High Performance Cluster Computing

VOLUME 2

Programming and Applications

Rajkumar Buyya

High Performance Cluster Computing:

Programming and Applications, Volume 2

Edited by
Rajkumar Buyya
(rajkumar@dgs.monash.edu.au)

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

Prentice Hall PTR
Upper Saddle River, New Jersey 07458
http://www.phptr.com

Library of Congress Cataloging-in-Publication Data

High performance cluster computing / edited by Rajkumar Buyya.
p. cm.
Includes bibliographical references and index.
Contents: v. 2. Programming and applications.
ISBN 0-13-013785-5 (v.2)
1. High performance computing. I. Buyya, Rajkumar
QA76.88.H489 1999
004'.3--DC21 99-17906
CIP

Editorial/Production Supervision: *Joan L. McNamara*
Acquisitions Editor: *Greg Doench*
Editorial Assistant: *Mary Treacy*
Marketing Manager: *Bryan Gambrel*
Cover Design Director: *Jerry Votta*
Cover Designer: *Anthony Gemmellaro*
Cover Illustration: *Rob Colvin (The Stock Illustration Source)*
Manufacturing Manager: *Alexis R. Heydt*

© 1999 Prentice Hall PTR
Prentice-Hall, Inc.
Upper Saddle River, New Jersey 07458

Prentice Hall books are widely used by corporations and government agencies for training, marketing, and resale. The publisher offers discounts on this book when ordered in bulk quantities.
For more information, contact: Corporate Sales Department, Phone: 800-382-3419;
FAX: 201-236-7141; email: corpsales@prenhall.com
Or write: Corp. Sales Dept., Prentice Hall PTR, 1 Lake Street, Upper Saddle River, NJ 07458

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

All product names mentioned herein are the trademarks of their respective owners.

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 0-13-013785-5

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall (Singapore) Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

the structure of either the application or the data and both types may exist in different parts of the same application. Parallelism arising from the structure of the application is named as functional parallelism. In this case, different parts of the program can perform different tasks in a concurrent and cooperative manner. But parallelism may also be found in the structure of the data. This type of parallelism allows the execution of parallel processes with identical operation but on different parts of the data.

1.7.1 Choice of Paradigms

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

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

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

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

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

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

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

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

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

Chapter 1

Parallel Programming Models and Paradigms

Luís Moura e Silva† and Rajkumar Buyya‡

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

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

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

1.1 Introduction

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

Scalable computing clusters, ranging from a cluster of (homogeneous or heterogeneous) PCs or workstations, to SMPs, are rapidly becoming the standard platforms for high-performance and large-scale computing. The main attractiveness of such systems is that they are built using affordable, low-cost, commodity hardware (such

as Pentium PCs), fast LAN such as Myrinet, and standard software components such as UNIX, MPI, and PVM parallel programming environments. These systems are scalable, i.e., they can be tuned to available budget and computational needs and allow efficient execution of both demanding sequential and parallel applications. Some examples of such systems are Berkeley NOW, HPVM, Beowulf, Solaris-MC, which have been discussed in Volume 1 of this book [8].

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

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

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

1.2 A Cluster Computer and its Architecture

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

A computer node can be a single or multiprocessor system (PCs, workstations,

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

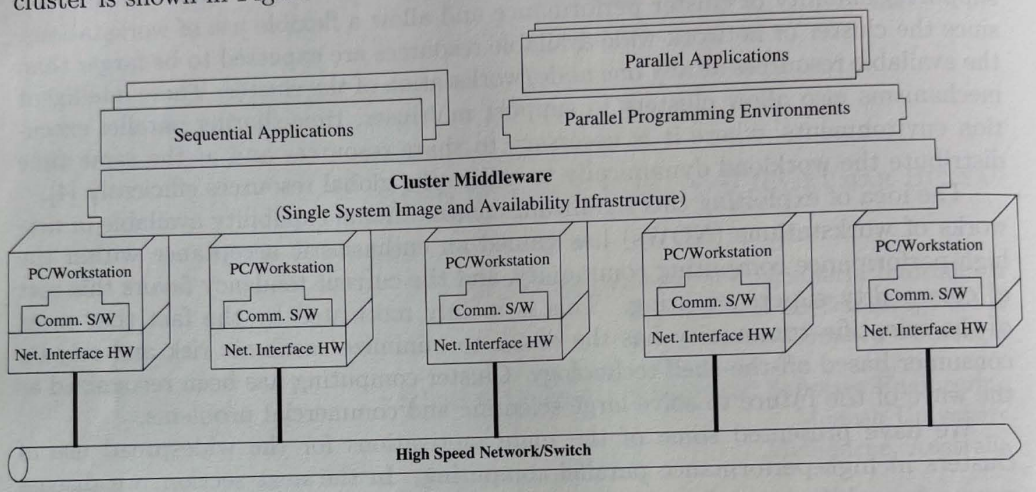

Figure 1.1 Cluster computer architecture.

The following are some prominent components of cluster computers:

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

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

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

- Applications
  - Sequential
  - Parallel or Distributed

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

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

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

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

1.3 Parallel Applications and Their Development

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

A typical example of a grand challenge problem is the simulation of some phenomena that cannot be measured through experiments. GCAs include massive

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

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

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

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

1. synchronous problems, which correspond to regular computations on regular data domains;

2. loosely synchronous problems, that are typified by iterative calculations on geometrically irregular data domains;

3. asynchronous problems, which are characterized by functional parallelism that is irregular in space and time; and

4. embarrassingly parallel applications, which correspond to the independent execution of disconnected components of the same program.

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

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

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

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

- Data Pipelining
- Divide and Conquer
- Speculative Parallelism

1.7.2 Task-Farming (or Master/Slave)

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

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

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

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

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

1.7.3 Single-Program Multiple-Data (SPMD)

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

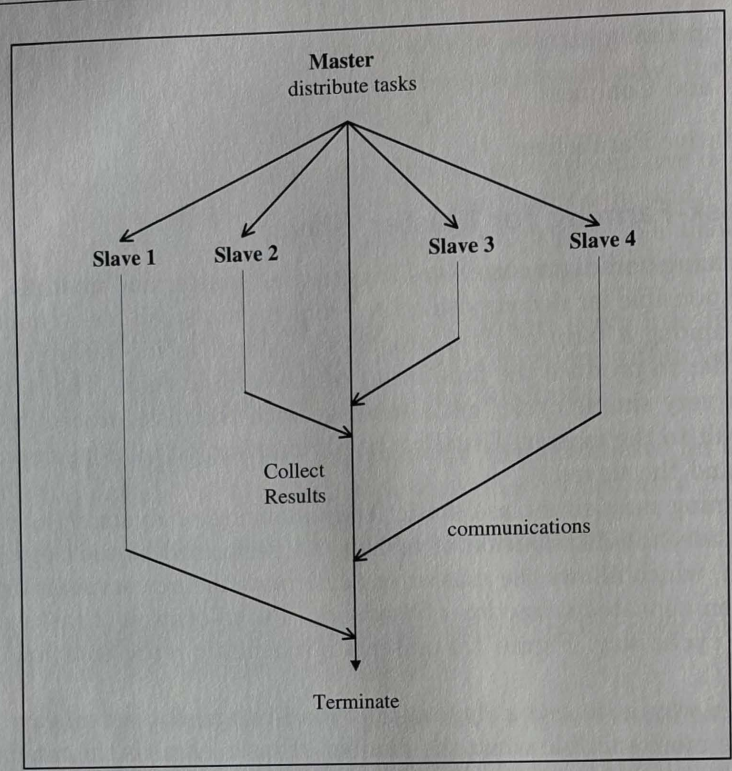

Figure 1.4 A static master/slave structure.

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

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

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

This paradigm is highly sensitive to the loss of some process. Usually, the loss

of a single process is enough to cause a deadlock in the calculation in which none of the processes can advance beyond a global synchronization point.

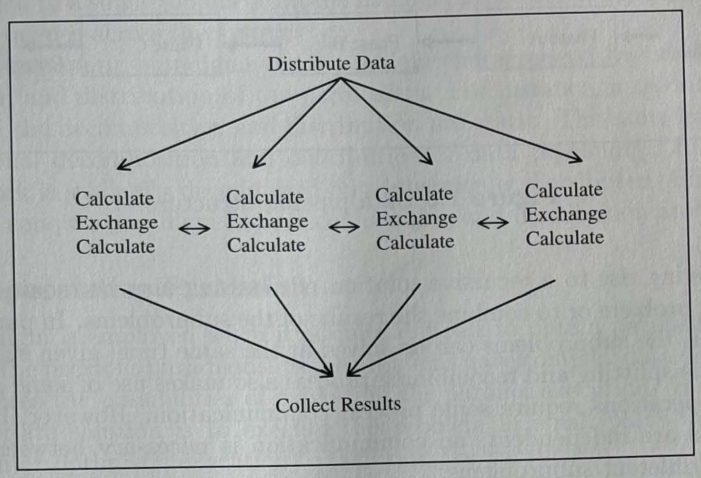

Figure 1.5 Basic structure of a SPMD program.

1.7.4 Data Pipelining

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

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

1.7.5 Divide and Conquer

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

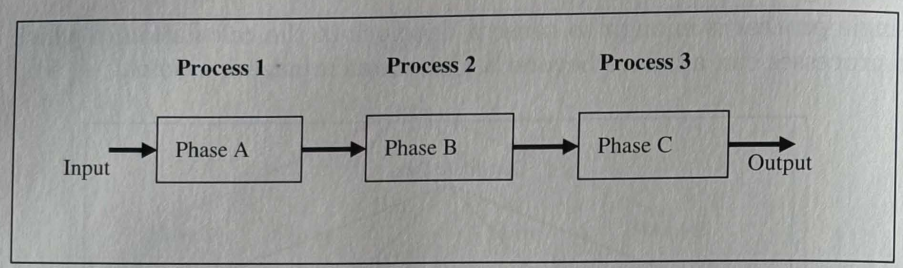

Figure 1.6 Data pipeline structure.

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

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

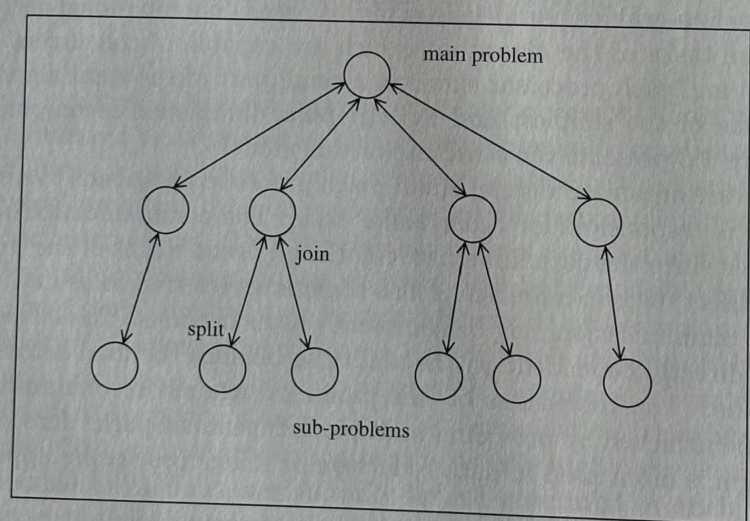

Figure 1.7 Divide and conquer as a virtual tree.

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

all the other processes are only responsible for the computation.

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

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

1.7.6 Speculative Parallelism

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

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

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

1.7.7 Hybrid Models

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

1.8 Programming Skeletons and Templates

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

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

Hiding the underlying structure from the user by presenting a simple interface results in programs that are easier to understand and maintain, as well as less prone

1. A = matrix representing the distances between pairs of vertices, initialized to the weight matrix W.
2. For $k = 0$ *to* $(n-1)$, perform steps 3 through 5.
3. For $i = 0$ *to* $(n-1)$, perform steps 4 through 5.
4. For $j = 0$ *to* $(n-1)$, perform step 5.
5. If $A_{ij} > (A_{ik} + A_{kj})$, then $A_{ij} = A_{ik} + A_{kj}$.

Algorithm 3.2 Sequential Floyd's.

Hence, in terms of the result matrix A, the entire matrix is updated throughout the algorithm application.

3.3.4 Parallel Algorithms

The parallel versions of the above two algorithms for the all pairs shortest path problem represent two distinct approaches to parallel processing, in terms of their communication and computation patterns. In this section, we will also see that their parallel implementations use two distinct parallel processing paradigms, the master-slave paradigm and the all-peers paradigm, for mapping the algorithm onto parallel tasks.

In the master-slave paradigm, one of the tasks is designated the master task; it controls the operations of the remaining tasks which are designated the slave tasks. In general, the master assigns each slave some work to do, and then collects the results from all the slaves and combines these into a solution. The slaves typically participate in the computation and the master oversees the entire procedure. Depending on the algorithm being implemented, the slaves may or may not communicate among themselves while they are computing their results. The slave tasks are also often referred to as worker tasks. This paradigm is illustrated in Figure 3.2.

In the all-peers paradigm, all tasks participate in the computation and are referred to as worker tasks in this scenario. Each of these tasks decides what portion of the problem it will solve, based on some common algorithm. In general, all tasks control the computation as a whole, communicating with each other as necessary both for computation and control purposes. Only one of the peer tasks may still read in the input data and distribute it to the rest of its peers, and then collect the results in the end. This paradigm is illustrated, in its most general form, in Figure 3.3, where it is assumed that a task communicates with all other tasks.

Since PVM has a dynamic task spawning capability, where a task spawns other tasks, most PVM implementations tend to fall naturally in the master-slave paradigm. Here, the first task that is started by hand takes the role of the master task, spawning several slave tasks according to the requirements of the application.

In the two versions of the all pairs shortest path problem solution that are discussed in this chapter, for the parallel Dijkstra's solution, the master-slave paradigm is employed for both the MPI and the PVM implementation. On the other hand,

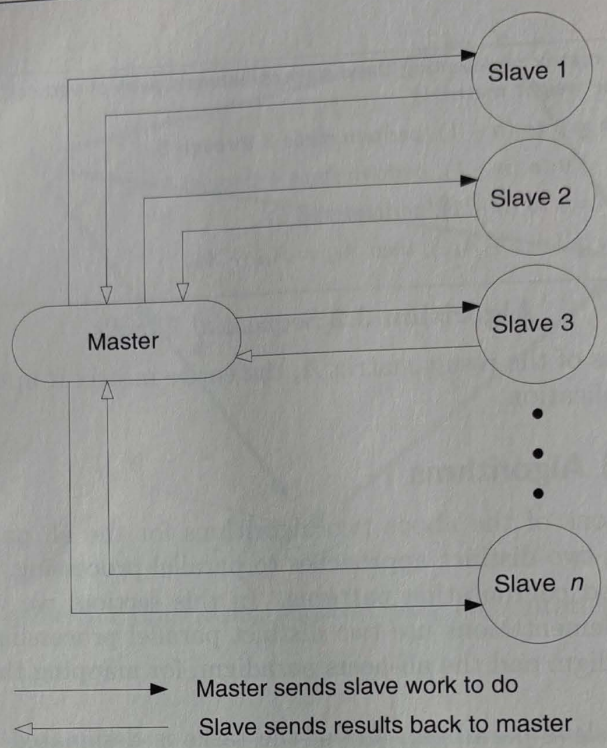

Figure 3.2 The master-slave paradigm.

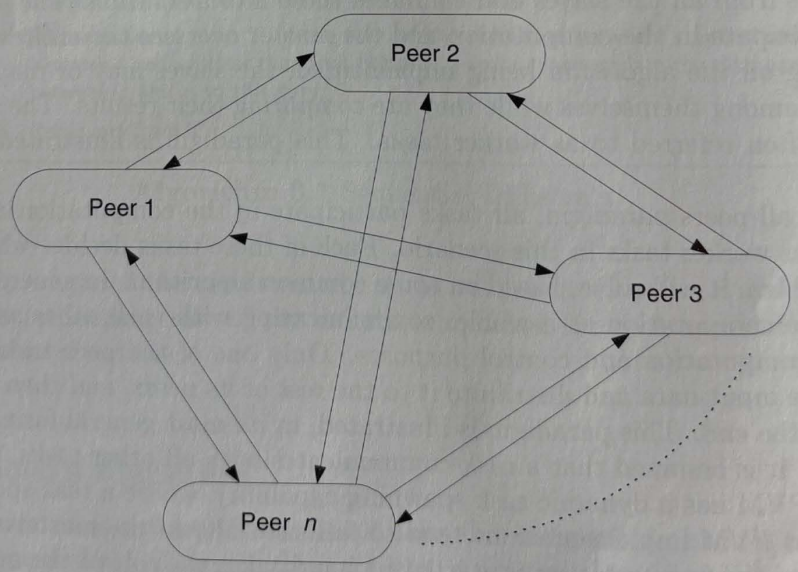

Figure 3.3 The all-peers paradigm.

for the parallel Floyd's solution, the all-peers paradigm is used for the MPI implementation and the master-slave paradigm is used for the PVM implementation. The PVM parallel implementations have been reported in [7] and the corresponding MPI implementations have been coded thereafter.

Parallel Dijkstra's Algorithm

Parallel Dijkstra's solution uses concurrent applications of the sequential algorithm for distinct sets of vertices, with each set assigned to different processors in the cluster. The collection of results from these tasks constitutes the all pairs solution. Each task needs the entire weight matrix for its calculation. This solution fits naturally into a master-slave paradigm. The master task has the input graph W, and it broadcasts this graph to all the slave tasks and waits for results back from the slaves. A slave, upon receiving the input matrix W, computes the shortest paths for its subset of vertices and sends its partial result back to the master. The master collects all the partial results and forms the complete result. The slave or worker tasks each execute Algorithm 3.3.

1. Wait for matrix W from master.
2. $p =$ number of slaves.
3. $m =$ my instance id.
4. For $i = m$ *to* $(n-1)$, in increments of p, perform Dijkstra's algorithm with source i.
5. Send results to master.

Algorithm 3.3 Worker task for parallel Dijkstra's.

We will refer to the MPI implementation of the parallel Dijkstra's algorithm as PDIJK_MPI and to its PVM implementation as PDIJK_PVM.

Parallel Floyd's Algorithm

In the parallel Floyd's solution [6] used in this chapter, the matrix A is partitioned into distinct stripes assigned to distinct tasks. A stripe consists of a set of consecutive columns of the matrix. A task is responsible for updating its stripe according to Floyd's algorithm (discussed in Section 3.3.3). Each stripe consists of contiguous columns of A. This solution lends itself well to the master-slave paradigm for PVM, and to the all-peers paradigm for MPI. In the master-slave approach used for PVM, the master assigns an instance ID to each of the slaves, and sends each slave its stripe of W. In the all-peers approach used for MPI, each task is a worker, with the *task identifier* (*taskid*) assigned by MPI serving as its instance identifier. One of the tasks is designated as the one responsible for reading in the input matrix and distributing the relevant portions to its peers. This task is the one with the taskid of 0 in our implementation. At the end, it collects the results from its peers,

which together yield the solution to the all pairs problem. The workers each execute Algorithm 3.4.

1. p = number of slaves.
2. Determine my stripe by dividing n by p, the first instance taking any remainder columns.
3. For $k = 0$ *to* $(n-1)$ do steps 4 and 5.
4. If column k belongs to my stripe, then broadcast that column to all other peers. Else, wait to receive the kth column from the owner peer of that column.
5. Update the elements of my stripe, according to Floyd's basic sequential algorithm.

Algorithm 3.4 Worker task for parallel Floyd's.

We will refer to the MPI implementation of the parallel Floyd's algorithm as PFLOYD_MPI and to its PVM implementation as PFLOYD_PVM.

3.4 The MPI Programming Environment

This section describes the MPI programming environment by first giving a brief overview, then discussing the most commonly used library functions in a typical program, and finally illustrating the above points via example programs that implement the two parallel algorithms described in the previous section. In the discussion that follows, the places where MPI calls are listed, only the C function from the MPI library is listed, for the sake of brevity. The corresponding FORTRAN call can be easily derived by converting all the letters in the C function to upper case. For example, the C call `MPI_Recv()` corresponds to the FORTRAN call `MPI_RECV()`. Additionally, the reader is referred to [1] for a complete listing of all library functions, both in C and FORTRAN, available in MPI-1. The example code excerpts given in this section have been stripped of variable declaration sections, error checking in most parts, and diagnostic information print statements, for the sake of brevity. Also, only those statements that pertain to MPI calls and associated comments are included in the excerpts as C program statements and comments; the other steps having been replaced with corresponding comments describing their general functionality.

As mentioned in Section 3.2, MPI consists of a specification of a set of library functions that tasks of a parallel program can use to communicate among themselves. By itself, the standard does not specify the exact method of program invocation and the run-time environment, although some of the conventions adopted by the public domain implementations of MPI have also been adopted by several vendor implementations. For example, there is a command called `mpirun`, which is almost the universal way to launch an MPI program. Additionally, each implementation of the MPI standard may come with different sets of environment variable

that need to be set to achieve different options for the run-time environment. A simple example of this is MPI_DIR, which denotes the working directory for an MPI program on a remote host for some implementations.

An MPI parallel program consists of MPI tasks, all of which may be the same or different executables. Each MPI task needs to register with the MPI environment before it can start using the MPI library.

Once an MPI task has registered with the MPI environment, it can start communicating with other MPI tasks, either directly between itself and a destination task using point-to-point communication primitives from the MPI library, or between all members of a group to which it belongs using collective communication functions. The group can be the default group that it belongs to (corresponding to when it registers), or a new group to which it has explicitly become a member.

Once the MPI task has completed its job, which will typically consist of some computation interspersed with communication with other tasks, it can unregister from the MPI environment.

3.4.1 Communication Models in MPI

Intertask communication in MPI is based on the concept of communicators. A communicator is used to specify a communication context for a communication operation and the set or group of tasks that share this context. That is, communication within a communicator is limited to the group or groups of tasks that belong to that communicator. When a task registers/initializes itself with the MPI environment, the task is automatically placed in a base, predefined communicator called MPI_COMM_WORLD. An MPI program may define other communicators to which subsets of its component tasks may belong, depending on the communication requirements among them. There are two types of communicators in MPI: intra-communicators, which are used to provide safe communication, both point-to-point and collective, within a group of tasks; and inter-communicators, which are used to provide safe communication between groups of tasks.

The process group within an intra-communicator such as MPI_COMM_WORLD is ordered and processes are identified by their rank within this group. This rank lies in the range $(0 \,..\, n-1)$ where n is the number of processes in the group. Considering the communicator MPI_COMM_WORLD, to which all MPI tasks belong, the rank of a task in the corresponding group is unique across all MPI tasks within an MPI program, and hence can be used as its task identifier, commonly referred to as its taskid. This is done in most MPI applications that need a task identifier. Similarly, the size of the group corresponding to MPI_COMM_WORLD denotes the number of parallel tasks in the program. In MPI-2, which will support dynamic spawning of tasks, this size may change depending on whether new tasks are being spawned or not.

Point-to-Point Communication

For direct communication between a source and a destination task, MPI provides various point-to-point send and receive functions. For each flavor, the sender task

puts data in a send buffer and sends it off to a specific destination which receives this data from a corresponding receive buffer. The two main types of point-to-point communication functions are blocking and non-blocking communication, depending on whether the return from the send/receive function signals the availability of the send/receive buffers for reuse by the application. This is illustrated in Figure 3.4. Additionally, a blocking send can be matched with a non-blocking receive and vice versa. Within each of these categories, there are several flavors depending on when the send or receive of a message can start and/or complete. The reader is referred to [1] for a detailed explanation of each of these flavors.

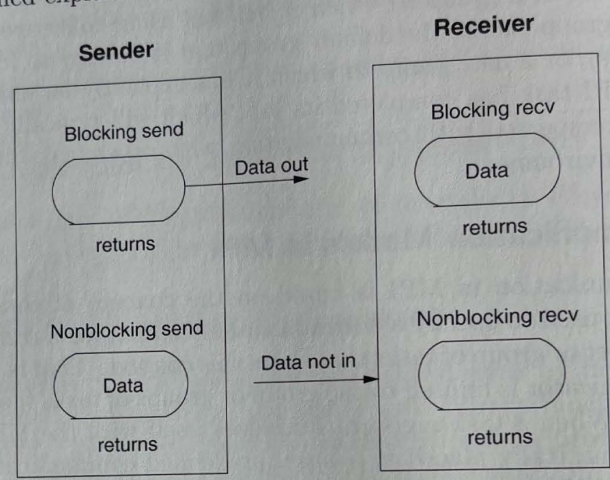

Figure 3.4 Point-to-point communication in MPI.

Collective Communication

For communication between a group of MPI tasks, where all tasks belonging to that group participate in the communication, MPI provides a rich set of collective communication primitives. Each such communication is characterized for a communicator, which defines the group of participating tasks. In collective communication, more than a single task may be the source of a message and more than a single task may be its destination. Each task participating in a collective communication operation calls the corresponding function with outgoing data in one or more send buffers and one or more receive buffers to receive incoming data. This is shown in Figure 3.5. For some types of such operations, there is a single sender or a single receiver in the group, and such a task is referred to as the root.

3.4.2 Creating an MPI Program

The MPI environment supports both the all-peers and the master-slave programming models very well. The only restriction as far as task control goes is that in

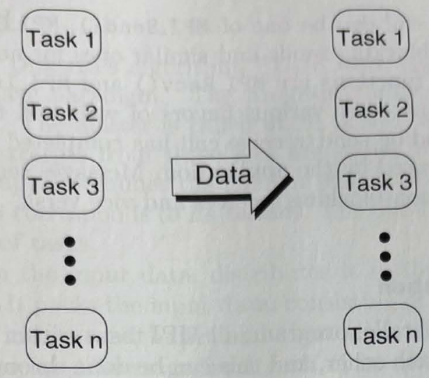

Figure 3.5 Collective communication in MPI.

MPI-1, the number of tasks in an MPI program has to be fixed prior to running the program, and no tasks can be spawned dynamically during runtime. MPI-2 will remove this restriction; however, implementations of MPI-2 will not become common in the immediate future. Although the MPI library is rich with several functions, all MPI programs will typically use the following minimum set of functions, arranged according to their type, and some additional functions depending on their requirements. The examples given in this section will illustrate the use of these and some additional MPI functions.

Environment Management

The first MPI function to be called in an MPI program must be `MPI_Init()`, and this must be called exactly once. As the name suggests, this function initializes the MPI environment for the program. The last MPI function to be called in an MPI program must be `MPI_Finalize()`. This function cleans up all MPI state in the program.

After `MPI_Init()` has been called in a program, the base communicator `MPI_COMM_WORLD` is established which includes all the tasks in the program and can be used to determine a task's rank and the number of tasks. To determine the number of tasks in the program, an MPI task needs to call the function `MPI_Comm_size()` with `MPI_COMM_WORLD` as the communicator. Similarly, the rank or taskid of an MPI task can be determined via the function `MPI_Comm_rank()`.

Point-to-Point Communication

MPI provides various flavors of both blocking and non-blocking point-to-point communication for each, including standard, buffered, synchronous and ready modes. Each flavor differs from the others in the mode of buffering used for the message in question, and the dependence of a matching receive for the send call to complete or

even start. The send call can be one of `MPI_Send()`, `MPI_Bsend()`, `MPI_Ssend()`, and `MPI_Rsend()` for blocking sends and similar ones for non-blocking sends. The corresponding receive functions are `MPI_Recv()` and `MPI_Irecv()`. For the asynchronous primitives, there are various flavors of wait and test functions to check whether the corresponding send/receive call has completed and whether the associated buffer can be reused by the application. Messages sent using blocking sends can be received using non-blocking receives and vice versa.

Collective Communication

As happens often in a parallel program, all MPI tasks within an MPI group want to exchange data among each other, and this can be done through one of the collective communication calls. A collective communication routine has to be called by all tasks in a group with matching arguments. A simple barrier synchronization is achieved via the `MPI_Barrier()` function. The remaining collective communication tasks take a root argument, which defines a single originator/receiver task. Then, depending on how the data is distributed from/to/among the root and the remaining tasks, there is `MPI_Bcast()` and various flavors of scatter/gather functions. Finally, there are `MPI_Reduce()` and `MPI_Allreduce()` which are used to perform reduction operations across the group.

3.4.3 Running an MPI Program

Once an MPI program is completely written and compiled using the platform and language specific compilers, it can be invoked by the environment-specific MPI program launcher. The most popular of these is `mpirun`, which typically takes, for each host in the cluster, at least the following arguments: number of tasks, arguments to each task, working directory. Depending on the platform in question and the MPI implementation being used, several environment variables need to be defined to be able to run the application, and some environment variables need to be defined to fully exploit the platform specific features of the underlying architecture. For example, on a Silicon Graphics DSM hardware, the environment variable `MPI_DSM_MUSTRUN` is used to pin MPI tasks to CPUs on the DSM, which can be useful for MPI jobs running on dedicated systems. An example of a typical command line using `mpirun` to start a program consisting of one instance of a master task, `master_mpi` and four instances of slave tasks, `slave_mpi`, with `master args` and `slave args` as their arguments, respectively, is:

```
mpirun -d <working directory> <hostname> \
   1 master_mpi <master args> : <hostname> 4 slave_mpi <slave args>
```

The `-d` option of `mpirun` is used to specify the location of the MPI task executables and other related files.

```
    exit(0);
} /* end of "main" */
```

Listing 3.2 MPI steps for pdijk_mpi.

3.4.5 PFLOYD_MPI

The parallel Floyd solution described in Section 3.3.4 has been implemented in MPI using the all-peers SPMD model. The peer tasks are called `pfloyd_mpi`. excerpts from which are given in listing 3.3. Each peer task determines the number of tasks in the program and its own taskid by querying the size and its rank in `MPI_COMM_WORLD`. One of the tasks, the one with taskid of `MASTERTID`, acts as a pseudo-master in that it reads in the input data from a file, distributes the data to its peers, collects results from them, and then writes the output data back to a file. It also participates in the computation of the output together with all its peers. So, from the standpoint of receiving input data and sending results back, the pseudo-master does not send itself a message, but just copies over the data and results into appropriate data structures.

Each peer, after registering with MPI, waits for its input stripe matrix. After receiving and unpacking the input data, each peer synchronizes with all its peers by calling a barrier function. After this synchronization takes place, the main computation (interspersed with communication) phase starts. During this phase, the peers work on their respective stripes of the input data matrix, and they need to communicate among themselves in every iteration of updates to the matrix to get the most up-to-date value of the next column in question. In other words, there is a fairly tight synchronization among them, and this is achieved here via the `MPI_Bcast()` function. Hence, for each update communication, depending on which column needs to be broadcast to all the peers, the corresponding task is the originator of data (i.e., the root in the collective communication) for the `MPI_Bcast()` call.

At the end of this main phase of the algorithm, the peers send their results to the pseudo-master. Since all the tasks are peers, and the pseudo-master is arbitrarily chosen to be the task with rank `MASTERTID`, all the peers know a priori who to send the results back to.

This example illustrates the use of both point-to-point and collective communication in MPI. As in the previous example of parallel Dijkstra's MPI solution, packing and unpacking of data is needed here for the same reasons: the receiver of a message needs the first piece of data before it can unpack the next piece.

Listing 3.3 is a code excerpt from the Parallel Floyd's MPI program, with the MPI calls again being **highlighted**.

```
main(int argc, char *argv[])
{
    /* Initialize MPI. */
    MPI_Init(&argc, &argv);
```

```
    /* Get our task ID (our rank in the basic group). */
    MPI_Comm_rank(MPI_COMM_WORLD, &myTid);

    /* Get the number of tasks. */
    MPI_Comm_size(MPI_COMM_WORLD, &nTasks);

    /* The master reads the input, and distributes it to the workers. */
    if (myTid == MASTERTID) {
        /*
         * Read in the number of vertices and the input matrix itself from a
         * specified file.
         */
        :
        /* Assign stripes to everyone, including ourselves. */
        sendWork();
    }
    else {
        collectWork();
    }

    /* Calculate shortest paths for the stripe assigned to us. */
    calcShort();

    if (myTid == MASTERTID) {
        /* Receive results from others and consolidate with our own. */
        collectResults();

        /* Write results in the specified output file. */
        :
    }
    else {
        sendResults();
    }

    /* Say goodbye to MPI. */
    MPI_Finalize();
    exit(0);
} /* end of "main" */


void sendWork()
{
    for (i = 0; i < nTasks; i++) {
        if (i == MASTERTID) {
            /* Read in my stripeMatrix entries directly from inMatrix. */
            :
        }
        else {
            /* Calculate the stripe for each peer task and send it to them. */
            bufPos = 0;
            MPI_Pack(&nVert, 1, MPI_INT, buf, BUFSIZE, &bufPos,
                     MPI_COMM_WORLD);
```

```
            for (j = 0; j < nVert; j++) {
                MPI_Pack(&(inMatrix[j][begCol]), width, MPI_INT,
                         buf, BUFSIZE, &bufPos, MPI_COMM_WORLD);
            }
            MPI_Send(buf, bufPos, MPI_PACKED, i, WORK_MSG,
                     MPI_COMM_WORLD);
        }
    }
} /* end of "sendWork" */


void collectWork()
{
    bufPos = 0;

    /* Get stripe matrix from master. */
    MPI_Recv(buf, BUFSIZE, MPI_PACKED, MPI_ANY_SOURCE, WORK_MSG,
             MPI_COMM_WORLD, &status);

    /* Unpack data. */
    MPI_Unpack(buf, BUFSIZE, &bufPos, &nVert, 1, MPI_INT,
               MPI_COMM_WORLD);

    /* Calculate my stripe width and allocate stripeMatrix. */
    :
    for (j = 0; j < nVert; j++) {
        MPI_Unpack(buf, BUFSIZE, &bufPos, stripeMatrix[j],
                   stripeWidth, MPI_INT, MPI_COMM_WORLD);
    }
} /* end of "collectWork" */


void collectResults()
{
    /*
     * Collect results and put it together in the output matrix.
     * First, myself directly.
     */
    :
    /* Then, others. */
    for (i = 0; i < nTasks-1; i++) {
        MPI_Recv(buf, BUFSIZE, MPI_PACKED, MPI_ANY_SOURCE, RESULT_MSG,
                 MPI_COMM_WORLD, &status);
        peerTid = status.MPI_SOURCE;

        /* Calculate starting column and stripe width for the results sent. */
        :
        for (j = 0; j < nVert; j++) {
            MPI_Unpack(buf, BUFSIZE, &bufPos, &(outMatrix[j][begCol]),
                       width, MPI_INT, MPI_COMM_WORLD);
        }
    }
```

```
} /* end of "collectResults" */

void
sendResults()
{
    /* Send stripeMatrix results back. */
    bufPos = 0;
    for (j = 0; j < nVert; j++) {
        MPI_Pack(&(stripeMatrix[j][0]), stripeWidth, MPI_INT, buf, BUFSIZE,
                 &bufPos, MPI_COMM_WORLD);
    }
    MPI_Send(buf, bufPos, MPI_PACKED, MASTERTID, RESULT_MSG,
             MPI_COMM_WORLD);
} /* end of "sendResults" */

void calcShort()
{
    /* All peers need to synchronize here. */
    MPI_Barrier(MPI_COMM_WORLD);

    /* Determine amount of data exchanged at each broadcast. */
    bcastSize = nVert * sizeof(int);

    /* Begin processing the kth (outermost) loop. */
    for (k = 0; k < nVert; k++) {
        /* Determine the broadcasting task. */
        rootTid = k/(nVert/nTasks);
        if (rootTid >= nTasks)
            rootTid = nTasks-1;

        if ((k >= begCol) && (k < begCol+stripeWidth)) {
            /* Broadcast column to all other "pfloyd_mpi" processes. */
            bufPos = 0;
            for (i = 0; i < nVert; i++) {
                MPI_Pack(&(stripeMatrix[i][k-begCol]), 1, MPI_INT, buf,
                         BUFSIZE, &bufPos, MPI_COMM_WORLD);
            }
            MPI_Bcast(buf, bcastSize, MPI_PACKED, rootTid,
                      MPI_COMM_WORLD);

            /* Update my stripe. */
            :
        }
        else {
            /* Wait for the kth column from another process */
            bufPos = 0;
            MPI_Bcast(buf, bcastSize, MPI_PACKED, rootTid,
                      MPI_COMM_WORLD);
            MPI_Unpack(buf, BUFSIZE, &bufPos, colRecd, nVert, MPI_INT,
                       MPI_COMM_WORLD);
            /* Update my stripe. */
            :
        }
```

```
    }
} /* end of "calcShort" */
```

Listing 3.3 MPI steps for pfloyd_mpi.

3.4.6 Tools for MPI Programs

The MPI standard includes support for tools for MPI programs by supporting communicators, contexts, and topologies, details of which can be found in [1]. Among visualization tools, XMPI [8], a public domain tool which has been also adopted by some vendors, is the most widely used. Among parallel debuggers, TOTALVIEW [9], and P2D2 [10] are the notable ones with support for MPI program debugging. In addition, there are several resource management tools available for clusters that have MPI support, including general purpose load balancers such as Platform Computing's LSF, vendors' proprietary load balancers for their parallel environment such as Silicon Graphics' NQE and IBM's LOADLEVELER, in addition to several public domain ones.

3.5 The PVM Programming Environment

This section describes the PVM programming environment by first giving a brief overview of this environment, followed by a discussion of the most commonly used library functions in a typical program. It illustrates the use of PVM via example programs that implement the two parallel algorithms described in Section 3.3. In the discussion that follows, at the places where PVM calls are listed, only the C functions from the PVM library are listed for the sake of brevity. The corresponding FORTRAN call is easy to extrapolate by replacing the "_" with the letter "f" in the pvm_ part of the C function name. For example, the FORTRAN counterpart for the C function pvm_recv() is pvmfrecv(). Also, the entire set of library routines (both C and FORTRAN) available in the latest PVM version, PVM 3.4, is listed in [2]. The example code excerpts given in this section have been stripped of variable declaration sections, error checking in most parts, and diagnostic print statements, for the sake of brevity. Also, only those statements that pertain to PVM calls and associated comments are included in the excerpts as C program statements and comments, the other steps having been replaced with corresponding comments describing their general functionality.

The PVM environment consists of a set of library routines to perform parallel/cluster communication between tasks, a defined start method for the parallel environment, and a console that allows a user to manipulate the parallel virtual machine by adding to and deleting hosts from it. The console also allows a user to start, monitor, and stop PVM programs. Finally, it has a set of other useful functions for debugging both the PVM environment and a PVM program. An alternative to setting up the parallel virtual machine via the console is to use a hostfile, and directly invoke the PVM daemon on the local host, which in turn invokes PVM daemons on remote hosts listed in the hostfile.

Appendix A

Glossary[1]

adaptive Taking available information into account. For example, an adaptive mesh-generating algorithm generates a finer resolution mesh near to discontinuities such as boundaries and corners. An adaptive routing algorithm may send identical messages in different directions at different times, depending on the local density information it has on message traffic.

address mask In Internet communications, a 32-bit long mask used to select an IP address for subnet addressing. The mask selects the network portion of the IP address and one or more bits of the local LAN address.

address space A region of a computer's total memory within which addresses are contiguous and may refer to one another directly. A shared memory computer has only one user-visible address space; a disjoint memory computer can have several.

address translation In networking, the process of converting external addresses into standardized network addresses and vice versa. This facilitates the interconnection of multiple networks each of which has its own address plan.

Amdahl's Law A rule first formalized by Gene Amdahl in 1967, which states that if F is the fraction of a calculation that is serial and 1-F the fraction that can be parallelized, then the speed-up that can be achieved using P processors is: $\frac{1}{F+\frac{1-F}{P}}$ which has a limiting value of $\frac{1}{F}$ for an infinite number of processors. It does not matter how many processors are employed, if a calculation has a 10% serial component, the maximum speedup obtainable is 10.

ANSI American National Standards Institute, a United States-based organization which develops standards and defines interfaces for telecommunications, among other things.

AppleTalk Apple Computer's proprietary network suite of protocols.

[1] A modified version by Mark Baker and Rajkumar Buyya. The original version (Courtesy of Geoffrey Fox) can be found at "The National HPCC Software Exchange (NHSE) HPCC Roadmap" Website: `http://nhse.npac.syr.edu/roadmap/`

Ex.1011 / Page 28 of 34
Intel Corporation

block To suspend one's own operation, or the operation of another process. A process may block itself, or be blocked by the system, until some event occurs. If all processes are simultaneously blocked, and no external event can cause any of them to become unblocked, then deadlock has occurred. The term, "block," is also often used to refer to a chunk of data or program.

blocking An operation that causes the process executing it to block. Usually applied to communications operations, where it implies that the communicating process cannot perform any other operations until the communication has completed.

BPS Bytes Per Second is a unit of memory access speed or communications transfer speed.

bridge A LAN internetworking device that filters and passes data between LANs based on Layer 2 (MAC layer) information. Bridges do not use any routing algorithms.

broadcast To send a message to all possible recipients. Broadcast can be implemented as a repeated send, but is more efficiently implemented by using spanning trees and having each node in the tree propagate the message to its descendants.

buffer A temporary storage area in memory. Many methods for routing messages between processors use buffers at the source and destination, or at intermediate processors. See also packet switching, virtual cut-through and wormhole routing.

bus A single physical communications medium shared by two or more devices. The network shared by processors in many distributed computers is a bus, as is the shared data path in many multiprocessors.

busy-waiting A situation whereby processor cycles are used to test a variable until it assumes a desired value.

cache A high-speed memory, local to a single processor, whose data transfers are carried out automatically in hardware. Items are brought into a cache when they are referenced, while any changes to values in a cache are automatically written when they are no longer needed, when the cache becomes full, or when some other process attempts to access them.

cache consistency The problem of ensuring that the values associated with a particular variable in the caches of several processors are never visibly different.

cache hit A cache access that successfully finds the requested data.

cache line The unit in which data is fetched from memory to cache.

clock time Physical or elapsed time, as seen by an external observer. Nonrelativistic time. In small computer systems where all components can be synchronized, clock time and logical time may be the same everywhere, but in large systems it may be difficult for a processor to correlate the events it sees with the clock time an external observer would see. The clock times of events define a complete order on those events.

CLTP Connectionless Transport Protocol, which provides end-to-end transport data addressing and is OSI's equivalent to UDP.

clustered computing A commonly found computing environment consists of many workstations connected by a local area network. The workstations, which have become increasingly powerful over the years, can together be viewed as a significant computing resource. This resource is commonly known as cluster of workstations, and can be generalized to a heterogeneous collection of machines with arbitrary architecture.

cluster middleware A layer that resides between the operating system and user-level environment and offers single system image and system availability services.

CMIS Is the service performed by the Common Management Internet Protocol.

co-processor An additional processor attached to a main processor, to accelerate arithmetic, I/O or graphics operations.

combining Joining messages together as they traverse a network. Combining may be done to reduce the total traffic in the network, to reduce the number of times the start-up penalty of messaging is incurred, or to reduce the number of messages reaching a particular destination.

combining switch An element of an interconnection network that can combine certain types of requests into one request and produce a response that mimics serial execution of the requests.

communication overhead A measure of the additional workload incurred in a parallel algorithm due to communication between the nodes of the parallel system.

compiler directives Special keywords often specified as comments in the source code, but recognized by the compiler as providing additional information from the user for use in optimization.

compiler optimization Rearranging or eliminating sections of a program during compilation to achieve higher performance. Compiler optimization is usually applied only within basic blocks and must account for the possible dependence of one section of a program on another.

IPX Integrated Packet Exchange, for example, Stratacom's Packet switch for public and private T1 and E1 networks.

ISO International Standards Organization, which, among other things, sets standards for programming languages.

Java A distributed computing language (Web Technology) developed by Sun, which is based on C++, but supports Applets.

join To wait for the termination of one or more descendent processes that were forked at some earlier time.

Kermit A public domain file transfer and terminal emulation program. Similar in functionality to uucp.

kernel A process providing basic services. A service kernel can run on every processor to support minimal operating system services, while a routing kernel can handle or forward incoming messages.

key Unique object of a search.

knowledge Information plus semantic meaning.

LAN Local Area Network, a network of multiple interconnected data terminals or devices within a local area to facilitate data transfer. Most notable of LAN topologies is ethernet, token ring, FDDI, etc.

LAP Link Access Procedure is a modified form of HDLC that CCITT specified for X.25 networks. LAP-B is link access procedures- balanced and is the X.25 implementation of SDLC and similarly, LAP-D is the ISDN and frame relay implementation of SDLC.

LAPACK A Linear Algebra Software Package, which has been mounted on a wide range of platforms. It evolved from the older LINPACK package from Netlib.

latency The time taken to service a request or deliver a message which is independent of the size or nature of the operation. The latency of a message passing system is the minimum time to deliver a message, even one of zero length that does not have to leave the source processor; the latency of a file system is the time required to decode and execute a null operation.

light-weight process A process which executes concurrently with other processes, in the same address space and in an unprotected fashion. Light-weight processes are used by systems such as MACH to reduce the overhead of process startup.

memory protection Any system that prevents one process from accessing a region of memory being used by another. Memory protection is supported both in hardware and by the operating system of most serial computers, and by the hardware kernel and service kernel of the processors in most parallel computers.

mesh A topology in which nodes form a regular acyclic d-dimensional grid, and each edge is parallel to a grid axis and joins two nodes that are adjacent along that axis. The architecture of many multicomputers is a two- or three-dimensional mesh; meshes are also the basis of many scientific calculations, in which each node represents a point in space, and the edges define the neighbors of a node.

message passing A style of interprocess communication in which processes send discrete messages to one another. Some computer architectures are called message passing architectures because they support this model in hardware, although message passing has often been used to construct operating systems and network software for uniprocessors and distributed computers.

message typing The association of information with a message that identifies the nature of its contents. Most message passing systems automatically transfer information about a message's sender to its receiver. Many also require the sender to specify a type for the message, and let the receiver select which types of messages it is willing to receive.

MPI The Message Passing Interface standardizes the communication subroutine libraries used for programming on massively parallel computers such as Intel's Paragon and Cray's T3D, as well as networks of workstations. MPI not only unifies within a common framework programs written in a variety of exiting (and currently incompatible) parallel languages, but allows for future portability of programs between machines.

metacomputer This term describes a collection of heterogeneous computers networked by a high-speed wide area network. Such an environment would recognize the strengths of each machine in the metacomputer and use it accordingly to efficiently solve so-called metaproblems. The World Wide Web has the potential to be a physical realization of a Metacomputer.

metaproblem This term describes a class of problem which is outside the scope of a single computer architectures, but is instead best run on a Metacomputer with many disparate designs. These problems consist of many constituent subproblems. An example is the design and manufacture of a modern aircraft, which presents problems in geometry grid generation, fluid flow, acoustics, structural analysis, operational research, visualization, and database management. The Metacomputer for such a Metaproblem would be networked workstations, array processors, vector supercomputers, massively parallel processors, and visualization engines.

OSF Open Software Foundation; an organization established by a number of the major computer manufacturers to set software standards.

packet switching A routing technique in which intermediate nodes wait until they have received the whole of a message before forwarding any of it. Packet switching often requires a large amount of buffer space, and contention for access to this space can lead to deadlock.

page The smallest unit of a virtual memory system. The system maintains separate virtual-to-physical translation information for each page.

parallel computer A computer system made up of many identifiable processing units working together in parallel. The term is often used synonymously with concurrent computer to include both multiprocessor and multicomputer.

parallel slackness Hiding the latency of communication by giving each processor many different tasks, and having them work on the tasks that are ready while other tasks are blocked (waiting on communication or other operations).

PVM Parallel Virtual Machine was developed at Emory and Tennessee Universities, and Oak Ridge National Laboratory. It supports the message passing programming model on a network of heterogeneous computers (`http://www.epm.ornl.gov/pvm/`).

parallelization Turning a serial computation into a parallel one. Also sometimes turning a vector computation into a parallel one. This may be done automatically by a parallelizing compiler or (more usually) by rewriting (parts of) the program so that it uses some parallel paradigm.

parsing Process whereby a compiler analyzes the syntax of a program to establish the relationship among operators, operands, and other tokens of a program. Parsing does not involve any semantic analysis.

partitioning Process of restructuring a program or algorithm into independent computational segments. The goal is to have multiple processors simultaneously work on these independent computational segments.

percentage parallelization The percentage of processor expenditure processed in parallel on a single job. It is not usually possible to achieve 100 percent of an application's processing time to be shared equally on all processors. See Amdahl's Law.

percentage vectorization The percentage of an application executing in vector mode. This percentage may be calculated as a percentage of CPU time or as the percentage of lines of code (usually Fortran) in vector instructions. The two approaches are not consistent and may give very different ratings. The first calculation method leads to performance improvement as measured by

Parallel Computing

High Performance Cluster Computing

Programming and Applications VOL. 2

A comprehensive guide to today's most advanced R&D in highly parallel programming and applications.

Volume I of this two-volume set collected today's best work on the systems aspects of high performance cluster computing. Now, in ***High Performance Cluster Computing: Programming and Applications, Volume 2,*** Rajkumar Buyya brings together the world's leading work on programming and applications for today's state-of-the-art "commodity supercomputers."

The book is organized into three areas: programming environments and development tools; Java™ as a language of choice for development in highly parallel systems; and state-of-the-art high performance algorithms and applications. All three areas have seen major advances in recent years—and in all three areas, this book offers unprecedented breadth and depth.

Coverage includes:

- New parallel programming techniques and tools, including MPI and PVM, active objects, scoped behavior, and LiPS
- State-of-the-art debugging techniques: Code liberation, global renaming, name reclamation, and debugging interfaces
- The WebOS: Designing operating system services for wide-area applications
- Leveraging Java to the fullest: Distributed objects, the HPspmd model, and more
- Object-oriented implementation of parallel genetic algorithms
- Real-time resource management, climate ocean modeling, parallel reflexive reasoning, content-based image retrieval, biomedical applications, and more.

Together with Volume I, ***High Performance Cluster Computing: Programming and Applications, Volume 2,*** captures the growing power of the cluster computing revolution. Whether you are a developer, researcher, administrator, manager, or user, these pages will show you the future of computing.

About the Editor

RAJKUMAR BUYYA is a researcher at the School of Computer Science and Software Engineering, Monash University, Melbourne, Australia. He was guest editor for the *Special Issue on High Performance Computing on Clusters, Parallel and Distributed Computing Practices Journal*, and co-author of *Mastering C++* and *Microprocessor x86 Programming*. He is a speaker in the IEEE Computer Society Chapter Tutorials Program and chairman of the IEEE Computer Society Task Force on Cluster Computing.

ISBN 0-13-013785-5

90000

9 780130 137852

Prentice Hall
Upper Saddle River, NJ 07458
http://www.phptr.com