be modified in a similar way to avoid the need for a separate broadcast. That is, multiple tree summations are performed concurrently so that after **logN** steps each task has a copy of the sum. One might expect this approach to result in **O(N^2)** additions and communications, as in the ring algorithm. However, in this case we can exploit redundancies in both computation and communication to perform the summation in just **O(NlogN)** operations. The resulting communication structure, termed a *butterfly*, is illustrated in Figure 2.9. In each of the **logN** stages, each task receives data from two tasks, performs a single addition, and sends the result of this addition to two tasks in the next stage.

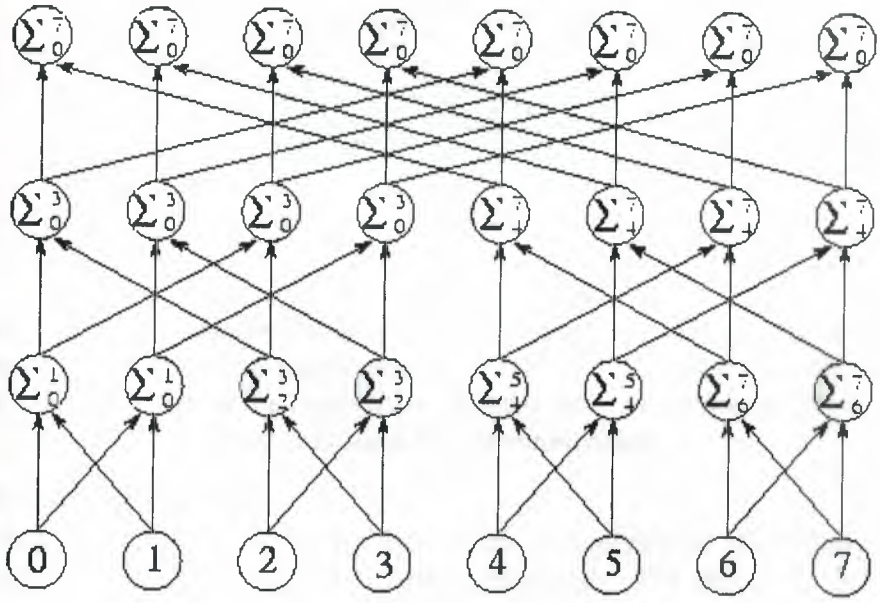

Figure 2.9: *The butterfly communication structure can be used to sum N values in* **logN** *steps. Numbers located in the bottom row of tasks are propagated up through* **logN** *intermediate stages, thereby producing the complete sum in each task in the top row.*

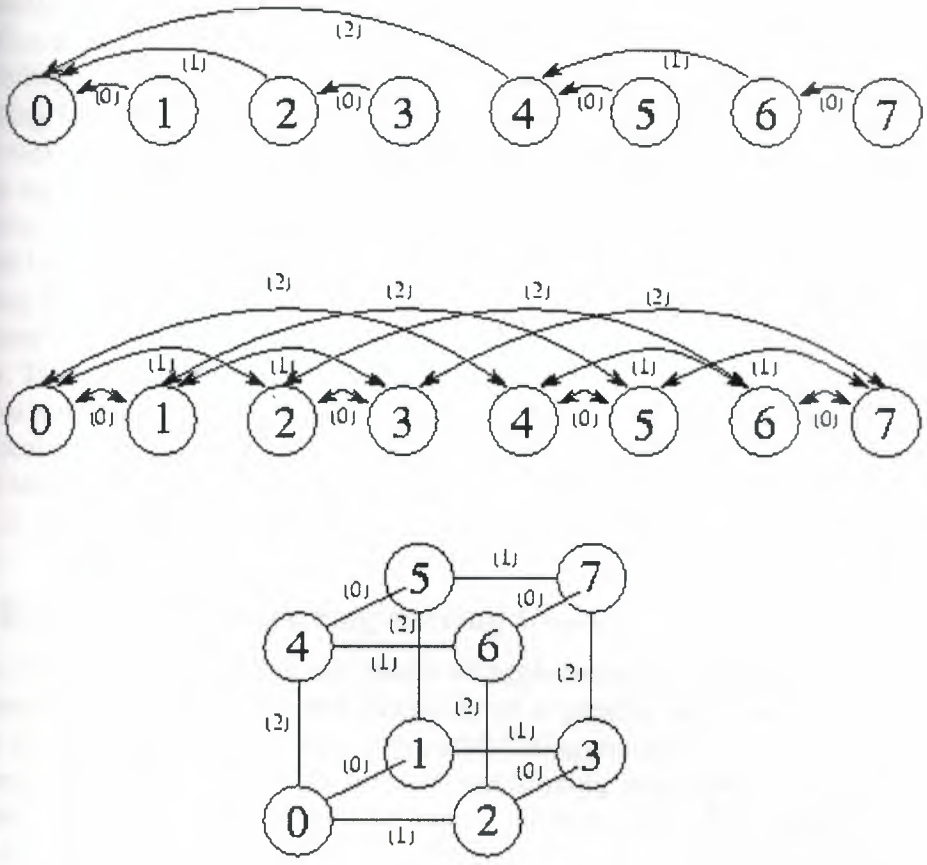

Figure 2.10: *The communication structures that result when tasks at different levels in a tree or butterfly structure are agglomerated. From top to bottom: a tree, a butterfly, and an equivalent representation of the butterfly as a hypercube. In each case, N=8 , and each channel is labeled with the step in which it is used for communication.*

Avoiding Communication

Agglomeration is almost always beneficial if analysis of communication requirements reveals that a set of tasks cannot execute concurrently. For example, consider the tree and butterfly structures illustrated in Figures 2.4 and 2.9. When a single summation problem is performed, only tasks at the same level in the tree or butterfly can execute concurrently. (Notice, however, that if many summations are to be performed, in principle all tasks can be kept busy by pipelining multiple summation operations.) Hence, tasks at different levels can be agglomerated without reducing opportunities for concurrent execution, thereby yielding the communication structures represented in Figure 2.10. The hypercube structure shown in this figure is a fundamental communication structure that has many applications in parallel computing.

2.4.2 Preserving Flexibility

It is easy when agglomerating tasks to make design decisions that limit unnecessarily an algorithm's scalability. For example, we might choose to decompose a multidimensional data structure in just a single dimension, reasoning that this provides more than enough concurrency for the number of processors available. However, this strategy is shortsighted if our program must ultimately be ported to larger parallel computers. It may also lead to a less efficient algorithm, as discussed in Section 2.4.1.

The ability to create a varying number of tasks is critical if a program is to be portable and scalable. Good parallel algorithms are designed to be resilient to changes in processor count. This flexibility can also be useful when tuning a code for a particular computer. If tasks often block waiting for remote data, it can be advantageous to map several tasks to a processor. Then, a blocked task need not result in a processor becoming idle, since another task may be able to execute in its place. In this way, one task's communication is overlapped with another task's computation. This technique, termed *overlapping computation and communication*.

A third benefit of creating more tasks than processors is that doing so provides greater scope for mapping strategies that balance computational load over available processors. As a general rule of thumb, we could require that there be at least an order of magnitude more tasks than processors. This issue is discussed in the next section.

The optimal number of tasks is typically best determined by a combination of analytic modeling and empirical studies. Flexibility does not necessarily require that a design always create a large number of tasks. Granularity can be controlled by a compile-time or runtime parameter. What is important is that a design not incorporate unnecessary limits on the number of tasks that can be created.

2.4.3 Reducing Software Engineering Costs

So far, we have assumed that our choice of agglomeration strategy is determined solely by a desire to improve the efficiency and flexibility of a parallel algorithm. An additional concern, which can be particularly important when parallelizing existing sequential codes, is the relative development costs associated with different partitioning strategies. From this perspective, the most interesting strategies may be those that avoid extensive code changes. For example, in a code that operates on a multidimensional grid, it may be advantageous to avoid partitioning altogether in one dimension, if doing so allows existing routines to be reused unchanged in a parallel program.

Frequently, we are concerned with designing a parallel algorithm that must execute as part of a larger system. In this case, another software engineering issue that must be considered is the data distributions utilized by other program components. For example, the best algorithm for some program component may require that an input array data structure be decomposed in three dimensions, while a preceding phase of the computation generates a two-dimensional decomposition. Either one or both algorithms must be changed, or an explicit restructuring phase must be incorporated in the computation. Each approach has different performance characteristics.

2.4.4 Agglomeration Design Checklist

We have now revised the partitioning and communication decisions developed in the first two design stages by agglomerating tasks and communication operations. We may have agglomerated tasks because analysis of communication requirements shows that the original partition created tasks that cannot execute concurrently. Alternatively, we may have used agglomeration to increase computation and communication granularity and/or to decrease software engineering costs, even though opportunities for concurrent execution are reduced. At this stage, we evaluate our design with respect to the following checklist.

1. Has agglomeration reduced communication costs by increasing locality? If not, examine your algorithm to determine whether this could be achieved using an alternative agglomeration strategy.
2. If agglomeration has replicated computation, have you verified that the benefits of this replication outweigh its costs, for a range of problem sizes and processor counts?

3. If agglomeration replicates data, have you verified that this does not compromise the scalability of your algorithm by restricting the range of problem sizes or processor counts that it can address?
4. Has agglomeration yielded tasks with similar computation and communication costs? The larger the tasks created by agglomeration, the more important it is that they have similar costs. If we have created just one task per processor, then these tasks should have nearly identical costs.
5. Does the number of tasks still scale with problem size? If not, then your algorithm is no longer able to solve larger problems on larger parallel computers.
6. If agglomeration eliminated opportunities for concurrent execution, have you verified that there is sufficient concurrency for current and future target computers? An algorithm with insufficient concurrency may still be the most efficient, if other algorithms have excessive communication costs; performance models can be used to quantify these tradeoffs.
7. Can the number of tasks be reduced still further, without introducing load imbalances, increasing software engineering costs, or reducing scalability? Other things being equal, algorithms that create fewer larger-grained tasks are often simpler and more efficient than those that create many fine-grained tasks.
8. If you are parallelizing an existing sequential program, have you considered the cost of the modifications required to the sequential code? If these costs are high, consider alternative agglomeration strategies that increase opportunities for code reuse. If the resulting algorithms are less efficient, use performance modeling techniques to estimate cost tradeoffs.

2.5 Mapping

In the fourth and final stage of the parallel algorithm design process, we specify where each task is to execute. This mapping problem does not arise on uniprocessors or on shared-memory computers that provide automatic task scheduling. In these computers, a set of tasks and associated communication requirements is a sufficient specification for a parallel algorithm; operating system or hardware mechanisms can be relied upon to schedule executable tasks to available processors. Unfortunately, general-purpose mapping mechanisms have yet to be developed for scalable parallel computers. In general, mapping remains a difficult problem that must be explicitly addressed when designing parallel algorithms.

Our goal in developing mapping algorithms is normally to minimize total execution time. We use two strategies to achieve this goal:

1. We place tasks that are able to execute concurrently on *different* processors, so as to enhance concurrency.
2. We place tasks that communicate frequently on the *same* processor, so as to increase locality.

Clearly, these two strategies will sometimes conflict, in which case our design will involve tradeoffs. In addition, resource limitations may restrict the number of tasks that can be placed on a single processor.

The mapping problem is known to be *NP -complete*, meaning that no computationally tractable (polynomial-time) algorithm can exist for evaluating these tradeoffs in the general case. However, considerable knowledge has been gained on specialized strategies and heuristics and

classes of problem for which they are effective. In this section, we provide a rough classification of problems and present some representative techniques.

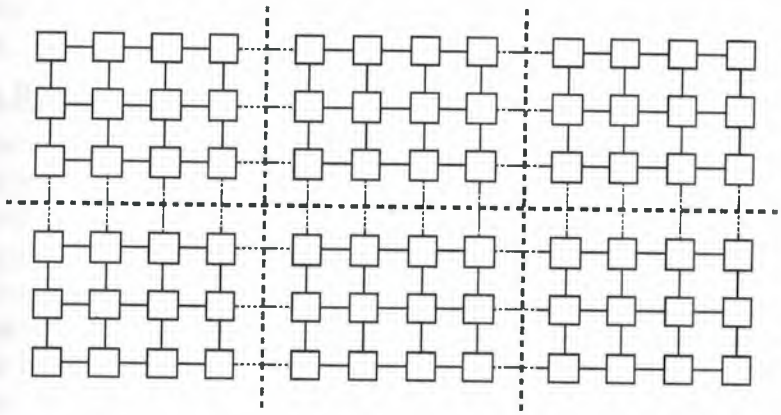

Figure 2.11: *Mapping in a grid problem in which each task performs the same amount of computation and communicates only with its four neighbors. The heavy dashed lines delineate processor boundaries. The grid and associated computation is partitioned to give each processor the same amount of computation and to minimize off-processor communication.*

Many algorithms developed using domain decomposition techniques feature a fixed number of equal-sized tasks and structured local and global communication. In such cases, an efficient mapping is straightforward. We map tasks in a way that minimizes interprocessor communication (Figure 2.10); we may also choose to agglomerate tasks mapped to the same processor, if this has not already been done, to yield a total of P coarse-grained tasks, one per processor.

In more complex domain decomposition-based algorithms with variable amounts of work per task and/or unstructured communication patterns, efficient agglomeration and mapping strategies may not be obvious to the programmer. Hence, we may employ *load balancing* algorithms that seek to identify efficient agglomeration and mapping strategies, typically by using heuristic techniques. The time required to execute these algorithms must be weighed against the benefits of reduced execution time. *Probabilistic load-balancing* methods tend to have lower overhead than do methods that exploit structure in an application.

The most complex problems are those in which either the number of tasks or the amount of computation or communication per task changes dynamically during program execution. In the case of problems developed using domain decomposition techniques, we may use a *dynamic load-balancing* strategy in which a load-balancing algorithm is executed periodically to determine a new agglomeration and mapping. Because load balancing must be performed many times during program execution, *local* algorithms may be preferred that do not require global knowledge of computation state.

Algorithms based on functional decomposition often yield computations consisting of many short-lived tasks that coordinate with other tasks only at the start and end of execution. In this case, we can use *task-scheduling* algorithms, which allocate tasks to processors that are idle or that are likely to become idle.

2.5.1. Load-Balancing Algorithms

A wide variety of both general-purpose and application-specific load-balancing techniques have been proposed for use in parallel algorithms based on domain decomposition techniques. We review several representative approaches here, namely recursive bisection methods, local

algorithms, probabilistic methods, and cyclic mappings. These techniques are all intended to agglomerate fine-grained tasks defined in an initial partition to yield one coarse-grained task per processor. Alternatively, we can think of them as partitioning our computational domain to yield one subdomain for each processor. For this reason, they are often referred to as *partitioning* algorithms.

Recursive Bisection

Recursive bisection techniques are used to partition a domain (e.g., a finite element grid) into subdomains of approximately equal computational cost while attempting to minimize communication costs, that is, the number of channels crossing task boundaries. A divide-and-conquer approach is taken. The domain is first cut in one dimension to yield two subdomains. Cuts are then made recursively in the new subdomains until we have as many subdomains as we require tasks. Notice that this recursive strategy allows the partitioning algorithm itself to be executed in parallel.

The most straightforward of the recursive bisection techniques is *recursive coordinate bisection*, which is normally applied to irregular grids that have a mostly local communication structure. This technique makes cuts based on the physical coordinates of grid points in the domain, at each step subdividing along the longer dimension so that if (for example) the cut is made along the x dimension, grid points in one subdomain will all have an x -coordinate greater than grid points in the other. This approach has the advantages of being simple and inexpensive. It also does a good job of partitioning computation. A disadvantage is that it does not optimize communication performance. In particular, it can generate long, skinny subdomains, which if an algorithm has significant local communication will result in more messages than will a decomposition that generates square subdomains.

A variant of recursive bisection called *unbalanced recursive bisection* attempts to reduce communication costs by forming subgrids that have better aspect ratios. Instead of automatically dividing a grid in half, it considers the *P-1* partitions obtained by forming unbalanced subgrids with *1/P* and *(P-1)/P* of the load, with *2/P* and *(P-2)/P* of the load, and so on, and chooses the partition that minimizes partition aspect ratio. This method increases the cost of computing the partition but can reduce communication costs.

Another technique, called *recursive graph bisection* , can be useful in the case of more complex unstructured grids, for example, finite element meshes. This technique uses connectivity information to reduce the number of grid edges crossing subdomain boundaries, and hence to reduce communication requirements. A grid is treated as a graph with N vertices (grid points) . The algorithm first identifies the two extremities of the graph, that is, the two vertices that are the most separated in terms of graph distance. (The graph distance between two vertices is the smallest number of edges that must be traversed to go between them.) Each vertex is then assigned to the subdomain corresponding to the closer extremity. Another algorithm called *recursive spectral bisection* is even better in many circumstances.

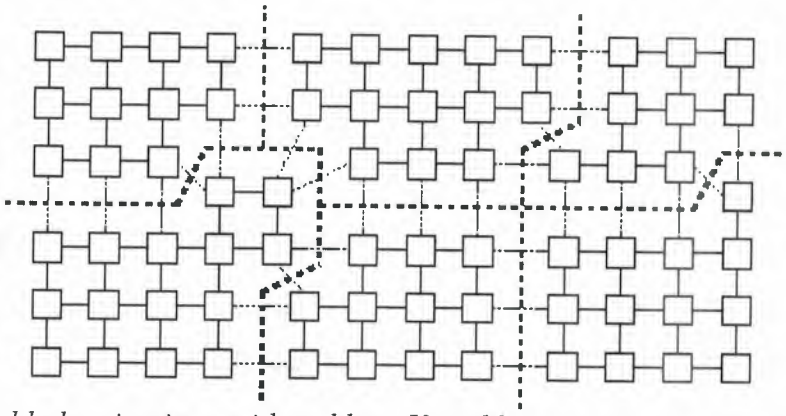

Figure 2.12: *Load balancing in a grid problem. Variable numbers of grid points are placed on each processor so as to compensate for load imbalances. This sort of load distribution may arise if a local load-balancing scheme is used in which tasks exchange load information with neighbors and transfer grid points when load imbalances are detected.*

Local Algorithms

The techniques just described are relatively expensive because they require global knowledge of computation state. In contrast, local load-balancing algorithms compensate for changes in computational load using only information obtained from a small number of neighboring processors. For example, processors may be organized in a logical mesh; periodically, each processor compares its computational load with that of its neighbors in the mesh and transfers computation if the difference in load exceeds some threshold. Figure 2.12 shows load distributions produced by such schemes.

Because local algorithms are inexpensive to operate, they can be useful in situations in which load is constantly changing. However, they are typically less good at balancing load than global algorithms and, in particular, can be slow to adjust to major changes in load characteristics. For example, if a high load suddenly appears on one processor, multiple local load-balancing operations are required before load ``diffuses'' to other processors.

Probabilistic Methods

A particularly simple approach to load balancing is to allocate tasks to randomly selected processors. If the number of tasks is large, we can expect that each processor will be allocated about the same amount of computation. Advantages of this strategy are its low cost and scalability. Disadvantages are that off-processor communication is required for virtually every task and that acceptable load distribution is achieved only if there are many more tasks than there are processors. The strategy tends to be most effective when there is relatively little communication between tasks and/or little locality in communication patterns. In other cases, probabilistic methods tend to result in considerably more communication than do other techniques.

Cyclic Mappings

If we know both that computational load per grid point varies and that there is significant spatial locality in load levels, then a *cyclic* (or *scattered*, as it is sometimes called) mapping of tasks to processors can be appropriate. That is, each of P processors is allocated every P th task according to some enumeration of the tasks (Figure 1.7). This technique is a form of probabilistic mapping. The goal is that, on average, each processor will be allocated about the same computational load. The benefits of improved load balance may need to be weighed against increased communication costs due to reduced locality. Block cyclic distributions are also possible, in which blocks of tasks are allocated to processors.

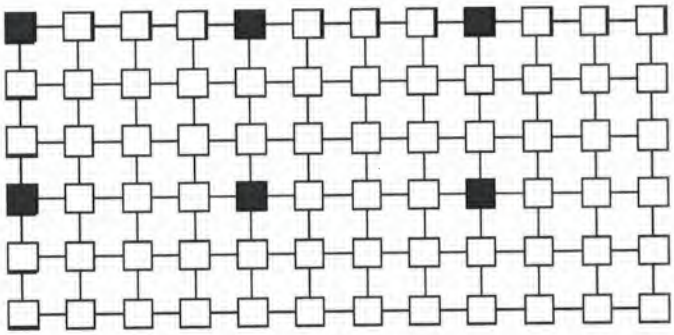

Figure 2.13: *Using a cyclic mapping for load balancing in a grid problem, when executing on 12 processors. Tasks mapped to a single processor are shaded. Notice that with this mapping, all communications are with tasks located on different processors (assuming a five-point stencil).*

2.5.2 Task-Scheduling Algorithms

Task-scheduling algorithms can be used when a functional decomposition yields many tasks, each with weak locality requirements. A centralized or distributed task pool is maintained, into which new tasks are placed and from which tasks are taken for allocation to processors. In effect, we reformulate the parallel algorithm so that what were originally conceived of as tasks become data structures representing ``problems," to be solved by a set of worker tasks, typically one per processor.

The most critical (and complicated) aspect of a task-scheduling algorithm is the strategy used to allocate problems to workers. Generally, the chosen strategy will represent a compromise between the conflicting requirements for independent operation (to reduce communication costs) and global knowledge of computation state (to improve load balance). We discuss manager/worker, hierarchical manager/worker, and decentralized approaches.

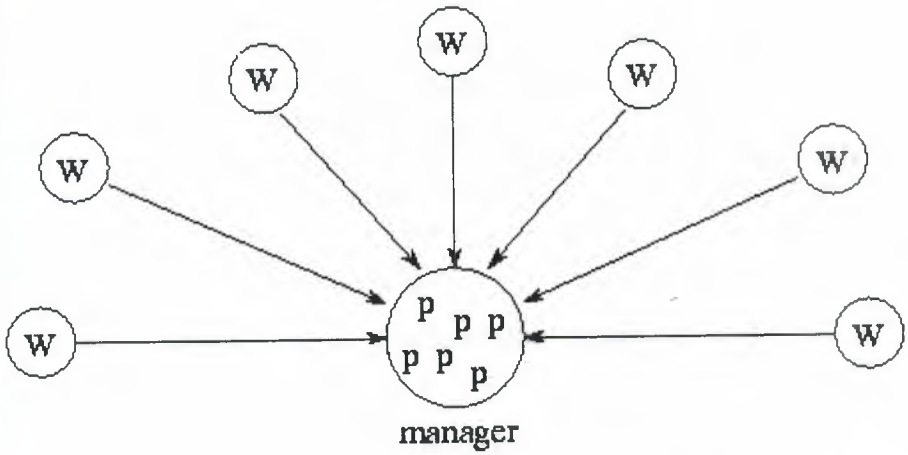

Figure 2.14: *Manager/worker load-balancing structure. Workers repeatedly request and process problem descriptions; the manager maintains a pool of problem descriptions (p) and responds to requests from workers.*

Manager/Worker

Figure 2.14 illustrates a particularly simple task scheduling scheme that is nevertheless effective for moderate numbers of processors. Central manager task is given responsibility for problem allocation. Each worker repeatedly requests and executes a problem from the manager.

2. If considering a design based on dynamic task creation and deletion, have you also considered an SPMD algorithm? An SPMD algorithm provides greater control over the scheduling of communication and computation, but can be more complex.
3. If using a centralized load-balancing scheme, have you verified that the manager will not become a bottleneck? You may be able to reduce communication costs in these schemes by passing pointers to tasks, rather than the tasks themselves, to the manager.
4. If using a dynamic load-balancing scheme, have you evaluated the relative costs of different strategies? Be sure to include the implementation costs in your analysis. Probabilistic or cyclic mapping schemes are simple and should always be considered, because they can avoid the need for repeated load-balancing operations.
5. If using probabilistic or cyclic methods, do you have a large enough number of tasks to ensure reasonable load balance? Typically, at least ten times as many tasks as processors are required.

We have now completed the design of one or more parallel algorithms designs for our problem. However, we are not quite ready to start writing code: several phases in the design process remain. First, we need to conduct some simple performance analyses in order to choose between alternative algorithms and to verify that our design meets performance goals. We should also think hard about the implementation costs of our designs, about opportunities for reusing existing code in their implementation, and about how algorithms fit into larger systems of which they may form a part.

CHAPTER3

Parallel Branch-and-Bound

3.1. General Overview

The Branch-and-Bound paradigm is a general purpose enumerative technique for solving a wide range of problems in Combinatorial Optimisation, Operations Research and Artificial Intelligence. We shall introduce the Branch-and-Bound technique in general terms followed by a more formal definition using the 0/1 Knapsack problem as an example. Although the solution of this particular problem is not an especially interesting example of a Branch-and-Bound algorithm, it does provide a simple example for illustrating many of the important aspects of the Branch-and-Bound paradigm.

Branch-and-Bound would appear to be well suited to parallel processing because packets of work which could be executed independently on different processors are generated during a Branch-and-Bound algorithm. This is very encouraging as many algorithms of this type involve a considerable number of packets of work and the use of parallel processing could reduce execution times dramatically. However, the design of parallel Branch-and-Bound algorithms is not necessarily straightforward, and we shall discuss several possible strategies and show how these have been implemented on various parallel machines.

3.2. An Informal Description

Informally, the Branch-and-Bound approach can be characterised as an intelligent search for an optimal solution within a space of potential solutions to the given problem. Typically, the search space is exponentially large with respect to the size of the problem and the aim of the technique is to find the optimal solution whilst minimising the number of solutions to be explicitly considered.

During the execution of a Branch-and-Bound algorithm the search space is partitioned recursively into smaller and smaller disjoint subsets, a process known as *expansion.* Each subset generated then requires a value to be computed, corresponding to a bound on the best possible value for any solution which may be found in that section of the search space. After each iteration of the partitioning process, all subsets with a bound not better than the value for a known solution are identified and are excluded from any further part in the algorithm.

The partitioning process continues until all possible subsets have either been expanded or have been specifically excluded from further consideration. At this point, the current best solution must be at least as good as any other solution for the given problem and the algorithm terminates.

In the implementation of Branch-and-Bound algorithms the subsets of the search space are represented as *problem-states* which are generated and stored in an *active pool* of states. Associated with each problem-state is a set of potential solutions to the initial problem. The problem-states in the active pool correspond to sections of the search space which are still to be examined. Initially the active pool contains a single element, the initial problem description, which represents the whole of the search space. On termination of the algorithm the active pool is empty, indicating that the whole of the search space has either been examined or has been excluded from consideration.

The procedure described above can be characterised by a number of rules that define how the search is performed and how the search space is partitioned. These rules are known as the selection rule, the expansion rule, the branching rule and the bounding rules and are described informally as follows.

The *selection rule* is used to choose a problem-state from the active pool. After being selected the problem-state is removed from the pool and is added to the *dead pool,* consisting of problems that have been both generated and expanded.

The *expansion rule* proceeds as follows. The selected problem-state is examined to see if it provides a single feasible solution to the initial problem. If it does then the solution is considered as a possible optimal solution to the initial problem. The value generated is compared to the current best solution and the solution with the better value is preserved. If the problem-state does not provide a single solution, then it may be possible to show that it cannot possibly provide any feasible solution. In this case the problem-state is infeasible and is discarded. Otherwise, we expand the problem-state using *a branching rule* to split the problem into a number of sub-problem-states which are then added to the active pool. 'The newly generated problem-states represent a partitioning of the search space of the parent problem-state.

Bounding rules are used to eliminate problems which cannot lead to an optimal solution. All Branch-and-Bound algorithms can thus be characterised by their expansion, branching, bounding and selection rules. We now present a more formal definition of Branch-and-Bound, using the 0/1 Knapsack problem as an illustrative example.

3.3 The 0/1 Knapsack Problem

In the 0/1 Knapsack problem, the objective is to maximise the possible profit which can be obtained from putting a number of items into a knapsack of known capacity. The constraints on this problem are that only whole items may be added to the knapsack and the maximum capacity of the knapsack must not be exceeded.

An instance of the 0/1 Knapsack problem consists of n items. Associated with the i^th item is a volume, v;, and a profit, p;. Profit, p; accrues if the i^th item is included in the knapsack. For a knapsack of capacity C the problem can be formulated as follows:

$$\text{Maximise} \quad \sum_{i=1}^{n} p_i x_i$$

$$\text{subject to} \quad \sum_{i=1}^{n} v_i x_i <= \mathbf{C},$$

$$1 <= i <= n.$$

3.3.1 An Example

We now show how Branch-and-Bound might be used to solve the 0/1 Knapsack problem by considering a simple example. In this example the knapsack has a capacity of 100 and there are 4 items with the following volumes and profits.

Item	Volume	Profit
1	50	60
2	40	46
3	40	45
4	20	20

The Branch-and-Bound algorithm produces a binary search tree based on a decision as to whether or not an item should be included or excluded from the knapsack.

The algorithm begins with an empty knapsack, node 1. This cannot be solved immediately so it is expanded to generate nodes 2 and 7 corresponding to the inclusion and exclusion, respectively, of item 1. The bounds on these can now be calculated using a greedy process which adds items until all of the available space is used; this process is allowed to add partial items to the knapsack. By sorting the items in descending order of profit/volume ratio, this bound provides an upper bound on the possible profit.

Node 2 has a current profit of 60 and still has 50 units of remaining space. This can hold all of item 2 and one quarter of item 3. This therefore has a bound of [60 + 46 + ((1/ 4) * 45)] = 117.

Node 7 has a current profit of 0 and 100 units of remaining space. It can therefore hold items 2, 3 and 4 and has a bound of [46 + 45 + 20] = 111.

We use a best-first selection rule, always selecting from the active pool that problem with the greatest bound. Thus node 2 is selected and expanded generating nodes 3 and 5 corresponding to the inclusion and exclusion, respectively, of item 2. Nodes 3 and 4 are expanded in turn but only generate 1 child apiece since, in each case, the child that includes the item exceeds the knapsack capacity.

The child generated by node 4 is actually a solution to the problem, but this fact cannot be established until the node is chosen for expansion. In this particular example it is never chosen, but is later pruned when a better solution is discovered. Nodes 5-9 are expanded in similar fashion.

When we come to select the next node, there is no node in the active pool with a bound better than the current best and so the algorithm terminates. Our optimal solution includes items 2, 3 and 4 and has a value of 111.

3.4 Parallel Branch-and-Bound

We begin by discussing the attractions of executing Branch-and-Bound algorithms in parallel.

3.4.1 Reasons for Parallel Branch-and-Bound

Branch-and-Bound algorithms are often used for solving NP-hard problems with very large search spaces. The time to solve such problems is normally proportional to the number of problem-states in the search tree. Problems with large search spaces therefore often place large demands on resources. A parallel implementation could allow these programs to be run over a number of machines and therefore allow them to complete more quickly.

There has been much research in the past on the possibility of using approximate Branch-and-Bound to reduce the execution time of this form of algorithm. By using a parallel machine it may be possible to generate an optimal solution in the same time as the sequential machine takes to generate an approximate solution. Even if it is not possible to generate an exact solution in reasonable time, the parallel machine should, hopefully, be able to generate a much closer approximation to the true optimal solution.

The use of parallel processing may allow larger problems to be solved than would be possible on a sequential machine. As many Branch-and-Bound problems are NP-hard the size of problem which can be solved in `reasonable time' is most unlikely to grow linearly with the number of processors but some increase in problem size should be possible.

3.4.2 Implications of Parallel Branch-and-Bound

We begin by making a slight change to the classical, sequential, view of Branch- and-Bound. The classical definition of Branch-and-Bound requires that the state space search generates all of the children of a problem before any other active node can become the selected node. This is obviously a requirement imposed on any sequential implementation and it is inappropriate for a parallel one. Indeed, there is no reason, in a parallel system, why a child problem cannot be selected for expansion on one processor while its parent is still being expanded on another processor.

Another point to consider is that a parallel Branch-and-Bound algorithm might take longer to execute than the equivalent sequential algorithm. This result has been known for some time and is known as a `detrimental anomaly'. Similarly, it is possible for a parallel algorithm to have super-linear speed up over the sequential one, this being referred to as a `speed up anomaly'.

The expansion of a problem-state in a Branch-and-Bound algorithm might involve the generation and manipulation of a complicated data structure such as a cost matrix. It is, however, often much easier to generate this structure using the data from an ancestor problem if such data can be made available. If the parallel machine has more memory than the sequential one then it may be possible to store more data and therefore reduce the amount of effort spent recreating the

data from scratch. As many parallel machines are created from standard sequential parts, it is often the case that a machine with *m* processors will have *m* times as much memory as one with a single processor and then efficiency anomalies can occur. Because of these anomalies it is possible for an m processor parallel machine to perform more problem expansions than a single processor one, but to still get greater than *m* fold speed up.

3.4.3 Parallelisation of Branch-and-Bound Algorithms

There are a number of ways in which a Branch-and-Bound algorithm may be made to run on a parallel machine. We now present a number of possible sources of parallelism and discuss their advantages and disadvantages. For this discussion, we assume that an *m* processor parallel machine is available for solving the problem.

· *Parallel expansion of the search tree with different initial bound values.* If an upper and lower bound, Ub and Lb, are known for the problem then the processors can begin with different initial bound values in this range. Each processor is assigned an initial upper and lower bound, *(Lb, Ub,;).* Suitable initial values for processor i might be:

(Lb, L6+ i* (Ub - Lb)/m).

Processors that have their initial bound set too low will finish quickly without finding a solution and could then be restarted with a new value in a higher range. When a feasible solution is found its value is passed to all processors and any processor whose work becomes bounded chooses a new range of values. The whole process finishes when one processor terminates with an optimal solution. This method of introducing parallelism should have very low communication costs but has the obvious problem that much of the work is performed many times. This is a fundamental problem with the algorithm as the initial problem-states will be expanded by every single processor. 'This method has the additional problem that it is not applicable for algorithms using the best-first search strategy. Using this strategy, no processor would terminate with an optimal solution before the processor running the algorithm with bounds *[Lb, Ub]* and any processor obtaining an optimal solution would have performed exactly the same set of expansions.

Although this method does not look particularly promising it may prove effective in a small number of cases when the lower bound happens (by chance) to be close to the optimal value. In this case, the processor assigned this initial range will prune large sections of the search tree and will quickly find an optimal solution.

Similar results to those described could be obtained by using an approximate Branch-and-Bound algorithm and assigning different tolerances to the different processors. A processor searching with a large tolerance would finish relatively quickly and could start again with a lower tolerance value. Its first attempt would, however, provide reasonably tight bounds on the value of the optimal solution and these could be used by all of the processors to prune problem-states. The process terminates when a processor finishes with a tolerance of 0%.

· *Parallel search using different algorithms.* Many optimisation problems can be solved in a number of different ways using different algorithms with different bound and priority functions. By running different algorithms on different processors it may be possible to complete the search more quickly. A possible implementation could have One processor performing a depth-first search while another used best-first search.

Once again, the same tree is being expanded by more than one processor so some nodes will be expanded unnecessarily. As different search strategies are used, however, the overlap in the areas searched should not be too great. Although the speed up of this system may not be very great, it may have some beneficial effects on memory usage. The generation of the bounds by the processor using the depth-first strategy may allow another processor using a best-first search to prune large sections of the search tree. The use of the best-first search strategy may otherwise have been impossible due to its large memory requirements.

· *Perform the expansion of a single node in parallel.* The amount of effort required to expand a single node may be very significant and it may often be possible to split this work between a

number of processors. For example, consider a Branch-and-Bound algorithm where each problem- state generates *m* children. Each of these children requires a bound value and a priority value to be computed. In many such algorithms these values could be generated in parallel on m different processors. If the generation of the bound and priority values forms a considerable part of the execution time then this could lead to a reasonable speed up.

This method may prove effective for a number of problems and has the advantage that it avoids the problems of detrimental speed up anomalies. Since the same search tree will be expanded, irrespective of the number of processors, such anomalies will never occur. Unfortunately, the ratio of communication to calculation is likely to be fairly high using this technique as data must be sent to processors every time a new child is generated.

An example where this could be used is in the solution of Integer Programming problems. One method of solving these problems requires an LP relaxation to be performed in the expansion of every problem. The solution of an LP relaxation can involve considerable effort and it may be possible to do this in parallel. There has been much interest in interior point methods for solving the LP component of these problems. These methods have the potential for parallelism and may prove a particularly effective method for solving this type of problem.

The main disadvantage of this form of parallelisation is that it is likely to be very problem dependent and cannot be done in a general way. This approach may, however, prove effective for certain classes of Branch- and-Bound algorithms and the fact that its performance is predictable may make it suitable for applications where predictable performance is more important than average absolute performance.

· *Parallel evaluation of subproblems.* As the search tree is expanded there is a pool of active problem-states which could be expanded in parallel. As problem-states are independent of each other they can be expanded in parallel on different processors. This appears to be the obvious source of parallelism with a shared memory multi-processor but with a message passing machine it may be difficult to follow the priority function accurately without large communication overheads.

There are many ways in which this type of parallelism could be exploited, different approaches involving different tradeoffs between the amount of communication necessary and the average number of problem-states expanded. Possible methods of parallelisation include:

1. Static distribution of the search tree. One strategy which would lead to very low communication costs would be to split the search tree statically between the available processors. Incumbent values could be broadcast to allow pruning of the search tree and termination of the algorithm occurs when all processors have completed the search of their section of the tree.

The disadvantage of this method is that the amount of workrequired to expand a section of the search tree is not normally known in advance so it is not possible to divide the work evenly between the processors. In general, if the amount of work required to expand a section of the tree is known in advance then it is likely that the work need not be done at all!

This method may prove effective when a complete expansion of the search tree is required and where no bounding takes place. A possible use of this technique is therefore to verify that a known solution is optimal. In this case the value of the proposed solution is already known and the task is to expand the whole of the remaining tree to verify that no better solution exists. Although the actual shape of the final tree may not be known, it may be possible to estimate its shape and to use this estimate when dividing the work between the processors.

2. Dynamic distribution with farming of available work. Dynamic distribution of work allows work to be distributed more evenly between the available processors. Unfortunately, this also means higher communication costs. Although communication costs can be high with a. message passing computer, many of the current parallel Branch-and-Bound kernels use an approach based on this principle . On a shared memory machine this approach seems particularly suitable though memory contention for accessing the pool of work can prove problematical.

3. Dynamic distribution with farming of large tasks. Rost and Maehle (1989) have noted the large communication overheads present in the farming technique and have come up with a similar scheme where larger tasks are farmed between the processors.

The initial problem is first split into a number of sub-problems by running the algorithm sequentially on one processor using a breadth-first search. This processor continues until M problem-states have been generated where $M \gg m$. The problem-states in the active pool now represent large sections of the search space. A farming technique is then used to spread the work over the available processors. Each processor is given one of these problems to expand and does not communicate until it generates a new incumbent or it has expanded the whole of the search tree represented by the problem-state. When the whole sub-tree has been expanded a message is sent to the master processor requesting another piece of work.

Two main disadvantages of this method have been noted. First, the initial breadth-first search is done on a single processor while all other processors are idle. Rost & Maehle do not consider this to be a major problem as the initial brendth-first search is assumed to be a very small part of the total search.

The second potential disadvantage is that, when the pool of problem-states on the master processor is, exhausted, processors finishing the work allocated to them have to wait until all of the other processors have finished before the program can terminate. As the sub-trees are specifically chosen to involve a large computation component (in order to reduce communication overheads) this could lead to a considerable time when only some of the processors are busy.

4. *Dynamic distribution with local priority scheme.* In order to reduce the communication costs inherent in the farming method it is possible to store the newly generated child problems in the memory of the processor that generated them and to only follow the priority scheme locally. This implies that there is no global pool of active problem-states so care must be taken to ensure that all processors have work to do and to ensure that the priority function is adhered to. These factors can prove difficult to overcome but much work has been done in this area and many of the currently available kernels use techniques based on this principle.

5. *Randomised algorithms.* The two methods described above perform extra work to ensure that the same problem-state is not expanded on two different processors. Randomised algorithms accept that some work may be done more than once but use a random choice of which problem to expand to minimise the chance of two processors choosing the same problem. As well as allowing work to be done several times, this method also tends to ignore the priority function and may therefore expand work which has very little chance of generating a new incumbent. Nevertheless, experiments with this approach have shown that it can be reasonably successful for some classes of algorithms.

3.5 Architectures for Branch-and-Bound

Although there are many ways of introdusing parallelism into Branch-and-Bound algorithims, the main way in which this can be achieved is by dynamically dividing the search space between the available processors and expanding multiple problem-states in parallel. The main decision is therefore whether to have a global control, where problem-states of globally highest priority are chosen for expansion, or to have a local control, where only a subset of the problem-states are available. The use of a global control ensures that the priority function is closely adhered to while a local control may lead to the unnecessary expansion of a number of nodes. The use of global control, however, requires that all processors have to a global pool of active nodes. In a message passing machne thia will lead to large communication overheads, while a shared memory machine this may lead to memory contention in certain memory blocks. The shared memory machine may also require the use of semaphores to ensure that processors do not attempt to update the contents of the same memory location simultaneously.

In all cases the current value of the incumbent must be made available to all processors so that it can be used to prune sections of the search tree. This value can either be stored in section of shared memory or can be broadcast to every processor each time a new incumbent is generated. This should not be an important consideration, however, as the value of the incumbent should not usually change very frequently.

Imlementation of extended branch-and-Bound requires access to dead pool for pruning problems; this is most easily done using shared memory to store the problems after they have been expanded. Implementation on message passing machines is likely lead to large communications overheads for accessing the dead pool.

The work required to expand a particular problem-state, and therefore the time required, is often dependent on the actual data in the problem being expanded. For example, it is often much quicker to determine that a problemis immediately feasible than it is to generate all of its children. Similarly, the expansion of problem-states at different levels in the search tree may require differing ammounts of work. This tends to suggest that enforced synchronisation may cause some loss in performance as some processors sit idle, waiting for others to finish. A MIMD machine is therefore is likely to be more effective at solving Branch-and-Bound problems than a similar SIMD ane.

Yu and Wah (1983) and wah and Ma (1984) have suggested a parallel architecture and operating system specifically designed to execute Branch-and-Bound algorithms. This architecture, referred to as MANIP, makes use of special hardware to improve the parallel performance of this type of algorithm.

3.5.1 The MANIP Architecture

The MANIP architecture is designed to perform a best-first expansion of the search tree and uses multiple processors to evaluate multiple sub-problems in parallel. A global data register is used for storing the current incumbent, and is implemented using sequential associative memory to prevent simultaneous updates. Other important features of the MANIP architecture are the sub-problem memory controllers which store the active pool of problem-states and the section and redistribution network which allows the various memory controllers to communicate with each other. The m processors are split into n groups, where each group uses a single memory controller. The n controllers are then connected using a ring network which allows problem-states to be distributed between them.

MANIP is a synchronised architecture so an *m* processor machine should expand *m* problem-states in parallel but must then have a synchronisation step so that the *m* problem-states of highest priority can be identified and distributed to the processors. The designers of MANIP have noted that the selection phase need only identify the m problem-states of highest priority but does not require these to be sorted in any particular way. They believe that the identification of these problem-states can be performed in parallel using a ring network to connect the memory controllers. Hardware comparators are used to identify the highest priority nodes in each memory controller and these are then sent to neighbours in the ring network. The process of identifying high priority work and passing it to neighbours continues until the *m* nodes of highest priority are distributed evenly between the memory controllers.

The MANIP architecture has been simulated on a DEC VAX 11/780 and encouraging results have been generated showing good speed up for the vertex covering problem.

3.5.2 MIMD Approaches

The most common method for performing Branch-and-Bound algorithms in parallel is to expand multiple problem-states in parallel by expanding one problem-state on each processor. There are four main approaches for performing this, depending on whether a local or global active pool is used and whether the system is synchronised or unsynchronised. These approaches are briefly mentioned above but we now explain them in greater detail. We assume a MIMD message passing machine for this discussion and we assume that a heap-tree is used for storing and sorting the active pool.

Synchronised with a Central Active Pool (SCAP)

This strategy uses a global control to ensure that only the problem-states of the very highest priority are expanded at any stage. In every iteration, each of the *m* processors is allocated one of the *m* active problem-states of highest priority to expand. Clearly, a functional requirement of this strategy is that these *m* can be identified and thus a master/slave approach is usually adopted. The master processor is responsible for identifying the *m* problem-states of highest priority at the start of every iteration and one is sent to each of the *m* slave processes. Note, several implementations allow a master process and a slave process to be run in parallel on a single processor.

This task can obviously be performed using a farming technique where all active problem-states are stored on the master processor. At the start of each iteration the master processor identifies the problem-states of highest priority and sends one to each of the slave processors.

During the expansion phase, each slave processor expands the problem- state it has been allocated. As described earlier, expanding a problem-state involves either generating a new feasible solution or generating the children of the problem-state. In the former case, if a new incumbent value has been generated, it is sent to the root while in the latter case the newly generated child problem-states are sent.

Termination of the expansion phase can be determined when a message is received from each processor. Termination of the algorithm is easily detected when there are no active problem-states to distribute during the selection phase.

A disadvantage of this approach is that it can lead to large communication overheads as communication time is usually proportional to message size and the amount of data that must be sent between processors is likely to be very large. The size of a problem-state depends on the definition of the problem, but it will typically require at least O(n), {O=Theta},bytes and possibly O(n^2) or O(n^3) bytes to describe a single state in a problem of size *n*. The master processor can thus become a communication bottleneck as problem-states are sent to the master processor and then back to the slave processors for expansion.

The use of a synchronised system can lead to further overheads as some processors may be forced to remain idle while other processors complete the work allocated to them.

Asynchronous with a Central Active Pool (ACAP)

The implementation of this strategy is similar to that of SCP but does not involve a synchronisation step. Thus, this strategy requires the master processor to allocate an active problem-state (if one exists) to a processor as soon as that processor finishes its expansion phase.

If a processor becomes idle when the active heap on the root is empty, the master processor can increment a count. When new nodes are generated, this count can be decremented and the idle processors can be sent work. Termination of the algorithm occurs when the idle processor count is equal to *m*, the number of processors. This strategy should perform more efficiently than the SCP strategy as processors will not have to wait for the synchronisation step. It will, however, have the same large communication overheads as the earlier strategy and the root processor can become a communication bottleneck.

Synchronised with a Local Active Pool (SLAP)

This strategy uses a local form of control to reduce the communication overheads of the strategy mentioned above. In order to do this, each processor only considers the work available in its own local heap. The processor removes the highest priority problem-state from its heap, expands it, and adds any newly generated children problem-states to the heap. When a processor empties its own local heap, it does not terminate but instead sends a request to its neighbours for more work.

When a new incumbent is generated, its value can be broadcast to each processor in the network so that it can be used to prune unexpanded problem- states. Rather than communicating after each expansion, the processors now need only issue a message if they have run out of work or have generated a new incumbent.

If this system were to be synchronised, it would be necessary for each processor to synchronise after every problem expansion and this would be likely to negate the advantages of using a local control strategy.

Asynchronous with a Local Active Pool (ALAP)

This is similar to the strategy just described, but does not have a synchronisation step so processors continually remove problems from their local heap, expand them and add the newly generated children problems to the heap.

Two potential disadvantages of this method are, firstly, that it is now much more difficult to detect when the algorithm has terminated and, secondly, that processors have no knowledge of the active problem-states stored on other processors.

The first of these difficulties means that it is necessary to use some form of distributed termination detection algorithm while the second means that the *m* problem-states being expanded at any moment in time may not be the *m* highest priority problem-states in the system. In a case where a large proportion of the search tree must be examined this approach is likely to be very efficient as communication costs are minimised, but in cases where only a small part of the search tree is examined it is likely to lead to a large number of unnecessary expansions.

Further important considerations concern how requests for work are handled when a processor empties its local heap of work and how the work is redistributed after this occurs. As already mentioned, this strategy is unlikely to follow the priority function very accurately. This may result in more nodes being expanded than is necessary in implementations which follow the priority function more closely.

In order to spread the high priority work more evenly between the available processors it may be necessary to use some strategy for sending work to other processors in the network after certain intervals. It is likely that the most effective strategy will involve sending high priority work to other processors, but the optimal interval between each such message is likely to depend on the particular algorithm and the hardware being used. Sending work frequently will help to distribute the work more evenly but will obviously also increase the communication overheads as well as the overheads for adding and removing problems from the queue of work.

3.6 Parallel Implementations

In this section we discuss a number of implementations of parallel Branch-and- Bound algorithms using different parallel machines. We begin by discussing two implementations based on the iPSC hypercube, then one on the NCUBE and finally a number of kernels on networks of INMOS Transputers.

3.6.1 Pardalos and Rodgers

Pardalos and Rodgers (1990) have used an iPSC hypercube for parallel processing of a number of quadratic zero-one problems. They use parallel processors to expand multiple problem-states simultaneously but use a depth-first search strategy to reduce memory demands. While MANIP uses specially designed hardware to ensure that only the best nodes are chosen for expansion, Pardalos & Rodgers have opted for a much more local form of control.

This system is similar to the ALAP strategy described earlier and must solve the problem of how to request work when the local heap of work becomes empty.

Obviously it is not possible for processors to predict when a neighbouring processor will empty its heap of work so they must be prepared to receive messages at any time. Unfortunately, on the hypercube, the operations to send and receive messages are very expensive compared to other operations. It is therefore important not to issue communication commands more often than is necessary.

Ideally, messages indicating new incumbent values or requesting more work should be received and processed as quickly as possible to allow pruning or to prevent processors sitting idle. Unfortunately, for the reasons mentioned above, it is inefficient to continually check for incoming messages. The kernel therefore uses a variable MAXV which indicates how often a processor should check for communication from its neighbours.

After every MAXV problem expansions the processor issues a `probe' operation to check for incoming messages. If messages are waiting they are dealt with immediately, otherwise the processor begins another MAXV problem expansions. The choice of value for the parameter MAXV has been found to be fairly critical and is also likely to be highly problem dependent. Too large a value indicates that processors may sit idle for long periods of time while too small a value indicates that processors spend a large proportion of their time communicating. Empirical results from the quadratic zero-one problems being investigated suggest that MAXV should be set to a value of approximately 1000.

Note that the first few iterations of the algorithm are treated as a special case to ensure that all processors have work, and it is only after this point that the MAXV parameter is used.

3.6.2 Clausen and Traff

Clausen and Traff have used a similar method to that described above to run experiments on the graph partitioning problem (Clausen and Traff 1991), also using an iPSC hypercube. The Clausen/Traff kernel uses a similar scheme to the one mentioned above, where messages may only be received after STEPS iterations of the expansion process. They refer to this kernel as the `on demand` strategy where messages are only sent in response to a request for work.

One problem with this strategy is that it is possible for a processor to exhaust its supply of work and then to be idle for a considerable time as its neighbours complete their STEPS node expansions. A second kernel was therefore developed which uses an `on overload` strategy. This allows problem-states to be sent to neighbouring processors if the queue of work on the current processor gets too large.

After completing STEPS expansions the processors check the size of their queue of work; if the queue is larger than some value QUEUEMAX then work is sent to the neighbouring processors. This will hopefully spread the work more evenly between the processors and it should reduce the chance that a processor will run out of work. Processors should therefore spend a greater proportion of their time actually expanding problem-states.

The designers of this system have come up with a number of rules for altering the value of the QUEUEMAX parameter as the algorithm proceeds. These rules are used to allow for the changing amount of work in the system as the algorithm proceeds and seeks to reduce the number of unnecessary messages.

Results presented for these kernels show that, for this particular problem, the use of the `on overload` strategy increases the processor usage dramatically, but often does so at the cost of extra node expansions. This implies that the processors are kept busy but not necessarily doing useful work. This should, however, prove advantageous when a large porion of the search tree must be examined in order to identify the optimal solution.

3.6.3 Quinn

Quinn has performed a number of experiments on an NCUBE hyperube computer using the Travelling Salesman Problem as an example (Quinn 1990). The experiments compared 5 different strategies for the implementation of parallel Branch-and-Bound, one semi-synchronous algorithm and 4 asynchronous ones. The implementation of the Travelling Salesman Problem used by Quinn generates a binary search tree where at any stage a particular edge is chosen and may be included or excluded from the tour.

1. A semi-synchronous algorithm with a global pool of active problem-states. Each time a processor completes a node expansion it sends a request to a master processor asking for more work. This is similar to the **AGAP** strategy mentioned earlier and has problems of high communication overheads as the master processor is a communications bottleneck. This bottleneck actually results in a drop in performance for large networks of processors.

The asynchronous algorithms are all variations on the theme described above for **ALAP.** The different strategies use various methods for distributing the work between the available processors. All of these implementations require a problem-state (if available) to be sent to a neighbouring processor after every problem expansion is completed. The variants are:

2. The problem-state with the included edge is retained by the processor on which it is generated while the problem-state with the excluded edge is sent to one of the neighbours. This

achieves good processor utilisation, but poor speed up as a large number of unnecessary expansions are performed. Only a few processors (those next to the root processor) perform useful work, all other processors performing low priority work which is pruned in the sequential version.

3. A similar algorithm to the one just described, but where the newly created problem-state with the smaller bound value is retained and the one with the higher bound value is sent to a neighbour. Again, a large number of unnecessary expansions are performed so poor speed up is reported.

4. Both newly generated problem-states are added to the queue of work and the problem of second highest priority is removed and sent to the neighbour. The problem of highest priority is thus kept for expansion on the current processor. This is more effective at spreading the high priority work between the available processors but processors very far from the root processor still perform little useful work.

5. Both newly generated problem-states are added to the queue of work and the problem of highest priority is sent to a neighbour. This strategy spreads the work more evenly between the available processors and all processors perform a reasonable amount of useful work. This strategy therefore achieves good speed up as well as reasonable processor utilisation.

The results from these strategies show the importance of distributing the high priority work between all of the processors. If this is not done effectively some processors may only perform low priority work and poor speed up will result. Adding and removing problems from the priority queue will lead to some overheads but these are likely to be insignificant compared to the benefits of ensuring that all processors are performing useful work.

3.6.4 McKeown *et al.*

McKeown *et al.* have developed a number of kernels for running Branch- and-Bound algorithms on a network of Transputers. These kernels are based on a higher-order definition of the Branch-and-Bound paradigm and therefore allow an algorithm to be used interchangeably in any of the available kernels. The kernels, which we now describe, are referred to as `Select Highest Overall', `Select Highest available', `Select Highest Locally' and a hybrid kernel known as `Select Highest Hybrid' . Although these kernels have been developed for a network of Transputers it is hoped that they could be implemented on other architectures and there is a current project to implement similar kernels on a shared memory machine.

Select Highest Overall (SHO)

SHO is a synchronised strategy which uses a global control to ensure that only the problem-states of very highest priority are expanded at any stage. It is thus similar to SGAP and suffers from the communication bottleneck mentioned earlier.

The method used for reducing the communication overheads is for each processor to store the problem-states it generates in its own local memory but to send a small token describing the problem-state to the master processor. This still allows the master processor to identify the problem-states of highest priority but reduces the amount of communication necessary. The tokens sent by the slave processors contain the priority and bound values of the newly generated problem-state together with the number of the processor which is currently storing it. The priority value is used for sorting the tokens while the bound value is used to prune problem-states that cannot lead to a better incumbent. While the master processor identifies the m nodes of highest priority, each slave sorts its active problem-states into a heap ordered by priority values, and considers pruning the heap if memory overflow is imminent.

If the master finds that a processor has more than one of the selected problem-states it sends a message indicating how the problems should be distributed. Processors are also sent copies of the incumbent if a new value has been generated in the previous iteration.

After expanding a node the slave processor compares the newly generated child problems to the current incumbent and sends a token to the master processor for each surviving child

problem. Since this is a synchronised system, there is no advantage in sending the representation of each child individually so all of the new children are sent in a single packet.

Select Highest Available (SHA)

The implementation of the SHA strategy is based on that of SHO but does not involve a synchronisation step. When the master allocates a problem-state to a processor it must also send the priority and bound values of the node in order to identify it. This is necessary because the information stored by the master processor could be slightly out of date with the information stored on the slave processors. When a processor sends a problem-state to another processor it must therefore search its heap for a problem-state with the correct bound and priority values (it is likely to be near to the top of the heap) to ensure that the information in the root processor is kept up-to-date.

A strict implementation of the **SHA** strategy requires that the highest priority work known to the root is sent in response to a request, but a possible relaxation would be to check for work within a certain tolerance of the highest that is already on the requesting processor.

Select Highest Locally (SHL)

The **SHL** strategy is similar to the **ALAP** idea described earlier. The termination detection algorithm used in this kernel is based on one suggested in Topor which uses a spanning tree topology and sends differently coloured tokens between processors depending on the processor status.

When a processor exhausts its own local supply of work it requests work from neighbouring processors. Work is allowed to spread quickly around the network, however, as several pieces of work are sent in response to a single request.

When a processor receives a request for work it checks its status; if it has spare work in its heap (in which case it must currently be expanding a problem-state) then it sends some of the work to the requesting processor using a Fibonacci series (2,3,5,8,13,21. . .) to decide how much work to send. It sends the nodes of 2.(second), 3.(third), 5.(fifth) . . . highest priority until the bottom of the heap is reached. The advantage of this particular approach is that the amount of work sent in response to a request is directly related to the amount of work available on the sending processor and it is also highly biased towards sending work of high priority.

If the processor is currently idle it immediately sends a message indicating this. If the processor is currently expanding a problem-state but has no spare work to send, it will save the request and wait until it either has spare work available or becomes idle. These rules are used to prevent messages swamping a processor which is busy working, but allow messages to be sent freely between idle processors.

In order to spread the high priority work more evenly between the available processors they are allowed to send high priority problem-states to their neighbours at regular intervals. In order to do this an iteration count is implemented indicating how many nodes need to be expanded before work may be sent to neighbours. Thus, when the count is equal to infinity it is a pure implementation of **SHL**, where work is only ever sent in response to a specific request. Otherwise it is a slight variant on **SHL** where unsolicited work is occasionally sent to neighbouring processors.

As an example, in SHL(5) each processor sends work to a neighbouring processor (cycling between each of its neighbours) after every 5 expansions without receiving any request. The work sent in these circumstances is the highest priority work known to the sending processor.

Select Highest Hybrid (SHH)

As mentioned previously, the **SHA** kernel has the potential disadvantage of large communication costs while the **SHL** kernel risks performing large numbers of low priority expansions. The **SHH** kernel attempts to overcome the worst of these problems by dividing the high priority work between the processors and then allowing them to do the work on their own.

The **SHH** strategy begins by running the **SHA** process to generate a number of problem-states and to ensure that the high priority states are distributed between the processors. The kernel then switches to the **SHL** strategy so that those states of high priority can be expanded without the communication overheads present in **SHA**.

The current version of the kernel never attempts to change back to the **SHA** strategy. Although this is possible it would require a considerable overhead and at present there does not appear to be any need for this to be done.

The optimal time to perform the switch between the **SHA** and **SHL** strategies is likely to be problem dependent but one choice would be to change when the active pool reaches a size of approximately 16 * *m*. By using a suitable priority function for this first phase of the algorithm it should be possible to ensure that every processor has at least 3 or 4 nodes of high priority to expand whilst still only forming a small part of the search tree.

The **SHH** kernel appears to have many of the benefits of the system described by Rost and Maehle (1989) where large packets of work are farmed between the processors, but manages to avoid most of the problems inherent in their system.

The initial search to generate a number of problems is now carried out in parallel using the **SHA** strategy and then continues with minimum communication using the **SHL** strategy. When processors exhaust their supply of work they need not sit idle, but can request work from neighbouring processors.

Experiments using a network of Transputers have shown that the amount of communication with this strategy is much lower than with the **SHA** strategy but the number of nodes expanded is much closer to **SHA** than to **SHL**.

Results

The Transputer kernels have been used for the implementation of a wide range of Branch-and-Bound problems and we present here some of the findings.

The first important result is that the **SHO** kernel generally performs poorly when compared to **SHA.** Although **SHO** performs well for a number of algorithms, its synchronisation step hinders performance in cases where the time to expand a problem-state is not constant but depends on the data in the node being expanded. The synchronisation step in **SHO** prevents processors which finished quickly from continuing until the other processors are ready.

Using the **SHA** and **SHL** kernels we have encountered many cases of acceleration anomalies but most cases tend to require slightly more problem expansions when more processors are used. This can partly be explained by the fact that processors are always given work to do if they are idle, even though the work may be of low priority.

The variants of **SHL** which send unsolicited work to neighbouring processors demonstrate the importance of following the priority function and spreading the high priority work between the processors. Generally, the kernels which distribute the work frequently have slightly higher communication costs but require considerably less node expansions.

Experiments with **SHH** show that this kernel has much lower communication costs than **SHA** or **SHL**(1) but expands many fewer problems than **SHL**(oo). In our experiments with the Transputer kernels, however, we have found that the actual communication costs for **SHA** are fairly small for most algorithms and the **SHH** strategy has proved unnecessary. Machines with different communication characteristics (such as the **iPSC** hypercube) may, however, find that the reduced communication in **SHH** allows it to outperform the other kernels.

The 0/1 Knapsack Problem

This algorithm is based on the one described earlier in this chapter and is for an instance of the problem with 750 items, each with a volume chosen randomly from the range 1...800 and with a profit based on the size of the item *(pro fit = size* + 0 . . . 50). The capacity of the knapsack was

$$\sum_{i=1}^{n} v_i/4.$$

As stated earlier, the 0/1 Knapsack problem does not lead to a particularly interesting example of a Branch-and-Bound algorithm and when analysing these results it is important to consider how the algorithm is likely to proceed when expanding the search tree.

The algorithm begins by generating an initial incumbent which may be used for pruning the search tree. This is necessary to reduce the overheads of storing problem-states. If an initial incumbent is not provided then it will be necessary to store every problem-state generated until an incumbent is generated. With an initial incumbent, it is often possible to prune many of these problems immediately. This incumbent is generated by greedily including items in decreasing order of profit/volume ratio and generates a fairly tight initial bound.

The algorithm then expands the search tree as explained earlier by including items or excluding items from the knapsack. The bounding function is, however, quite tight and the search tree generated is very deep but very thin. Such a tree does not, in general, provide much opportunity for exploiting parallelism and thus we would not expect the algorithm to have very good speed up.

The shape of the search tree is due to the very tight bounds which are generated. In the knapsack problem, feasible solutions always appear at the same depth in the tree and in this example they appear at a depth of 750 (every item must be included or excluded). As the expansion rule chooses items to include in the knapsack in the same order as the bound function the search begins with a deep stab into the tree. Each newly generated problem-state will have the same bound value as its parent and will therefore be selected for expansion in the following iteration.

When the knapsack is eventually filled there will be a number of objects which have been included but only marginally ahead of other similar objects. The search tree will therefore grow out from this point as similar objects are included and excluded from the knapsack to find the optimal assignment of items.

When a good assignment is found the algorithm will again stab down towards the leaf as all of the remaining items are excluded. Once again, the children problem-states will probably have the same bound as the parent (e.g. if the knapsack is completely full) and will therefore be chosen in the next iteration.

Unfortunately, in Branch-and-Bound, the amount of processing required is not known in advance so it is very difficult to perform load balancing. Even more importantly, it is not necessary for all of the nodes in the search tree to be expanded to generate an optimal solution to the problem (indeed the whole point of Branch-and-Bound is to reduce the number of nodes that actually need to be expanded) so processors cannot simply be given work at random. These results illustrate these difficulties and show examples of both speed up and detrimental anomalies.

Results for the experiments with the 0/1 Knapsack problem are shown in Tables 5.1-5.4. Looking at the sequential version, we see that only a very tiny section of the search tree is generated (just 4704 problem-states out of a total of nearly 6 * 10^255) and that most of these only generate a single child problem (4704 expansions required only 5088 bound calculations). These results tend to support our statements about the probable shape of the search tree and show that much of the search is inherently sequential.

TABLE 3.1 : 0/1 Knapsack, SHA

Number of Processors	Nodes Expanded	Bounds Calculated	Time (seconds)	Speed-up
1	4707	5088	190.07	1.00
2	1541	2175	32.65	5.82
4	3116	4388	35.69	5.33
8	6145	8856	40.27	4.72
16	14745	21969	48.83	3.89

Table 3.1 shows an example of an acceleration anomaly where the two processor system performs far fewer expansions than the sequential one and achieves a speed up of greater than 2.

This anomaly is due to the two processor system identifying a good solution more quickly than the sequential one and therefore being able to prune more of the search tree.

As the 2 processor system is almost optimal (750 expansions must be performed sequentially so with 2 processors 1500 expansions should be expected) using more processors does not improve the speed up but simply causes more problem-states to be expanded.

TABLE 3.2 : 0/1 Knapsack, SHL

Number of Processors	Nodes Expanded	Bounds Calculated	Time (seconds)	Speed-up
1	4707	5088	183.87	1.00
2	9491	10267	185.07	0.99
4	11278	13004	106.64	1.72
8	17950	21590	86.49	2.13
16	27479	35049	67.61	2.72

Table 3.2 shows an example of a detrimental anomaly where the two processor system actually takes longer than the sequential one. This is due to the second processor not performing any useful work at all and possibly even providing extra unnecessary work for the first processor. As processors will always be sent work if they become idle, the search tree is split at the first possible moment (probably after the expansion of the initial problem) and the two processors independently search for an optimal solution.

Increasing the number of processors does give some improvement in speed up, but much of the work being done is unnecessary due to the algorithm following the priority function very poorly.

The difference in times between the sequential version of **SHA** and that of **SHL** shows the slight overhead associated with storing the heap of tokens as well as the heap of problem-states.

TABLE 3.3 : 0/1 Knapsack, SHL(5)

Number of Processors	Nodes Expanded	Bounds Calculated	Time (seconds)	Speed-up
1	4707	5088	183.87	1.00
2	1675	2337	33.86	5.43
4	3154	4437	32.46	5.66
8	6308	8977	32.85	5.60
16	12644	17968	33.17	5.54

TABLE 3.4 : 0/1 Knapsack, SHL(1)

Number of Processors	Nodes Expanded	Bounds Calculated	Time (seconds)	Speed-up
1	4704	5088	183.87	1.00
2	3665	4332	77.17	2.38
4	5665	7082	62.86	2.93
8	6138	8731	36.96	4.97
16	12269	17375	37.97	4.84

Tables 3.3 and 3.4 show how these variations on **SHL** perform on this particular problem. Little can be gained from these results however due to the particular characteristics of this problem which, as we have already demonstrated, can have both severe acceleration and deceleration anomalies.

One interesting point that can be seen from these results, however, is the communication costs of sending problem-states too frequently. The 16 processor version of **SHL**$_5$ performs more expansions than the equivalent **SHL**$_1$ one but does so more quickly due to smaller communication overheads.

Tbe Travelling Salesman Problem

This algorithm is based on one by Little *et al.* (1963) and uses a reduced cost matrix to generate a search tree. The tree formed is a binary tree where edges between cities are either included or excluded from the solution. The choice of edge to branch on is made on the basis of trying to maximise the cost difference between the two branches of the tree. This will, hopefully, exclude the largest possible part of the tree in one go.

The search tree generated by this algorithm is very different from the one for the Knapsack problem. Solutions may occur at different depths in the search tree (there must be 30 included edges but the number of excluded edges may vary) which will generally be fairly bushy. We would therefore expect this algorithm to get an increase in speed up as more processors are added.

The results refer to an instance of the travelling salesman problem consisting of 30 cities with inter-city costs being randomly chosen in the range 0 . . .99 and using a best-first search strategy.

TABLE 3.5 : Travelling Salesman Problem, SHA

Number of Processors	Nodes Expanded	Time (seconds)	Speed-up	Pseudo Efficiency
1	1526	203.7	1.0	100.00
2	1457	97.3	2.1	99.9
4	1456	48.8	4.2	99.6
8	1459	24.9	8.2	97.8
16	1746	15.0	13.6	97.1

Table 3.5 shows that the **SHA** strategy performs consistently well and gets some slight speed up anomalies in some cases. The number of problem-states expanded is reasonably predictable and the performance is therefore also fairly predictable.

TABLE 3.6 : Travelling Salesman Problem, SHL

Number of Processors	Nodes Expanded	Time (seconds)	Speed-up	Pseudo Efficiency
1	1526	201.9	1.0	100.00
2	1713	104.8	1.9	108.1
4	2078	67.9	3.0	101.2
8	2211	36.6	5.5	99.9
16	3310	29.3	6.9	93.4

Table 3.6 however shows how important it is to ensure that high priority work is distributed evenly between the processors. Although this strategy is very efficient at expanding work, much of this work is unnecessary and speed up is therefore not as good as may be expected.

TABLE 3.7 : Travelling Salesman Problem, SHL(10)

Number of Processors	Nodes Expanded	Time (seconds)	Speed-up	Psedo Efficiency
1	1526	201.9	1.0	100.00
2	1510	94.3	2.1	105.9
4	1512	45.6	4.4	109.6
8	1766	27.7	7.3	105.4
16	1965	16.3	12.4	99.8

TABLE 3.8 : Travelling Salesman Problem, SHL(1)

Number of Processors	Nodes Expanded	Time (seconds)	Speed-up	Psedo Efficiency
1	1526	201.9	1.0	100.0
2	1462	96.5	2.1	100.2
4	1499	49.4	4.1	100.3
8	1565	25.4	8.0	101.9
16	1565	12.9	15.7	100.6

Tables 3.7 and 3.8 show that by sending unsolicited work between the processors the high priority work is spread more evenly and the number of nodes expanded is considerably reduced. A small communication overhead therefore leads to a great saving in unnecessary work, as can be seen by the 16 processor version of **SHL**1 which is still achieving very near linear speed up.

Chapter 4

DISTRIBUTED PROCESSING

4 .1 PROCESSES AND THREADS

A Process is a logical representation of a physical processor that executes program code and associated state and data (a process is sometimes described as a virtual processor). A cess is also the unit of resource allocation and so is defined by the resources it uses and by location at which it is executing. A process can run either in a separate address space (i.e. a private range of addresses available to it) or may share the same address space. processes are created either implicitly (for example by the operating system when a program is to be executed) or explicitly using an appropriate language construct or operating system function such as fork ( ) in the UMX environment.

In uni-processor computer systems many processes appear to be running at the same time. In reality there is never more than one process executing on a single CPU; a time slicing tech- nique is used to enable multiple processes to use it, switching between them so rapidly that, under the right conditions, each process seems to be executing continuously. Switching between processes involves saving the state of the currently active process and setting up the state of another process (this is sometimes known as **context switching).** Context switching is carried out by very low-level code in the operating system kernel. Many operating systems (e.g. UNIX) allow programs to create additional processes (other than the one it is running in) thus enabling multiple concurrent `child' processes to be executing, each competing for CPU and other resources as with other processes. In the UMX fork ( ) mechanism, when a child process is created using fork ( ) all the resources belonging to the original program (running in the `parent' process) are duplicated thus making them available to child processes. Thus child processes have their own address spaces but with duplicated resources.

It is a common requirement for a program to create multiple processes which are required to share memory and other resources. For example, a program may wish to create a subprocess to wait for a particular event. Some operating systems support this situation efficiently by allowing a number of processes to share a single address space. Processes in this context are often referred to as **lightweight processes** or **threads** and the operating system is **said to support multi-threading.** A thread is thus the unit of scheduling and execution in a multi-threaded OS. Memory and other resources are allocated to the address space within which threads are executing (an `address space' and associated resources are variously known as a task, actor, process, cluster, etc.). Context switching between threads in the same address space is much faster than context switching between (heavyweight) processes in separate address spaces. The main drawback is that because all threads share memory and other resources, there is much more scope for conflicts and programs need to be carefully written to detect and recover from such conflicts. This can be achieved by use of semaphore and locking mechanisms for co-ordinating actions on specific resources as discussed in the next section. In this part, the term `process' is used to refer to heavyweight and lightweight processes (threads) unless stated otherwise.

In multi-processor systems multiple processes and threads can execute simultaneously (in parallel); one per active CPU without the need to re-write programs. Thus, this approach to implementing concurrent programs is independent of the underlying processor architecture.

4.2 SYNCHRONIZATION OF CO-OPERATING PROCESSES

There are two main reasons why there is a need for synchronization mechanisms .

1. Two or more processes may need to *co-operate* in order to complete a given task. This implies that the operating mechanism must provide facilities for identifying (naming) co-operating processes and synchronization of processes with each other.

2. Two or more processes may need to *compete* for access to shared services or resources. The main implication is that the synchronization mechanism must provide facilities to allow one process to wait for a resource to become available and another process to signal the release of a resource.

When processes are running on the same computer, process synchronisation is straight forward since all processes use the same physical clock and can share memory. Synchronisation via shared memory is achieved using well-known mechanisms such as semaphores which are designed to provide mutually exclusive access to a non-shareable resource by preventing concurrent execution of the critical region of program code through which the non-shareable resource is accessed. Mutual exclusion is achieved by enclosing each critical section of code by WAIT (mutex) and **SIGNAL** (mutex) semaphore operations where mutex is the name of the semaphore and is initialised to the value 1. For example, if a semaphore called mutex has been allocated and initialised to 1, then the program code fragment in following below Figure 4.1 illustrates how the critical section is coded. Semaphores can also be used to synchronize co-operating processes by ensuring that one process will wait on another based on the occurrence of a particular event, and for the waiting process to be signalled that it can now proceed. For example, Table 4.1 shows the use of WAIT and SIGNAL operations to synchronize two concurrent processes.

```
/* execute non-critical section of code                    */

 /* block the current process until it can

     acquire the mutual exclusion lock:-                    */

     WAIT(mutex);

/* execute critical section of code                        */
     */ critical section(.....)

/* release the mutual exclusion lock:-                     */
    SIGNAL (mutex) ;

/* execute non-critical section of code                    */
```

Figure 4.1 Mutual exclusion using semaphores.

Table 4 .1 Synchronization using semaphores

Process A	*Process B*
/*synchronize with process **B** */	-
WAIT(sync)	-
/* proceed since now synchsonized */	**SIGNAL** (sync)
-	
-	

Semaphores can be used as a synchronisation mechanism in all types of process interactions when processes share memory by using multiple semaphores with appropriate initial values. The initial value of a semaphore is the maximum concurrent usage of a resource. The value of a semaphore at any instance indicates the number of units of resource available.

An alternative approach to synchronisation of concurrent server processes is eventcounts and sequencers (Reed and Kanodia, 1979). Eventcounts allow processes to co-ordinate their actions by observing the sequencing of event occurrences. An eventcount is an object that keeps count of the number of events of a particular type that have occurred so far. It is typically represented by a non-decreasing integer variable, initialised to zero, and with the following primitive operations:

ADVANCE (eventcount) - signals the occurrence of an event associated with an eventcount by increasing the eventcount by 1.

READ(eventcount) - returns the current value of eventcount.

AWAIT(eventcount, value) - blocks the calling process until the eventcount is greater or equal to value.

Sequencers are required to totally order events. A sequencer is also typically represented by a non-decreasing integer variable, initialised to zero, but with the following primitive operation:

TICKET(sequencer) - returns the current value of sequencer then increases the value.

The TICKET operation is analogous to the use of numbered tickets in restaurants and busy shops to control the order of service. Customers are served in the order of the number on their ticket. To illustrate the use of eventcounts and sequencers, suppose a banking service is defined with an operation CreateAC which creates a new bank account so that, for example, unique account and PIN numbers are generated and to ensure that account data consistency is maintained. An eventcount and sequencer can be used to implement a mutual exclusion (mutex) mechanism to ensure that only one CreateAC operation can execute the critical section of code at any moment in time. When a server receives a client request it creates a child process to handle it. Thus multiple child processes may be executing in the non-critical section of the CreateAC operation. An eventcount called CREATE EC and sequencer called CREATE SQ are defined and initialised on server startup. A child process about to enter the critical section first acquires a ticket by executing:

myturn = TICKET(CREATE -SQ)

which returns a value in the variable myturn to represent the position of the process in the queue of client processes waiting to execute the critical section. An AWAIT (CREATE - EC, myturn) operation is executed which has the effect of blocking the child process until CREATE EC is greater or equal to myturn. When this occurs, the child process has permission to execute the critical section. When the critical section has been executed, the child process hands control to another process by executing operation ADVANCE (CREATE-EC) ,which effectively allows the next child process to enter the critical section.

4.3 INTER-PROCESS COMMUNICATION

When processes in the same computer wish to interact they need to make use of an inter process communication (IPC) mechanism which is usually provided by the native operating system or presentation management components. A number of mechanisms are available. Perhaps the most primitive IPC is a synchronous filter mechanism. Most UNIX and MS-DOS users are familiar with the *pipe* mechanism which is an example of a filter mechanism. for example:

ls -1 / more

The commands ls and more run as two concurrent processes, with the output of ls connected to the input of more. This has the overall effect of listing the contents of the current directory one screen at a time. Many processes can be piped together in this way. As a synchronisation mechanism, normally the source process will write to the pipe until it is full (the maximum size of a pipe data stream is implementation dependent) then block until some data is read by the target process. If the target process reads an empty pipe which has not been closed by the source process it will block until the source process writes some data to the pipe. If the source process closed its end of the pipe, a read on an empty pipe receives an end-of-file condition. The main advantage of pipes is simplicity. Pipe linkage mechanisms are unidirectional and bound to specific source and target processes, however, and do not offer a secure means of communication.

Named pipes overcome some of these limitations by allowing a pipe link to exist without being bound to source and target objects. Unrelated processes can establish communication using a named pipe facility because the interface is much like an ordinary file, indeed, named pipes are often registered in the operating system's file system with an access control list equivalent to an ordinary file for security purposes. Once named pipes are created they behave in a similar manner to ordinary pipes.

An alternative IPC mechanism is use of the local file. The principal advantages of this method are that it can handle large volumes of data and it is a well-understood approach which has been the basis of online information systems for decades. A database is a collection of data items which can be read from or written to by co-operating processes as a means of passing data or synchronising. The main drawbacks are that there are no inherent synchronisation mechanisms between communicating processes to avoid state data corruption, therefore synchronisation mechanisms (e.g. file and record locking) need to be developed to allow many concurrent processes to communicate while preserving data consistency. Secondly, communication is inefficient since it usually relies on a relatively slow, non-volatile disk storage facility.

Since all processes are local, the computer's random-access memory can be used to implement a *shared memory* facility using random-access memory. A common region of memory addressable by all concurrent processes is used to define shared variables which are used to pass data or for synchronisation purposes. Processes must use semaphores or other techniques since there is no inherent synchronisation mechanism. This is a very efficient mechanism but normally

cannot cope with large data transfers. An example of a shared memory mechanism is *a clipboard* facility supplied by most presentation management software.

A common asynchronous linkage mechanism is *a message queuing* mechanism which provides the ability for any process to write to a named queue and for any process to read from a named queue. Synchronisation is inherent in the read/write operations and the message queue which together support asynchronous communications between many different processes. Messages are identified by a unique identifier or by message type. Security is implemented by associating an access control mechanism which identifies the owner process and read/write rights permission for other processes. The main limitation is that these systems are designed to hold relatively small amounts of data.

A common synchronous IPC is the use of normal procedure calls using dynamic link libraries. When a procedure which resides in a dynamic link library is called, the binding between caller and called procedure is resolved at that point. This mechanism is widely used since it is well defined and well understood by application developers and facilitates software component reuse. Data is passed as parameters of the procedure call which means that only relatively small amounts of data can be passed between processes.

4.4 STRUCTURING A DISTRIBUTED SYSTEM

Six main paradigms are commonly used to structure an information system

- **the master-slave model**
- **the client/server model**
- **the peer-to-peer model**
- **the group model**
- **the distributed object model**
- **the multimedia streams model.**

4.4.1 The master slave model

A master-slave model may be an appropriate model for structuring a distributed system. In this model, a master process initiates and controls any dialogue with other (slave) processes. Slave processes exhibit very little intelligence, responding to commands from a single master process and exchange messages only when invited to by the master process. The master process defines the command set and appropriate responses associated with the dialogue. The slave process merely complies with the dialogue rules. . This model has limited application in a distributed IT infrastructure because it does not make best use of distributed resources and the master process represents a single point of failure.

4.4.2 The client/server model

The client/server model is the most widely used paradigm for structuring distributed systems. A client requests a particular service. One or more processes called *servers* are responsible for he provision of a service to clients. Services are accessed via a well-defined interface which is made known to the client community. On receipt of a valid request, a server executes the appropriate operation and sends a reply back to the client.

This type of interaction is known as a **equest/reply** or **interrogation.** Alternatively, an interaction could be initiated by a client resulting in the conveyance of information to the server requesting a function to be performed by the server process. This type of interaction is known as an **announcement** and is clearly one-way communication as no reply is sent back to the client.

A client can potentially generate a request for service at any time. A server can have multiple interfaces (e.g. a management interface and a separate interface through which client requests are handled).

Both clients and servers normally run as user processes. A single computer may run a single client or server process or may run multiple client or server processes (or both). A server process is normally persistent (non-terminating) and provides services to more than one client process.

The main distinction between master-slave and client/server models lies in the fact that client and server processes are on an equal footing but with distinct roles. The use of a (small) manageable number of servers (i.e. increased centralisation of resources) improves systems management compared with the case where potentially every computer can be configured as client and server. This model, known as a **peer-to-peer** model, is so-named because every process has the same functionality as a peer process as illustrated .

4.4.3 The group model

Client/server interaction involves two communicating parties: a client and a server. Another model which is appropriate to some types of distributed systems is the **group** model. In many circumstances, a set of processes need to co-operate in such a way that one process may need to send a message to all other processes in the group and receive responses from one or more members. For example, in a video conference involving multiple participants and a whiteboard facility, when someone writes to the whiteboard, every other participant must receive the new image. A second example is a computer conferencing system (e.g. the Internet USENET system). When a subscriber sends a message to a news item, all other subscribers receive it. This is modelled conveniently as set of group members which behaves as a single unit called a group. When a message is sent to the *group interface,* all members of the group receive it. How does a `group' message get routed to every member? There are three main approaches

1. Send a separetecopy of the message to be individually routed to each member. This is known as unicasting.An implicit assumption is that the sender knows the address of every member in the group. This may not be possible in some system. In the absence of more sophisticated mechanisms, a system can resort to unicasting if member addresses are known. The number of members in the group.
2. Send a single message with *a group address* which can be used for routing purposes. This is known as *multicasting* and relies on an underlying multicasting network facility. For example, multicasting is supported in most LAN protocols and the TCP/IP protocol. . when a group is first created it is assigned a unique group address. When a member is added to a group, it is instructed to listen for messages stamped with the group address as well as ,for its own unique address. This is an efficient mechanism since the number of network transmissions is significantly less than for unicasting.
3. *Broadcast* the message by sending a single message with a broadcast address. The mesage is sent to every possible entity on the network. Every entity must read the message and determine whether they should take action or discard it. This may be an appropriate mechanism in the case when the address of members is not known since most network protocols implement a broadcast facility. If messages are broadcasted frequently, however, and there is no efficient network broadcast mechanism, the network soon becomes saturated as broadcasts are propagated over an internet (known as a broadcast storm). Therefore broadcast-based group communications do not scale well unless an efficient low-level broadcast facility is available.

In some cases, a group message (e.g. an update request to a group of replica servers) must be received by *all* group members or none at all. Group communication, in this case, is said to be **atomic** or **all-or-nothing.** Achieving atomicity in the presence of failures is difficult, resulting in many more messages being exchanged. Another aspect of group communications is the ordering

of group messages. For example, in a computer conferencing system a user would expect to receive the original news item before any response to that item is received. This is known as **ordered multicast** and the requirement to ensure that all multicasts are received in the same order for all group members is common in distributed systems. Atomic multicasting does not guarantee that all messages will be received by group members in the order in which they were sent. Clearly, implementing group communications protocols is non-trivial. An example of a group communication system which addresses ordered multicasts and, to some extent, atomic multicast, is the ISIS system developed at Cornell University In ISIS, broadcast primitives are defined to ensure that all messages arrive in the same order to all parties (e.g. group members) with fault tolerant features. ISIS is described in detail in Tanenbaum (1995); and Coulouris *et* al.(994).

Often group processes are directly supporting groups of people who are working informally or formally on projects for which co-operation is of mutual benefit. Information system to support computer-supported co-operative (or group) working (commonly abbreviated to CSCW) are commonly based on the group model,

4.4.4 The distributed object model

The terms *object* and *object-oriented programming* are used ambiguously in the field of computer science which causes much confusion if not appropriately defined. Object technology offers a different programming model from traditional structured programming which is based on the separation of data and the processing of data through functions and procedures. An object is a computational entity which encapsulates both private data describing its state and a set of associated operations as a representation of a real-world object. An object is described in terms of its *attributes* (internal state data) and methods (procedures which operate on the state data).

An object's state is visible only within the object and is completely protected and hidden from other objects as a way of hiding complexity and as a protection from misuse. The only way to examine or modify an object's state data is by sending *a message* to the object which has the effect of invoking one of a well-defined set of methods that define the object's inter- face to the outside world. Thus without knowing the detail of how an object is implemented, it can be used by simply knowing the methods that define the object's interface. An object may invoke other objects, allowing the creation of a potentially complex web of object invocations. An object-based or object-oriented model is recognized as offering important advantages in the development and maintenance of distributed information systems for the following reasons.

- Implementation detail is hidden and the use (and re-use) of objects emphasized through the definition of well-defined object interfaces. This can lead to greater development productivity, and reduced software maintenance costs.
- Objects can be written to handle a wide range of media types (including text, voice, animation and video) using essentially the same object interface which provides a consistent approach to handling multimedia objects.
- It provides the foundation for DIS integration into specific line-of-business solutions by re-packaging existing functionality.
- An object is a natural unit of distribution. The message-passing nature of inter-object communications maps easily to a distributed systems environment. A message-passing or RPC remote IPC mechanism can be used to implement the communications channel through which messages can be passed between objects.

A class is a template from which objects may be created (every object is therefore an *instance* of some class in as much as a class exists at program compile time, but objects exist at execution time with state representations and associated operations as defined by the class). *Inheritance* is a mechanism that permits new classes to be defined as an extension of another (previously defined) class which provides support for creating new objects by extending existing

templates. p*olymorphism* (which means `many forms') provides the ability for different (but related) objects to share the same method names, thus allowing different objects to be accessed through the same interface. A technique called *overloading* implements polymorphism by ensuring that, even though there may be multiple methods with the same name but different method definitions, the method definition used is one which is appropriate to the type of object involved. The overall effect is, for example, of multiple object types each representing a different media type which can be hidden behind a common object interface (e.g. all objects define open, create, display, print and delete methods but with different method definitions).

Programming languages that support objects as a language feature are known as *object based* and *object oriented* if they also support inheritance. The *sequential* object. model is based on the notion of active and passive objects. An object is activated when it receives a message from another object. The object that sent the message becomes passive as control is transferred, and waits for a result. After servicing the message and sending the result, control is transferred to the waiting object which now becomes active and the sender passive. At any instant, only one object in the system is active. This is similar to process activation and passivation.

Current widely-used object-oriented systems usually assume the sequential object model with all objects running in a single name space on a single computer. The distributed object model allows objects to persist in an active state irrespective of whether they have sent a message or received a result. In fact the model allows an object to send messages to multiple objects concurrently and receive results asynchronously.

Programming languages that support distributed objects define an object as the unit of distribution. The distributed object interaction model differs from the client/server model in that an object can both request and provide services . In this sense, objects interactions resemble peer-to-peer interactions except that objects can exhibit different functionality. Since each object-to-object interaction is request/reply dialogue, for the purposes of object modelling it can be useful to view an object as a composite client and server assemblage. A popular standard for implementing distributed objects is the Object Management Group's common object request broker architecture (CORBA).

4.4.5 The multimedia stream model

Modelling of interactions between multimedia objects uses object-oriented and distributed object paradigms. Object-orientation is particularly appropriate as features such as exchanging object references and content-specific operations (using polymorphism) are readily supported. A **multimedia stream** can be described as a continuous medium with a well-defined start and end, which takes place at a defined rate and exhibits unstructured behaviour (Linington 1994). Thus, *time* is a very important parameter. A stream may be labelled with a set of events which can signify changes in presentation state or used to trigger display actions. A multimedia stream may also have attributes such as the default display size. One important consequence of the encapsulation of potentially diverse, unstructured data is the need to provide the controlling application with more freedom to configure resources to support it.

A multimedia stream interface is one in which all interactions are flows. A flow is an abstraction of a sequence of interactions, resulting in conveyance of information from a **producer** object to a **consumer** object. For example, a videoconferencing stream interface may consist of three flows: audio, video and data. In the MIME convention this was modelled as a multipart / parallel content-type

Each stream interface comprises a finite set of **action templates,** one for each flow type in the stream interface. Each action template for a flow contains:

1. the name of the flow;
2. the information type of the flow (e.g. audio, video or data);
3. whether it is a consumer or producer (but not both).

A flow has a set of **frames** (or signals).

A frame has a name and a set of typed arguments **(attributes).**

Streams are themselves typed and can be conformance type checked. The analogy here is the content-type field in MIME which can be regarded as indicating the *type* of the MIME stream.

Frames are transmitted by producer via non-blocking writes and read by consumers via blocking reads. This is analogous to writing frames to a buffer; and application reads from the buffer.

This model provides a flexible approach to modelling multimedia streams.

4.5 REMOTE IPC

In a distributed TT infrastructure, processes interact in a logical sense by exchanging messages across an information network. We refer to this form of communication as *remote IPC.* As with local processes, remote processes are either co-operating to complete a defined task or are competing for the use of a resource. In the physical network remote IPC can be implemented using the message passing paradigm or the shared memory paradigm. typical remote IPC functions based on the message passing paradigm are:

- process registration for the purposes of identifying communicating processes;
- establishing communication channels between processes;
- reliably routing messages to the destination process;
- synchronisation of processes in the case where processes are competing or co-operating in a y that requires it
- closing down communication channels.

In general, the message passing paradigm is more widely used in commercial systems. Object-oriented concepts naturally embrace the concept of message passing and therefore map easily to message passing or remote procedure calls as remote IPC mechanisms for inter object communication in a distributed object environment.

4.5.1 Binding

At some point, a process needs to determine the identity of the process with which it desires to establish a connection. This is known as binding. A process generally can be bound to another process at one of two stages:

1. Destination processes are identified explicitly at program compile time (and therefore cannot easily be changed). This is known as static or early binding;

2. Source to destination bindings are created, modified and deleted at program run-time. This is known as **dynamic** or **late binding.**

While static binding is the most efficient approach and is most appropriate when a client almost always binds to the same server, in some systems (e.g. accessing external information services) it is often not possible to identify all potential destination processes. Dynamic binding facilitates location and migration transparency when processes are referred to indirectly (e.g. by name) and mapped to the location (address) at run-time. This is often facilitated by a facility, known as a **directory service,** which can be used by the sender to locate a server. When a server is first activated it **exports** information to the directory service regarding the type of service being offered and where it can be located. The sender **imports** its requirements to the directory service, which returns the address of a server which can meet its requirements. . This is analogous to finding the telephone number of `Harry Smith' in a telephone directory and, like the telephone

directory enquiry service, is normally implemented as a `global directory service' that provides this service.

Both binding mechanisms connect source to destination regardless of the quality of connection required by the application. For example, an application may be time sensitive and a slow network link is almost as catastrophic as a network failure. It may be that judicious choice of network connection and protocols and with priority access, a more acceptable quality of service may have been negotiated. This leads to the idea of **explicit binding** which gives applications control over the binding mechanism, allowing negotiation of much more complex configurations to meet a particular application's quality of service requirements. If the requirements cannot be met, the application can take appropriate action. To facilitate explicit binding, each client and server must initiate some negotiation which establishes whether the necessary resources can be marshalled to meet the particular quality of service (Qos) specified (usually by the client). A **binding manager** (also known as a binding object) encapsulates the explicit binding mechanism and is visible to client and server processes. An explicit binding manager: (1) negotiates with other binding managers with a view to matching user quality-of service requirements to IT infrastructure component capabilities (QoS para- meters are crucial as a basis for matching requirements to capabilities); and (2) assists in local resource scheduling and control.

4.5.2 Connectionless and connection-oriented communication

A consideration when supporting process interactions is whether to establish successful contact first (a connection) with the destination process before any messages are sent. If this **connection-oriented** (also known as *virtual circuit)* approach to IPC is used then it minimises the overheads of subsequent message transfer by, for example, setting up a routing path during connection which all messages follow avoiding the need to send full addressing information for each subsequent message sent (a connection identifier is used instead which requires fewer bytes). Also it is more straightforward to employ error control, flow control, sequence control and other protocols for enhanced reliability. If the destination process is contacted initially then this is an opportunity for negotiation of some aspects relating to subsequent dialogue. At the end of the dialogue, the connection must be closed down to release network resources .the telephone system is a good example of use of the connection-oriented approach.

However, if a relatively small number of messages are exchanged during a dialogue between co-operating processes then the protocol overhead due to connection establishment and subsequent close down is significant. In this case a **connectionless** (also known as *data grams)* approach is appropriate whereby no initial connection is made; instead, each message is transported as an independent unit of transfer and carries data sufficient for routing from originating process to destination process. The postal system is an example of the use of this approach. The use of elaborate protocols to enhance reliability is normally dispensed with in order to minimize end-to-end transmission delay. Further error control can be achieved using higher-level protocols implemented on the end host computers.

The rule-of-thumb guideline for selecting the most appropriate communication type is to use the connectionless approach when a typical dialogue consists of the exchange of a small number of messages only, otherwise connection-oriented is more efficient. Most practical implementations of message passing support both approaches.

4.5.3 Synchronization

Another consideration in remote IPC mechanisms is whether a process should be delayed (known as *synchronous* or *blocked* until it receives a response from the destination process. A primitive is *non-blocking* (or *asynchronous)* if its execution never delays the invoking process. Non-blocking primitives must buffer messages to maintain synchronization. Non- blocking clearly maximizes flexibility but often make application development difficult due to the increased complexity of asynchronous time-dependent programs. Synchronization is easy to maintain and programs easier to write when blocking versions of message passing operations are used. When the send message operation is executed, the invoking process is blocked until the receiver actually receives the message. A subsequent receive message operation again blocks the invoking process until a message is actually received. This has the effect of synchronize sending and receiving processes. Co-operating processes, however, may need to synchronize without using blocking operations.

4.6 REMOTE IPC: MESSAGE PASSING

A `low-level' remote IPC (in as much as the application developer is usually explicitly aware of the message used in communication and the underlying message transport mechanisms used in message exchange) is message passing. Processes communicate directly using send and receive or equivalent language primitives to initiate message transmission and reception, explicitly naming the recipient or sender, for example:

send(message, destination-process)

receive(message, source-process)

This is known as message passing using **direct communication.** Message passing is the most flexible remote IPC mechanism in that it can be used to support all types of process interactions (e.g. client/server, group or distributed object) and underlying transport protocols and can be configured by the application according to the needs of the application.

Another useful technique for identifying co-operating processes is known as **indirect communication.** Here the destination and source identifiers are not process identifiers. Instead, a **port** (also referred to as a **mailbox)** is specified which represents an abstract object at which messages are queued. Potentially, any process can write to a port or read from it. To send a message to a process using this mechanism, the sending process simply issues a send operation specifying the **well-known port** number which is associated with the destination process. For example:

send(message, destination-port)
receive(message, source-port)

Normally, the receiver creates the port (i.e. is the owner). To receive the message, the recipient simply issues a receive specifying the same port number. Security constraints can be introduced by allowing the owning process to specify access control rights on a port. Messages are not lost providing the queue size is adequate for the rate at which messages are being queued and dequeued. Clearly multiple ports can be associated with communication between two processes thus supporting multiple, unidirectional or bi-directional channels. A good example of this approach is the UNIX Sockets IPC which will be examined in the next section. This approach provides the flexibility for programming distributed systems and is used extensively in implementing distributed services.

An extension of message passing by indirect communication is known as **message queuing.** In this higher-level form of message passing, store-and-forward techniques are used to propagate general-purpose messages reliably and asynchronously from a local message queue (usually held on non-volatile storage) to a remote queue associated with the destination process. The creation

of message queues and routing of messages is handled by **queue managers.** One significant advantage of message queuing is the ease in which it supports parallel communication with multiple processes. Message queuing also moves much of error handling logic to systems software which is hidden from users.

4.7 REMOTE IPC THE REMOTE PROCEDURE CALL

The client/server model is essentially a request/reply interaction. This interaction is very similar to the traditional procedure call in a high-level programming language except that the caller and procedure to be executed are on a different computer. A procedure call mechanism which permits the calling and called procedures to be running on different computers is known as a **remote procedure call (RPC)** (Birrell and Nelson, 1984). **RPC** is a popular mechanism for developing distributed systems because it looks and behaves like a well- understood, conventional procedure call in a high-level language (all programmers are familiar with the concept of calling a subroutine or procedure). A procedure call has proved to be an effective tool for implementing abstraction since to use a procedure all one needs to know is the name of the procedure and the arguments associated with it. RPC is therefore a remote operation with call semantics similar to a local procedure call, and can provide a degree of:

- access transparency since a call to a remote procedure may be similar to a call to a local procedure. In practice, there will be differences in semantics due to the need, for example, to handle a wider variety of exceptions;
- location transparency since the developer can refer to the procedure by name, unaware of where exactly the remote procedure is located;
- synchronisation between processes since the process invoking the RPC call is normally suspended (blocked) until the remote procedure is completed, just as in a call to a local procedure.

An RPC protocol is implemented in the following way

1. When a process makes an RPC call (e.g. **RemoteProcX (x,y, z, result)),** the address of the server process is first determined (e. g. via a directory service). The call parameters x, y and z are packed into a data structure suitable for transfer across the network. This is called marshalling. These steps could be carried out by the client process, but to simplify the RPC interface from the perspective of the application developer, the potentially complex steps of name resolution and marshalling are carried out by a special procedure called the *client stub.*
2. The data is then passed to lower level RPC transport protocol (e .g. message passing using Send and Receive primitives) for transporting the RPC call and call parameters across the network.
3. The server process unmarshalls the call parameters from the data structure in a form suiable for making a local procedure call. The server process has an associated *server stub* that carries this out transparent to the server process.
4. The call is then made to the required procedure (in this case **RemoteProcX (x, y, z)).**
5. The procedure result is marshalled by the server stub and sent back across the network to the client process using the same low-level RPC transport protocol
6. The client stub unmarshalls the result and passes the result (and control) back to the client process.

4.7.1 RPC exceptions

The above mechanism, however, needs to cope with a wider range of exceptions than is typiçal of a local procedure call. For example:

- what if parameters x ,y and z are either global variables or pointers? Many programming languages can support parameter passing using call-by-value (a copy of data is passed) or call-by-reference (a pointer to the data item is passed);
- what if there are differences in the way that client and server computers represent integers, floating point and other data types?
- what if the RPC call fails? Can the call be recovered?
- is the client authorized to call the named procedure?

Marshalling is complicated by use of global variables and pointers as they only have meaning in the client's address space. Client and server processes run in separate address spaces on separate machines. One solution may be to pass the data held by the global variable or pointed to by the pointer. However, there are cases when this will not suffice, for example, when a linked list data structure is being passed to a procedure that manipulates the list.

Differences in data representation can be overcome by use of an agreed language for representing the data being passed between client and server processes. For example, a common syntax for describing the structure and encoding of data, known as **abstract syntax notation** (ASN.1) (ISO 8824) was defined by as an international standard by the International Organisation for Standardisation (ISO) for this purpose. ASN.1 is very similar to the data declaration statements in a high-level programming language, and a useful description is found in Tang and Scoggins (1992). Marshalling is then a case of converting the data types from the machine's representation to a standard representation (e.g. ASN.1) before transmission. At the other end, the data is converted from the standard to the machine's internal representation.

4.7.2 Failure handling

RPC failures can be difficult to handle. There are four generalized types of failure that can occur when a **RPC** call is made:

1. the client request message gets lost;
2. the client process fails while the server is processing the request;
3. the server process fails while servicing the request;
4. the reply message is lost.

If the client message gets lost then the client will wait forever unless a time-out/retry error detection mechanism is employed. If the client process fails, the server will carry out the remote operation unnecessarily. If the operation involves updating a data value (e.g. updating bank account details) then this can result in loss of data integrity. Furthermore, the server would generate a reply to a client process that does not exist. This must be discarded by the client machine's communication system. When the client process re-starts, it may send the request again (as if for the first time) causing the server to execute the same operation more than once.
A similar situation arises when the server crashes. The server could crash just prior to execution of the remote operation or just after execution completes but before a reply to the client is generated. In this case, clients will time-out and continually generate retries until either the server restarts or the retry limit is met.

4.7.3 Execution semantics

The number of times a remote procedure executed can be difficult to determine in the presence of failures. Three kinds of RPC execution semantics are defined which can be guaranteed by a particular RPC mechanism in failure situations:

1- At-most-once semantics: the **RPC** mechanism guarantees that the remote procedure is either never executed or executed partially or once. This requires the server to keep track of invocation identifiers of all procedures previously executed to avoid duplication.
2- Exactly-once semantics: the **RPC** mechanism guarantees that the remote procedure is executed exactly once. This is difficult to achieve given the nature of failures which may occur.

3- At-least-once semantics: the **RPC** mechanism guarantees that the remote procedure is executed at least partially or once. This is easily achieved by re-requesting remote procedure execution if failures occur. Clearly, at-least-once semantics is appropriate only if executing a remote procedure once has exactly the same outcome as executing the same request multiple times An operation with this characteristic is said *to be* idempotent. An example *of* an idempotent operation is a **read** operation on a data file. Any operation which does not change the state of remote data can be classed as idempotent.

4- Transactional (zero-or-once) semantics: the RPC mechanism guarantees that the remote procedure is either never executed or executed once. This requires the server to keep track of invocation identifiers of all procedures previously executed to avoid duplication. The server must also ensure that state data is either updated permanently by an operation taking it from one consistent state to another, or that it is left in its original state if the operation is aborted. This type of **RPC** is commonly known as **transactional RPC.**

Ensuring that a remote operation is carried out exactly once is difficult under these circumstances. The server needs to distinguish between new requests and duplicate requests for execution of an operation that has been completed already. This state data must remain consistent despite client and server crashes or lost messages in order to ensure at-most-once **RPC** semantics. Many servers do not retain state data and are therefore known as `stateless' servers. In this case, clients are responsible for retaining state data on past requests (e.g. the value of the read-write pointer in file operations). Stateless servers can at best support only at- least-once RPC semantics.

The complexity of error recovery is reduced when it can be assumed that requests are idempotent, since the simpler to implement at-least-once RPC semantics can be implemented effectively. For example, a number of distributed file server protocols (e.g. Sun Microsystems' NFS) are designed to assume that all requests such as file **open, read, write, delete,** etc. exhibit idempotent behaviour (strictly speaking some operations are not idempotent but can be written to exhibit idempotent behaviour). The stateless servers and at-least-once RPC semantics are implemented to avoid the protocol overhead associated with suppressing duplicate requests after recovering from a failure. However, stateless servers do not cope well with concurrent access as concurrency control requires the retention of state data to facilitate locking.

At-most-once RPC semantics can be achieved only if both client and server reliably detect failure conditions. Use of techniques such as sequence control, connection-oriented RPC transport services and retention of state data to maintain mutual state and so on, implies a significant protocol overhead. However, these techniques must be implemented if non-idempotent requests are to be supported successfully.

In the discussion so far, we have assumed that any client can invoke any remote operation on any server via an RPC mechanism. In practice, before a remote operation is executed by a server, they should authenticate each other. This requires a sub-dialogue that is initiated when the server receives a client request. The sub-dialogue may involve the use of an `authentication server' and uses encryption techniques for further security. This requires both the client and server to

maintain state data for security purposes which means that sophisticated security techniques require stateful servers.

4.8 ADVANTAGES OF DISTRIBUTED SYSTEM

The. main advantageş of distributed systems is their ,ability to allow the sharing; of information and resources over a wide geographic area, giving. a systems designer freedom ta optimize the placement of distributed system components such as. data processing. This. then supports: improved flexibility. Çomputers and other IT infrastructure components can flexibility . Located at points within the organization where they can be. utilized most effectively. components can be added upgraded, moved and removed (usually in small increments over a wide range of capacities) without. impacting upon other components to meet present and future needs The. ability of an IT infrastructure or application to grow to meet increasing. user or application demand wlıile , minimizing , disruption is known as scalability.

Local autonomy, by allowing **domains** of **control** to be defined where decisions are made relating to purchasing, ownership, IT budgets, operating priorties, IS development and management, etc. Each, domain decides where .resources (including manpower and .IT infrastructure components). under its control are located. This autonomy is recognition of the distributed nature of many organizational activities. Domains can be *federated* when necessary for mutual benefit.

- **Increased reliability and availability.** In a centralized system, a component (hardware or software) failure can mean that the whole system is down, stopping all users from getting work done. In a distributed system, multiple components of the same type çan .be configured to fail independently and (e.g. through replication) provide a level of fault tolerance. Thus, failure of one component may isolate a group of users, but does not necessarily prevent others from operating.
- **Improved performance.** Large centralized systems can be slow performers due mainly to the sheer volume of data and transactions being handled. A service which is partitioned over many server computers, each supporting a smaller set of applications and users with access to local data and resources, results in faster access (response times). Another performance advantage is the support for parallel access (updates and retrievals) to distributed data across the organization
- **Security breaches are localized.** In distributed system with multiple security control domains, a security breach in one domain does not compromise the whole system. Each security domain has varying degrees of security authentication, access control and auditing.

4.9 DISADVANTAGE DISTRIBUTED SYSTEM

Against the above advantages are a number of disadvantages, most of which correspond to the advantages of centralized systems:

- **It is more difficult to manage and secure.** Centralized systems are inherently easier to secure (offering a single security domain, controlled by a single authority) and easier to manage. Distributed systems require more complex process for security , administration, maintenance and user support due to greater levels of co-ordination and control required. This usually results in higher costs associated with managing and securing distributed systems environment.
- **Reduced reliability-and availability.** In contrast to the potential improvements in reliability and availability discussed above, a centralized system can often offer more controlled physical, operational and environmental conditions borne out of years o development and

improvement. Moreover, a diştributed system consists of many more components which can potentially fail, causing loss of availability.

- **A shortage of skilled support and development staff.** In a decentralized operation scarce support and development can be dispersed, resulting in a loss of economies of scale which, in turn, leads to higher costs. Another issue is the level of support offered b; vendors. The commitment and level of support offered by vendors of distributed software and hardware to `corporate` organizations grappling with the issues of building large-scale distributed systems is not yet comparable to the traditional large mainframe-base vendors. This is due in part to the fact that many vendors are supplying only a small part of the overall system. In practice, whereas in a centralized system the mainframe supplier provided the equipment and integration support as a single package, distributed system implementation commonly involves multiple vendors and therefore requires the skills of systems integrator. Problem isolation and resolution can be much more difficult and costly when multiple vendors are involved. This is exacerbated by the need for staff to develop wide range of knowledge and skills. thus, distributed systems are not a panacea tor all the world computing problemş. Like many innovations they provide a solution to many problems, but also. introduce new. ones. With the potentially significant disadvantages highlighted above why should any organization take the risk of implementing a distributed system? The answer is that although .overcoming ,the disadvantages represents a significant challenge to any organization, it is.. often perceived that the . advantages (which are .generally user-centred) significantly. outweigh. the disadvantages (which are generally centred around Staffing, management and administration). Implementing a distributed IT infrastructure gives an organization the flexibility to choose an appropriate balance between the high levels of discipline and co-ordination associated with ,centralized control, and the flexibility and local autonomy offered by decentralization.A distributed IT infrastructure supports tight management of corporate data by locating it on a limited number .of tightly managed server computers, easily accessible to users via a desktop computer. Data can be duplicated in a controlled manner to improve performance and availability. Thus an appropriate balance between ,control, efficiency; flexibility and local autonomy can be realized.

CHAPTER 5

TYPES OF DISTRIBUTED SYSTEMS

There are several types of distributed processing systems in which the components are hooked together by telecommunications. This chapter categorizes them and gives examples.

5.1 HORIZONTAL. VERTICAL DISTRIBUTION

First we shall distinguish between horizontal and vertical distribution. By vertical distribution we mean that there is a hierarchy of processors the transaction may enter and leave the computer system at the lowest level. The lowest level may be able to process the transaction or may execute certain functions and pass it up to the next level. Some, or all, transactions eventually reach the highest level, which will probably have access to on-line files or data bases.

The machine at the top of a hierarchy might be a computer system in its own right, performing its own type of processing on its own transactions. The data it uses is, however, passed to it from lower-level systems. The machine at the top might be a head-office system which receives data from factory, branch, warehouse, and other systems.

By horizontal distribution we imply that the distributed processors do not differ in rank. They are of equal status-peers-and we refer to them as peer-coupled systems. A transaction may use only one processor, although there are multiple processors available. On some peer-coupled systems a transaction may pass from one system to another, causing different sets of files to be updated.

5.1.1 COOPERATIVE OPERATION?

In some networks the user has a choice of computer systems available to him, but he normally employs only one computer at a time. The computers are programmed independently, and each computer performs its own functions.

In other networks the computers are programmed to cooperate with one another to solve a common set of problems. This is often the case in a vertical system The lower-level machines are programmed to pass work to the higher-level machines. This is sometimes true also in a horizontal system. The processing of one transaction may begin on one machine and pass to another. The different computers perform different functions or maintain and update different files. The machines may be minicomputers in the same location or computers scattered across the world on a network.

5.1.2 FUNCTION DISTRIBUTION VS. SYSTEM DISTRIBUTION

In some distributed systems, usually vertical systems, *functions* are distributed, but not the capability to fully process entire transactions. The lower-level machines may be intelligent terminals or intelligent con- trollers in which processors are used for functions such as message editing, screen formatting, data collection dialogue with terminal operators, security, or message compaction or concentration. They do not complete the processing of entire transactions.

We refer to this distribution as *function distribution* and contrast it with *system distribution* in which the lower-level machines are systems in their own riglıt, processing their own transactions, and occasionally passing transactions or data up the hierarchy to higher level machines.

In *a systems distribution* environment the lower machines may be entirely different from, and incompatible with, the higher machines. In *a function distribution* environment, close cooperation between the lower-level and higher-level machines is vital. Overall system standards are necessary to govern what functions are distributed and exactly how the lower and higher machines form part of a common system architecture with appropriately integrated control mechanisms and software.

5.1.3 COMBINATIONS

Many configurations of the future will be neither purely vertical nor purely horizontal; neither purely homegeneous nor entirely heterogeneous They will be combinations of these, and function distribution and systems distribution will be combined in one configurations The system contains both function distribution and vertical systems distribution.

5.2 FUNCTION DISTRIBUTION

When the peripheral nodes are not self-sufficient systems but perform a function subservient to a higher-level distant computer, we speak of intelligent terminals, *intelligent terminal cluster controllers.* or *intelligent concentrators.* These terms imply a vertical distribution of function in which all or most transactions have to be transmitted, possibly in a modified form, to a higher-level computer system, or possibly to a network of higher-level computer systems.

The centralised teleprocessing system of 1970 employed simple terminals and carried out almost all of its functions in the central computer. At first system control and housekeeping functions were moved out, then functions such as data collection, editing, and dialogue with terminal operators, and finally many of the application programs themselves.

1-In the host computer
2-In a line control unit or front end network control computer

Many functions are necessary to control a terminal network. If the host computer performs all the operations itself, it will be constantly interrupting its main processing, and many machine cycles will be needed for line control. Some of the line control functions may be performed by a separate line control unit. In some systems, all of them are performed by a separate and specialised computer. The proportion of functions which are performed by a line control unit, which by the host computer hard- ware and which by its software, varies widely from system to system. Some application functions could be performed by the subsystem computer-for example, accuracy checking and message logging.

A major advantage of using a front-end network-control computer is that when the host computer has a software crash or brief failure, the network can remain ,functionally operational. Restart and recovery of the network without errors or lost transactions is a tedious and often time-consuming operation, and if it happens often it can be very frustrating to the end users.

3. In the mid-network nodes

The midnetwork nodes or concentrators may take a variety of different forms. They may be relatively simple machines with unchangeable logic. They may have wired-in logic, part or all

of which can be changed by an engineer. They may be microprogrammed. Or they may be stored-program computers, sometimes designed solely for concentration or switching, but sometimes also capable of other operations and equipped with files, high-speed printers, and other input-output equipment.

4.In the terminal control unit

Terminal control units also differ widely in their complexity, ranging from simple hardwired devices to stored program computers with much software Increasingly they are computers with storage units and there is a trend towards greater power and larger storage. They may control one terminal or many. They may be programmed to interact with the terminal operator to provide a psyhologically effective dialogue in which-only an essential kernel is transmitted to or from the host computer. They may generate diagrams on a graphics terminal or interact with the operators use of a light pen . They are often the main component in carrying out the assortment of distributed functions which this chapter will list.

5. In the terminal

Intelligent terminals" are becoming more intelligent. T'heir processing functions range from single operations such as accumulating totals in a system which handles financial transactions, to dialogues with operators involving much programming. Some intelligent terminals do substantial editing of input and output data. Some terminals perform important *security* functions.

Where several terminals share a control unit, F, such functions are probably better performed in the control unit, leaving the terminal a simple inexpensive mechanism in which the main design concern may be tailoring the keyboard and other operator mechanisms to the applications in question.

6. In a "back-end" file or data base management processor

File or data base operations may be handled by a "back-end" processor. This can carry out the specialised functions of data base management or file searching operations. It can prevent interference between separate transactions updating the same data. It can be designed to give a high level of data security protection.

"Back-end" processors, where they exist today, are normally cable-connected to their local host computer. They could, especially when high-bandwidth networks or communications satellite facilities are available, be remote from the host computers which use them.

5.2.1 CHOICE OF FUNCTION LOCATION

The designer, faced with different locations in which he could place functions, may choose his configuration with objectives such as the following:

1. Minimum total system cost. There is often a trade-off between distributed function cost and telecommunications cost.

2. High reliability. The value attached to system availability will vary from one system to another. The systems analyst must evaluate how much extra money is worth spending on duplexing, alternate routing and distributed processing to achieve high availability. On some systems reliability is vital. A supermarket must be able to keep its cash registers going when a communication line or distant host computer fails.

3. Security.

In some systems function distribution is vital for system security (as we discuss later).

4. Psychologically effective dialogue with terminal users. Function distribution is used to make the dialogue fast, effective and error-free.

5. *Complexity*

Excessive complexity should be avoided. The problems multiply roughly as the square of the complexity.

6. Software cost. Some types of function distribution occuring throughout a network incur a high programming expenditure. The use of stored-program peripheral machines may inflate cost.

7. Flexibility and expandability.

It is necessary to choose hardware and software techniques that can easily be changed and expanded later, especially because telecommunications and networking technology are changing so fast. Some approaches make this step difficult.

5.3 REASONS FOR FUNCTION

The following lists the main reasons for function distribution. They fall into three categories:

1. ***Reasons associated with the host***

Many machine instructions are needed to handle all of the telecommunications functions. The load on a central machine could be too great if it had to handle all of these functions. A single computer operates in a largely *serial* fashion executing one instruction at a time. It seems generally desirable to introduce parallelism into computing so that the circuits execute many operations simultaneously. This is the case when machine functions are distributed to many small machines.

2. Reasons associated with the network

There are many possible mechanisms which can be used to make the network function efficiently. We will discuss them later in the book. These mechanisms are used to lower the overall cost of transmission and increase its reliability. The network configuration is likely to change substantially on most systems, both because of application development and increasing traffic, and because of changes in networking technology which are now coming at a fast and furious rate. Function-distribution may be used to isolate the changing network from other parts of the so that the other parts do not have to be modified as the network changes.The term transparency is used to imply that changes which occur in the network be not evident to and not affect the users.

5.3.1 Reasons for function distribution

part1:

1. Psychologlcally Effectlve Dialogues

Local interaction.

Much of the dialogue interaction takes place locally rather than being transmitted, and hence can be designed without concern for transmission constraints.

· **Local panel storage.**

Panels or graphics displayed as part of the dialogue can be stored locally.

· **Speed**

Local responses are fast. Time delays which are so frustrating in many terminal dialogues can be largely avoided. The delays that do occur when host response is needed can be absorbed into the dialogue structures.

2. *Reduction of Telecommunications Costs*

· **Reduction of number of messages.**

In many dialogues the number of messages transmitted to and fro can be reduced by an order of magnitude because dialogue is carried on within the terminal or local controller.

· **Reduction of message size.**

Messages for some applications can be much shortened because repetitive information is transmitted.

· **Reduction of number of line turnarounds.**

Because the number of messages is reduced; and because a terminal cluster controller or concentrator can combine many small messages into one block for transmission.

· **Bulk transmission.**

Nontime-critical items can be collected and stored for later batch transmission over a switched connection.

· **Data compaction.**

There are various ways of compressing data so that fewer bits have to be transmitted. This effectively increases the transmission speed.

· **Minimum cost routing.**

The machine establishing a link could attempt first to set up a minimum-cost connection, e.g., a corporate tie-line network. If these are busy it could try progressively more expensive connections (e.g., WATS, direct distance dialing).

· **Controlled network access.**

Terminal users may be prevented from making expensive unauthorised calls.

3 . *Reliability*

· **Local autonomy.**

A local operation can continue, possibly in a fallback mode (using a minimal set of functions), when the location is cut off from the host computer by a circuit, network, or host failure. On certain systems this is vital.

· **Automatic dial backup.**

A machine may be able to dial a connection if a leased circuit fails.

· **Automatic alternate routing.**

A machine may be able to use an alternate leased circuit or network path when a network failure occurs.

· **Control procedures.**

Control procedures can be used to recover from errors, os failures and to ensure that no messages are lost or double-processed.

· **Automatic load balancing**.

A machine may be able to dial an extra circuit or use a different computer to handle high traffic peaks.

4. *Less Load on Host*

Parallel operations.

The parallel operation of many small processors relieves the host computer of much of its work load, and lessens the degree of multiprogramming. In some systems this is vital because the host is overburdened with data base operations.

Permits large numbers of terminals.

Some systems require too many terminals for it to be possible to connect them directly to a host computer. Distributed control and operations make the system possible.

5. Fast Response Times

Process mechanisms.

Local controllers can read instruments rapidly and give a rapid response to process control mechanisms when necessary.

Human mechanisms.

Fast reaction is possible to human actions such as the use of a plastic card or the drawing of a curve with a light pen.

· **Dialogue response times.**

Dialogues requiring fast response times (such as multiple menu selection) can be handled by local controllers.

6. Data Collection

· **Data entry terminals.**

Many inexpensive data entry terminals (for example, on a factory shop floor) can be connected to a local controller which gathers data for later transmission.

· ***Local error checking.***

Local checks can be made on the accuracy or syntax of terminal entries. An attempt is made to correct the entries before transmitting them to the host.

· **Instrumentation.**

Local controllers scan or control instruments, gathering the result for transmission to a host computer.

7. More Attractive Output

· **Local editing.**

Editing of output received at terminals can lay out the data attractively for printers or screen displays. Repetitive headings, lines, or text, and page numbers can be added locally. Multiple editing formats can be stored locally.

8. Peaks

· Interactive and real-time systems often have peaks of traffic which are difficult or expensive to accommodate without function distribution. Storage at the periphery allows the peak transactions to be buffered or filed until they can be transmitted and processed economically.

9. Security

· **Cryptography.**

Cryptography on some systems gives a high measure of protection from wire- tapping, tampering with magnetic-stripe plastic cards, etc. Cryptography is vital on certain electronic fund transfer systems.

· **Access control.**

Security controls can prevent calls from unauthorized sources from being accepted, and prevent terminals from contacting unauthorized machines.

10. Network Independence

· **Network transparency.**

Programmers of machines using networks should not be concerned with details of how the network functions. They should simply pass messages to the network interface and receive messages from it.

· **Network evolution.**

As networks grow and evolve, and as different networks are merged, programs in machines using the networks should not have to be rewritten.

· **New networks.**
Network technology is changing fast. As applications are switched to new types of networks (e.g:, DDS, value-added networks, Datadial, satellite networks), the programs in the using machines should not have to be rewritten.

11. Terminal Independence

· **New terminals.**
Terminal design is changing fast. If a new terminal is substituted, the old pro- grams should not have to be rewritten. Software in terminal controllers may make the new terminals appear like the old.

· **Virtual terminal features.**
Application programs may be written without a detailed knowledge of the terminal that they will use. For example, the screen size or print-line size may not be known. The programmers use specified constraints on output, and the distributed- intelligence mechanisms map their output to the device in question.

Mechanisms relating to the network may reside in any of the locations .A terminal or a controller for a cluster of terminals may have mechanisms intended to minimise the transmission cost. A front-end communications processor may relieve the host of all network functions, and maintain network operations without loss of data if the host or its software fails. Intelligence may also reside in midnetwork nodes such as packet-switching devices, concentrators, intelligent ex- changes, or telephone company equipment in systems such as AT&T's ACS. The phrase "intelligent network" is increasingly used to imply that the network itself uses computers to share transmission links or other resources in an efficient, dependable manner.

3. Reasons associated with the end user

Probably the most important of the three categories is that associated with the end user. On many systems built prior to the era of function distribution, the dialogue that takes place between the terminal and its operator is technically crude. It is often difficult for the user to learn, and clumsy and frustrating in operation. The user is forced to learn mnemonics and to remember specific sequences in which items must be entered. The response times are often inappropriate. The majority of the users who should be employing terminals are unable to make the machines work, and generally discount the possibility of ever using them because they perceive them as being difficult-designed for technicians, programmers, or a specially trained and dedicated staff. One psychologist describes many of these user-terminal interfaces as "unfit for human consumption."

In the past there has been good reason for the crudity of terminal dialogues. The terminals had no intelligence. Every character typed and displayed had to be transmitted over the network. The network often used leased voice lines serving many terminals, and to minimize the network cost, the number of characters transmitted was kept low. The response times were often higher than psychologically appropriate because of the queries on the lines.

With intelligent terminals or controllers the dialogue processing can take place in the local machine. Most of the characters are not transmitted over the telephone lines. The only characters transmitted are those which take essential information to the central computer and carry back essential information to the terminal. These characters will often be only a small fraction of the total characters typed and displayed in a psychologically effective dialogue.

Much of the future growth of the computer industry is dependent on making the machines easy to use and understand for the masses of people in all walks of life who will employ them, and distributed intelligence can play a vital part in this.

5.4 HIERARCHICAL DISTRIBUTED PROCESSING

So far this chapter has discussed *function distribution* in which the peripheral machines are not self-sufficient when isolated from their host by a telecommunications or other failure. Now let us expand the discussion to *processing distribution* in which the peripheral processors keep their own data and can be self- sufficient, but which are connected to higher level systems.

There is not necessarily a sharp boundary line between function distribution and system distribution. In some cases there has tended to be growth from function distribution to system distribution, with more and more power being demanded in peripheral machines. In other cases the peripheral machines started as standalone minicomputers and became linked into a higher-level system.

The application programming steps for most (but not all) commercial transactions do not require a large computer. Small, inexpensive, mass-produced processors such as those discussed in the previous chapter could usually handle the whole transaction. They would handle it with a much smaller software path length than a large computer. The difference in software path length greatly reinforces the arguments about there no longer being economies of scale. Some large mainframes with complex data base management systems more than 100.000 software instructions per transaction and only a few thousand application instruction per transaction.

In some cases there criteria does not apply, then the transaction reguires centrally. In other cases the data also can be kept in storage attached to the local machine.

As we commented earlier, criteria for determining whether a transaction ıs transmitted could be

1.It needs the power a large computer
2.It needs data which are stored centrally

If one of these criteria does not apply, ten the transaction is processed locally. Most commercial transaction and many scientific calculations do not need the power of a large computer.. there are exceptions such as simulations and complex models. Many of these exceptions would not use the teleprocessing anyway. But the second criterion-centralized data-is important to some , but not all data. Consequently data base and data communications techniques are closely related, end computer manufacturers produce *data base data communications (DBDC)* software.

5.4.1 EXAMPLES OF HIERARCHICAL CONFIRATIONS

Some examples of hierarchical configurations as follows:

1. Insurance

The branches of an insurance company each have their own processor with a printer and terminals. This processor handles most of the computing requirements of the branch. Details of the insurance contracts made are sent to a head office computer for risk analysis and actuarial calculations The head-office management has up-to-the- minute information on the company's financial position and exposure, end can adjust the quotations given by the salesmen accordingly

2. A chain store

Each store in a chain has a minicomputer which records sales and handles inventory control and accounts receivable. It prints sales slips (receipts) for customers at the time of sale. Salesman and office personnel can use the terminals to display pricing, inventory and accounts receivable information and customer statement the store management can display salesman performance information and goods aging and other analysis reports.

The store systems transmit inventory and sales information to the head office system. At night they receive inventory change information. The fast receipt of inventory and sales

information enables the head office system to keep the inventory of the entire organization to a minimum.

The store systems run unattended. Any program changes are transmitted to the systems from the head office computer.

3. Production control

Various different production departments in a factory complex each have a minicomputer. Work station terminals on the shop floor are connected to the minicomputer and the workers enter details of the operations they perform. The task of scheduling the operations so as to make the best utilisation of men and machines is done by the minicomputer. The shop foreman displays these operations schedules and often makes changes to them because of local problems and priorities. He frequently makes a change and instructs the machine to reperform its scheduling program.

Details of the work to be done are made up by a higher-level computer which receives information about sales and delivery dates, and performs a gross and net breakdown of the parts that must be manufactured to fill the orders. The central computer passes its job requirements to the shop floor minicomputers, and receives status reports from them.

5.4.2 PROCESS CONTROL

Hierarchies of processors were common in process control applications before ,they were used in commercial data processing. Many instruments taking readings in an industrial or chemical process are connected to a small reliable computer which scans the readings looking for exceptions or analysing trends. The same computer may automatically control part of the operation, setting switches, operating relays, regulating temperatures, adjusting values, and so on. Response time must be fast on some process control applications. A local mini- computer is used to ensure fast response. Increasingly today, tiny cheap microprocessors are being employed in instruments and control mechanisms. Many such devices may be attached to a minicomputer which stores data relating to the process being controlled.

A higher-level computer may be concerned with planning the operations, optimisation, providing information for management control, or general data processing. .

In hospitals, the elaborate patient instrumentation used in intensive-care wards is monitored and controlled by small, local and highly reliable computers. These in turn are linked to higher-level machines which can perform complex analyses, provide information to stations, record patient histories, and so on.

5.4.3 CAUSALLY COUPLED?

In some configurations the design of the peripheral systems is largely independent of the design of the higher level systems. In others the periphery and the center are so closely related that they are really separate components of the same system.

An example of a *causally coupled* configuration is a corporate head-office *information system* which derives its data from separate systems, separately installed in different corporate departments. These systems transmit data at the end of the day to the control system where it is

edited, reformatted, and filed in a different manner to that in the peripheral systems, to serve a different purpose. The installers of the peripheral systems designed them for their own needs and were largely unaware of the needs of the central system.

An example of a closely coupled design is a banking system in which all customer data is stored by a central computer. (This does not apply to all banks. Some have loosely distributed systems.) A small computer in each branch, or group of branches, serves the processing needs of that branch, providing the tellers and the officer with the information they need at the terminals. Customer data is also stored in the branch computers, largely in case of a failure of the central system or the telecommunications link to it. The peripheral files are strictly subsets of the central file. The programs developed for the peripheral computers are compiled on the central computer, and loaded from it into the peripheral computers. Changes in the peripheral programs are made centrally and transmitted. Account balancing requires tight cooperation of the peripheral and central machines.

5.4.5. MULTIPLE LEVELS

Vertically distributed configurations many contain more than two levels of processor The lowest level may consist of intelligent terminals for data entry, or microprocessors in a factory, scanning instruments.

The next level may be a computer in a sales region assembling and storing data that relates to that region, or a computer in a factory assembling the data from the microprocessors and being used for production planning.

The third level is a conventional large computer system in the divisional head office, performing many types of data processing and maintaining large data bases for routine operations. This computer center receives data from the lower systems and sends instructions to them.

The highest level is a corporate management information system, with data structured differently from that in the systems used for routine operations. This system may be designed to assist various types of high-management decision making. It may run complex corporate financial models or elaborate programs to assist in optimising certain corporate operations, for example, scheduling a tanker fleet. It receives summary data from other, lower systems.

5.4.6 REASONS FOR HIERARCHIES

Reasons for using hierarchical systems distribution are summarised in part.2 The set of reasons should include those in part 1 on function distribution.

An important group of reasons on some configurations is related to data where it is kept and how it is maintained. Also of great importance are arguments relating to human, political, and organisational reasons, in addition to technical .reasons .

5.5 HORIZONTAL DISTRIBUTION

So far we have discussed vertically distributed systems. Now we will consider horizontal distribution.

Some software, control mechanisms and system architectures are primarily oriented to vertical distribution, and some are primarily for peer-coupled systems. A transport subsystem which merely transmits data between computers *could* be designed to serve a horizontal or vertical configuration equally well. The differences are more important in the higher-level activities such as file management, or data base management, intelligent terminal control, data compression, editing, man-machine dialogues, recovery, restart, and so on.

In reality, major differences are found in the transport subsystems also. A transport subsystem designed for vertical distribution can have simpler flow control and routing control mechanisms, and hence simpler recovery procedures. It may use elaborate concentrators or other devices for maximising network utilisation, and may employ some of the function distribution features listed the following below.

Part2 Technical reasons for using hierarchical distributed processing

There are also human, political, and organisational reasons which are often more important than these technical reasons.

· **Cost.**
Total system cost may be lower. There is less data transmission and many functions are moved from the host machine.

· **Capacity.**
The host may not be able to handle the workload without distribution. Distribution permits many functions to be performed in parallel.

· **Availability**.
Fault tolerant design can be used. Critical applications continue when there has been a host or telecommunications failure. The small peripheral processors may be substitutable. In some systems high reliability is vital; e.g., a supermarket system, or hospital patient monitoring.

· **Response time.**
Local responses to critical functions can be fast; no telecommunications delay; no scheduling problems; instruments are scanned and controlled by a local device.

· **User interface**.
A better user interface can be employed, e.g., better terminal dialogue, when the user interacts with a local machine; also better graphics or screen design; more responses, faster response time.

· **Simplicity.**
Separation of the peripheral functions can give a simpler, more modular system design.

· **More function.**
More system functions are often found because of ease of implementing them on the peripheral machines. Salary savings often result from increased peripheral functions.

· **Separate data organizations.**
The data on the higher-level system may be differently organized from those on the peripheral systems (e.g., corporate management information organized for spontaneous searching versus local detailed operational data tightly organized for one application).

.

Part3 Reasons for horizontal computer networks

· **Resource sharing.**
Expensive or unique resources can be shared by a large community of users, as on **ARPANET.**

· **Diversity.**
Users have access to many different computers, programs, and data banks.

· **Transaction interchange**

Transactions are passed from one system to another or from one corporation to another: e.g., financial transactions passed between banks on SWIFT; airline reservations or messages passed between computers in separate airlines, as on SITA.

· **Separate systems linked.**
Separate previously existing systems are linked so that one can use another's data or programs, or to permit users to access all of them.

· **Local autonomy.**
Local autonomous minicomputer systems are favored, with their own files, and some transactions need data which reside on the file of a separate system.

· **Functional separation.**
Instead of one computer center performing all types of work, separate centers specialise in different types. For example, one does large-scale scientific computation. One does information retrieval. One has a data base for certain classes of application. One does mass printing and mailing.

· **Transmission cost.**
Separate systems share a common network designed to minimise the combined data (and possibly voice) transmission cost.

· **Reliability and security.**
When one system fails, others can process transactions. If one system is destroyed, its files can be reconstructed on another.

· **Load sharing.**
Unpredictable peaks of work on one machine can be off-loaded to other machines.

· **Encouragement of development.**
A corporate network can permit small data processing groups to develop applications.

5.6 PATTERNS OF WORK

Because of the mechanisms built into software or systems architecture, designers sometimes try to make all configurations vertical, or all configurations horizontal. This can result in excessive overhead, system inflexibility, or clumsy control. Whether or not a configuration *should* be vertical, or horizontal, or both, depends upon the *patterns of work* the configuration must accomplish and the *patterns of data usage.*

In designing a distributed system we are concerned with such questionş as:

· Where are the units of processing work required?
· How large are these units? What size of processing machine do they need?
· Are the units independent, or does one depend on the results of another?
· What stored data do the work units employ?
· Do they share common or independent data?
· What transactions must pass between one unit and another? What are the patterns of transaction flow?
· Must transactions pass between the units of work immediately, or is a delay acceptable? What is the cost of delay?

The answers to these questions differ from one organisation to another. The patterns of work are different. The patterns of information flow between work units are different. Different types of corporations tend, therefore, to have their own natural shapes for distributed processing. What is best for an airline is not necessarily best for an insurance company.

The nature of the work units may be such that they can be independent of one another and have no need to know what each of the others is doing. They may be standalone units having no communication with any other unit-possibly standalone minicomputers. On the other hand, they may need to share common data which resides centrally. In this case there are vertical links to a

common data store. there may be multiple such data stores which themselves pass information to a higher system. Alternatively the work units at one level may be such that they need to pass information to other units at the same level. This situation may lead naturally to horizontal communication; but it could also, if necessary, be handled vertically with a centralized processor relaying transactions between the units.

EXAMPLES

1. An airline reservation system requires a common pool of data on seat availability. Geographically scattered work units use, and may update, the data in this pool. Each of them needs data which is up-to-date second by second. This data needs to be kept centrally. The bulkiest data are those relating to passengers. A passenger may telephone the airline in cities far apart; when he does so the agent to whom he talks must be able to access needed data. In order to rind the data it is easier to keep it centrally also.

2. A car rental firm may permit its customers to pick up a car at one location and leave it at another. When the car is picked up a computer terminal prepares the contract. When the car is left a terminal is used to check the contract and calculate the bill. If a minicomputer at each location performed these functions, horizontal communication would be needed between the destination location and the location where the car was picked up. However, some centralized work is also needed because it is necessary to keep track of the company's cars and ensure that they are distributed appropriately for each day's crop of customers. Credit and other details about regular customers may also be kept centrally.

The shape of the work therefore indicates both vertical and horizontal distribution. However, because the centralised (vertical) links are needed, the customer contracts may also be kept centrally and the same links used to access them. The rental offices may then use intelligent terminals rather than complete minicomputers.

3. Insurance companies have offices in different locations. They keep details about customers and their policies. An office does not normally need to share these data with another office or pass transactions to it. The offices could therefore use standalone machines. Customers in different locations may have different requirements. In the U.S. different states have different insurance regulations and tax laws. The different machines may therefore be programmed somewhat differently. The insurance company's head office, however, needs to know enough details of all customers policies to enable it to evaluate the company's cash flow, and risks, and to perform actuarial calculations which enable it to control the company's financial exposure. enough data for this purpose is therefore passed upwards to the head office. This vertical communication does not need to be real-time, as in the case of an airline reservation system. It can be transmitted in periodic batches.

Although the pattern of the work in an insurance company is appropriate for a decentralized system, that does not necessarily mean that a decentralized system will be the cheapest or best. There are various arguments for centralization, among them economies of scale, centralized control of programming, and use of data base software. A function-distribution rather than a processing-distribution configuration is used in some insurance companies.

4. In a group of banks, each handles its own customers with its own data processing system. A customer in one bank, however, can make monetary transfers to customers in other banks. A network is set up by the banks to perform such transfers electronically. The money is moved very rapidly and hence is available for use or interest-gathering by banks for a longer period. The use of this "float" more than pays for the network. In this example we have a peer-coupled configuration with need for a horizontal transfer between the work units.

5.7 DEGREE OF HOMOGENEITY

We may classify horizontal configurations according to the degree of homogeneity of the systems which communicate. This affects the design, the choice of software and network techniques and, often, the overall management.

At one extreme we have identical machines running the same application programs in the same corporation. In other words the processing load has been split between several identical computers. At the other extreme we have incompatible machines running entirely different programs in different organisations, but nevertheless interconnected by a network. One of the best known examples of this is ARPANET.

5.8 NONCOOPERATIVE SYSTEMS

We may subdivide confirations into those composed of cooperative and noncooperative sysems. A noncooperative configuration consists of computer systems installed indepently by different authorities with no common agency controlling their design but linked by a common shared network.

When the networking capability becomes accepted and understood by the various system development groups, there may be slightly less noncooperation. Developers know that a certain data base exits on the another system.They may leran to link in terms of interchanging data,sharing resources and estaslishing compatible transaction formats.

Because the cost and ease of networking will improve greatly in the future, some corporations have attempted to impose certain standards on their diverse systems groups, which will eventually make interconnections of the systems more practical or more valuable. Among the types of standards imposed or attempted have been the following

1, Standardisation of transaction formats.
2. Standardisation of line control discipline.
3. Use of compatible computers (one large corporation decreed that all minicomputers should be DEC machines, possibly anticipating future use of DEC's network architecture).
4. Standardisation of data field formats and use of an organisation-wide data dictionary.
5. Standardisation of record or segment formats.
6. Use of a common data description language (e.g.. CODASYL DDL, or IBM's Dl/I n
7. Use of a common data base management software.
8. Use of a common networking architecture.

5.9 COOPERATING SYSTEM

Cooperating systems are designed to achieve a common purpose, serve a single organisation, or interchange data in an agreed-upon manner. We can subdivide cooperating systems into those in which the separate systems are used by the same *organisation* and those in which *separate corporations* are interlinked. Networks which interlink separate corporations are found today in certain industries. In the future they may become common in most industries to bypass the labor- intensive steps of mailing, sorting, and key-entering orders, invoices, and other documents which pass from a computer in one organisation to a computer in another.

Industries with intercorporate computer networks today include banking and airlines. Most major airlines have reservation systems in which terminals over a wide geographic area are

connected to a central computer. Worldwide airlines have world- wide networks. Many booking requests cannot be fulfilled completely by the airline to which they were made. The airline might have no seats available, or the journey may necessitate flights on more than one carrier. Booking messages therefore have to be passed from the computer in one airline to the computer in another, and often the response is passed back swiftly enough to inform the booking agent who initiated the request at his terminal. In order to achieve this linking of separate systems all participating airlines must agree to a rigorously defined format for the messages passing between the airlines. This format is standardised by an industry association, ATA in the United States and IATA internationally. To operate the interlinking network, the air- lines set up independent nonprofit organisations, ARINC (Aeronautical Radio Incorporated) in the U.S., and SITA internationally (Societe International de Telecommunications Aeronautique). The separate airlines must send ATA- or IATA-format messages using the ARINC or SITA protocols. These networks began as networks for sending low-speed off-line teleprinter messages. As the need arose they were upgraded to handle fast-response messages between computers as well as conventional teleprinter traffic.

5.10 SYSTEMS UNDER ONE MANAGEMENT

Much of the use of distributed computing is within one corporation under one management. This could result in a compatible configuration using a common networking architecture. Often, however, the systems to be linked were installed separately in separate locations without any thought about eventual interconnection. The files or data bases are incompatible; the same data field is formatted differently in different systems; programs cannot be moved from one computer to another without rewriting; where teleprocessing is used the terminals are incompatible; and even the line control procedures are different so the terminals cannot be changed without a major upheaval in the systems they are connected to. In this environment a major reprogramming and redesign effort is needed before networking becomes of much value, and often this effort is too expensive.

It is necessary that systems in different functional areas of a corporation be developed by different groups.Corporate data processing is much too complex for one group to develop more than a portion of it.The current trend to to decentralization is resulting in more and more autonomous groups carring out application development.This a valuable trend because it results in more people involved in application development,and the development being done locally where the application problem are understood.

5.11 INTERFACES

In order to make computer networking of value, it is desirable that the interfaces between the separately developed systems be rigorously defined and adhered to. If the interfaces are preserved, each development group can work autonomously. There are several levels of interface:

1. Interface to the transport subsystem which permits blocks of data to be moved between distant machines. This interface can be defined independently of the application or the firms which use the network.

2. Interfaces for the software services which are external to the transport subsystem but not part of the application programs; for example software for remote file access, compaction, conversion, cryptography, setting up sessions, editing messages, and so on.

3. Applications interfaces defining what transaction types are interchanged between different application systems. These can be defined independently of the choice of networking software or hardware.

Interface 1, is provided by some common carrier systems for computer networking (the CCITT X.25 standard, for example). Interfaces 1 and 2 are provided by some of the manufacturers' protocols for computer networks and distributed processing (for example IBM's and DEC's architectures for networks). Interface 3, is usually up to the systems analysts. A typiçal transaction would be given a rigorously defined format. When they are transmitted between machines, data would be in the format with additional headers and a trailer prescribed by interfaces 1 and 2.

As changing costs take the computer industry increasingly toward distributed processing, one highly desirable characteristic is portability of programs. Programs should be capable of being moved from one processor to another and gaining access to distributed data instead of centralised data. There are arguments for, and against, distributed processing, and there are many possible distributed configurations. It is advantageous for a manufacturer's product lines to possess the flexibility to change system configurations without the need to rewrite programs.

The interfaces and protocols that are desirable for distributed processing make the software complex, as we shall see. Furthermore there are so many different configurations, functions, machines, operating systems, access methods and data base management systems that need to be supported that it will be years before the software for distributed systems can do everything that is theoretically desirable. New machines, operating systems, and other software will increasingly be designed to plug into the rigorously defined architectures for distributed systems.

Computer networks and distributed processing are a vitally important and fundamental step in the growth of the computing and telecommunications industries. There is a long road ahead, and the journey will take years to come.

CONCLUSION

The brief survey of trends in applications, computer architecture, and networking suggests a future in which parallelism pervades not only supercomputers but also workstations, personal computers, and networks. In this future, programs will be required to exploit the multiple processors located inside each computer and the additional processors available across a network. Because most existing algorithms are specialized for a single processor, this situation implies a need for new algorithms and program structures able to perform many operations at once. *Concurrency* becomes a fundamental requirement for algorithms and programs.

This survey also suggests a second fundamental lesson. It appears likely that processor counts will continue to increase perhaps, as they do in some environments at present, by doubling each year or two. Hence, software systems can be expected to experience substantial increases in processor count over their lifetime. In this environment, *scalability* resilience to increasing processor counts is as important as portability for protecting software investments. A program able to use only a fixed number of processors is a bad program, as is a program able to execute on only a single computer. Scalability is a major theme that will be stressed throughout this book.

Computers capable of executing several instructions or processing several data items simultaneously are referred to as parallel. Parallism can be achived by using multiple copies of a processor such as an ALU or CPU, or by designing the system in the form of a multistage pipeline, or by a combination of these approaches. The motivations for parallel processing are to increase throughput beyond what is possible with sequential computers and also, in some instances, to enhance flexibility and reliability.

The performance of a parallel processor is difficult to analyze since it depends, often in compex ways,. on system architeture and program organisation. A sipmle numerical measure of performance is the speed up, defined as the ratio of the execution time of a particular task on a system whose degree of parallelism is one.

A single computer system containig more than one CPU is called a multiprocessor. A multiprocessor has higher potential troughput and reliability than the corresponding uniprocessor. Multiprocessors are distinguished by the way the processors communicate with one other. This communication may be tightly coupled via shared memory, or loosely coupled via messages transmitted between the processors` IO subsystems. Multiprocessors also have special programming requirements to permit a programmer to indicate groups of instructions that can be executed in parallel, and to specify communication between loosely coupled processors. It is generally difficult to redesign (parallelize) sequential code to run efficiency on a multiprocessor.

Multiprocessors have been designed around a variety of interconnection networks, of which the (multiple) shared bus is perhaps the easiest to implement, and therefore the most common for smaller multiprocessors. Recent advances in VLSI technology have made it feasible to construct massively parallel distributed-memory machines based on such interconnection structures as (statis) meshes, trees, and hybercubes. These machines exhibit various tradeoffs between processor troughput,

communication delays, and programming compexity. They can also be charecterised by their ability to simulate other interconnection structures efficiently.

Multiple processors are often include in a system with the primary goal of increasing its fault tolerance. This goal is achived by some form of redundancy such as the presence of multiple copies of (critical) hardware components like the CPU. An example of this n-modular redundancy (nMR) in which n copies of a component are present, along with voter circuits to determine the correct signals in the presence of faults. Fault tolerance may alos be obtained by providing facilities to detect and diagnos faults, to eliminatethem from the system by reconfiguration or repair, and to institude recovery by restoring thesystem to normal fault-free operation. This dynamic approach requires that the components suject to failure be at least duplicated as, for insatance, a duplex system. A computer`s degree of fault can be measured by its reliability, its availability, and its mean times to failure and repair.

The first chapter has introduced four desirable attributes of parallel algorithms and software: concurrency, scalability, locality, and modularity. *Concurrency* refers to the ability to perform many actions simultaneously; this is essential if a program is to execute on many processors. *Scalability* indicates resilience to increasing processor counts and is equally important, as processor counts appear likely to grow in most environments. *Locality* means a high ratio of local memory accesses to remote memory accesses (communication); this is the key to high performance on multicomputer architectures. *Modularity* the decomposition of complex entities into simpler components is an essential aspect of software engineering, in parallel computing as well as sequential computing.

The multicomputer parallel machine model and the task/channel programming model introduced in this chapter will be used in subsequent discussion of parallel algorithm design, analysis, and implementation. The *multicomputer* consists of one or more von Neumann computers connected by an interconnection network. It is a simple and realistic machine model that provides a basis for the design of scalable and portable parallel programs. A programming model based on *tasks* and *channels* simplifies the programming of multicomputers by providing abstractions that allow us to talk about concurrency, locality, and communication in a machine-independent fashion, and by providing a basis for the modular construction of parallel programs.

In the second chapter, we have described a four-step approach to parallel algorithm design in which we start with a problem specification and proceed as follows:

1. We first *partition* a problem into many small pieces, or tasks. This partitioning can be achieved by using either domain or functional decomposition techniques.
2. Next, we organize the *communication* required to obtain data required for task execution. We can distinguish between local and global, static and dynamic, structured and unstructured, and synchronous and asynchronous communication structures.
3. Then, we use *agglomeration* to decrease communication and development costs, while maintaining flexibility if possible.
4. Finally, we *map* tasks to processors, typically with the goal of minimizing total execution time. Load balancing or task scheduling techniques can be used to improve mapping quality.

We have also provided *design checklists* that can be used to evaluate designs as they are developed. These informal questions are intended to highlight nonscalable or inefficient features in designs.

Successful application of this design methodology, produces one or more parallel algorithms that balance in an appropriate fashion the potentially conflicting requirements for concurrency, scalability, and locality. The next stage in the design process is to consider how such algorithms fit into the larger context of a complete program.

It was shown how the Branch-and-Bound paradigm may be expressed formally and have shown several ways in which Branch-and-Bound algorithms can be implemented on a parallel machine. As it was dicussed anomalies in both speed up and efficiency and have shown that both of these do occur in real problems. Results from further experiments have shown the overheads of sychronisation and the dangers of not following the priority function closely.

In the fourth chapter the basic concepts of distributed processing have been explored. Six models for structuring a distributed system were identified: master-slave, client/server, peer-to-peer, group, multimedia stream and the distributed object model. The issues of identifying processes, remote **IFC,** process synchronization and remote operations are all fundamental to the successful implementation of a DIS. The message passing and RPC mechanisms provide the basic techniques for implementing a distributed system according to a specific model (e.g. the' client/server model). The distributed object model provides an alternative paradigm for designing and implementing distributed systems. An object is a natural unit of distribution. Implementing a DIS using these mechanisms, however, is very challenging for an application developer. In practice much of the requirements can be specified in an interface definition language from which much of the required program code can be generated automatically by pre-processors much in the same way as visual programming environments which are available for more conventional environments. Message passing and RPC remain the most common underlying (low-level) mechanisms for sup- porting inter-process and inter object interactions.

İn this project have described the main driving forces responsible for the evolution of distributed information systems implemented over a distributed IT infrastructure. The main challenge to organizations is how to utilize this new approach to build systems which deploy informaton technology mpore effectively to support operational and strategic business goals,and provides a more formal definition of a DIS and underlying distributed IT infrasucture

Distributing data introduces a number of new complex problems such as distributed transaction recovery, distributed query processing and the need to hide complexity from the users.and this project identified the characteristic of distributed information system, distributed IT infrastructures .The notion of a DIS as an information system with distributed components leads to the identification of several key distributed services.

REFERENCES

BOOKS

Advances in Parallel Algorithms : Lydia KRONJO, Dean SHUMSHER.

Computer Architecture and Organisation : John P.HAYES.

Distributed Systems – Software Design and Implementation : Albert FEISCHMANN.

Computer Networks and Distributed Processing : Sames MARTIN.

Distributed Information Systems : Errol SIMON.

Principles of Distributed Systems : M. Tamer OZTURK / Patrick VALDURIES.

WEB PAGES

http://www.ieee.org.

http://www.cs.reading.ac.uk.

http://ptools.org.