COMMUNICATION OPTIMIZATION IN PARSEC:
A PARALLEL PROGRAMMING ENVIRONMENT FOR MULTICOMPUTERS

by

SAMEER S. MULYE
B.E. (Computer Engineering) University of Bombay, India, 1990

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN THE FACULTY OF GRADUATE STUDIES
DEPARTMENT OF COMPUTER SCIENCE

We accept this thesis as conforming to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA
November, 1994

© Sameer S. Mulye, 1994
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.
Abstract

Multicomputers (distributed memory MIMD machines) have emerged as inexpensive, yet powerful parallel computers that promise high computational performance. A parallel application running on these machines consists of a set of user processes running on different processors. The only way processes share data in this environment is by message passing which can become the bottleneck to obtaining high performance.

This thesis examines the issues influencing communication in multicomputers. Two static analysis tools are designed: a mapping tool and runtime system for communication and synchronization. An efficient mapping of a parallel application onto a parallel computer is necessary to minimize the total communication cost. Delegating this task to the user makes application development a complex process as the user must know the underlying hardware. An automated hardware-independent mapping tool is designed to map arbitrary process graphs on a transputer-based multicomputer.

A native runtime system on these multicomputers consists of message-passing primitives which are efficient but only support simple send and receive operations. A high-level runtime system provides functionalities such as flow control, buffering, and group communication using a wide range of primitives. However, these primitives are difficult to use and it is up to the user to select the optimal primitives for efficiency. Hence a message-passing library is designed for a transputer system. It is built on top of Trollius, an operating system for multicomputers, which offers a variety of functions, each with different combination of performance and functionality. The library includes a few communication primitives which provide a common interface for all the Trollius functions while maintaining the efficiency of the low-level functions. The mapping tool statically compiles a program structure into the configuration information and based on this information, the message-passing library selects the optimal Trollius functions.
Parsec (the Parallel System for Efficient Computation), a parallel programming environment for multicomputers, is being developed at UBC. Our tools are integrated with Parsec and become the backbone in achieving high performance. Heterogeneity is one of the important design issues in Parsec. Currently these tools are available on two different hardware platforms: a transputer-based multicomputer and a cluster of Sun Sparc workstations. Thus the user can take advantage of either environment to write portable and efficient parallel applications.
Table of Contents

Abstract

Table of Contents

List of Tables

List of Figures

Acknowledgment

1 Introduction
   1.1 Problems ........................................... 2
   1.2 Motivations ........................................ 4
   1.3 Approach .......................................... 10
   1.4 Thesis Context .................................... 12
   1.5 Thesis Organization ............................... 12

2 Related Work and Background 14
   2.1 Parallel Programming Environment .................. 15
   2.2 Parallel Languages and Compilers .................. 18
   2.3 Mapping tools .................................... 18
   2.4 Message-passing Libraries ........................ 21
   2.5 Background ....................................... 22

3 System Design ........................................ 26
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 System Overview</td>
<td>27</td>
</tr>
<tr>
<td>3.2 The Mapping Tool</td>
<td>29</td>
</tr>
<tr>
<td>3.3 Runtime System</td>
<td>32</td>
</tr>
<tr>
<td>3.3.1 Parsec Process Model</td>
<td>32</td>
</tr>
<tr>
<td>3.3.2 Message Passing Primitives</td>
<td>34</td>
</tr>
<tr>
<td>4 Mapping Strategy on Multicomputers</td>
<td>37</td>
</tr>
<tr>
<td>4.1 Hardware System</td>
<td>37</td>
</tr>
<tr>
<td>4.2 Structural Overview</td>
<td>39</td>
</tr>
<tr>
<td>4.3 Mapping Strategy</td>
<td>46</td>
</tr>
<tr>
<td>4.3.1 Mathematical Formulation</td>
<td>46</td>
</tr>
<tr>
<td>4.3.2 Mapping Constraints</td>
<td>47</td>
</tr>
<tr>
<td>4.3.3 Algorithm</td>
<td>53</td>
</tr>
<tr>
<td>4.3.4 Experimentation</td>
<td>58</td>
</tr>
<tr>
<td>5 Runtime System for Transputers</td>
<td>61</td>
</tr>
<tr>
<td>5.1 Transputer Programming Model</td>
<td>61</td>
</tr>
<tr>
<td>5.2 Low Level Software Tools</td>
<td>62</td>
</tr>
<tr>
<td>5.2.1 Logical C</td>
<td>62</td>
</tr>
<tr>
<td>5.2.2 Trollius</td>
<td>64</td>
</tr>
<tr>
<td>5.3 Runtime System</td>
<td>67</td>
</tr>
<tr>
<td>5.3.1 Implementation Details</td>
<td>70</td>
</tr>
<tr>
<td>5.4 Performance Analysis</td>
<td>75</td>
</tr>
<tr>
<td>6 Programming A Network of Workstations</td>
<td>77</td>
</tr>
<tr>
<td>6.1 Workstation Environment</td>
<td>77</td>
</tr>
<tr>
<td>6.2 Mapping Tool</td>
<td>79</td>
</tr>
</tbody>
</table>
List of Tables

1.1 Message Passing Timings ............................................. 8
4.1 Wire Structures ...................................................... 41
4.2 Mapping Statistics .................................................... 59
5.1 Trollius Function Invocation ....................................... 72
5.2 Parsec: Message Passing timings ................................. 76
6.1 Parsec: TCP communication ....................................... 87
6.2 PVM3: daemon mediated UDP communication ................. 87
6.3 PVM3: TCP communication ........................................ 88
6.4 Spring: Performance ................................................ 96
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Process Graph</td>
<td>7</td>
</tr>
<tr>
<td>1.2</td>
<td>Mapping</td>
<td>8</td>
</tr>
<tr>
<td>2.1</td>
<td>Parsec Objects</td>
<td>23</td>
</tr>
<tr>
<td>3.1</td>
<td>System Overview</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>Transputer Network (part of Component 2)</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>Structural Overview</td>
<td>39</td>
</tr>
<tr>
<td>4.3</td>
<td>Resources (part of Component 0)</td>
<td>40</td>
</tr>
<tr>
<td>4.4</td>
<td>System graph</td>
<td>42</td>
</tr>
<tr>
<td>4.5</td>
<td>Conflict-graph for Figure 4.3</td>
<td>43</td>
</tr>
<tr>
<td>4.6</td>
<td>User Process Graph</td>
<td>44</td>
</tr>
<tr>
<td>4.7</td>
<td>Guest Graph</td>
<td>45</td>
</tr>
<tr>
<td>4.8</td>
<td>(a) Guest graph, (b) Host graph for case 1, (c) Host graph for case 2</td>
<td>51</td>
</tr>
<tr>
<td>5.1</td>
<td>Decision Tree</td>
<td>69</td>
</tr>
<tr>
<td>6.1</td>
<td>Process Graph for <em>Spring</em> in Parsec</td>
<td>90</td>
</tr>
<tr>
<td>6.2</td>
<td>Graphical Interface for Mapping Tool in Parsec</td>
<td>92</td>
</tr>
<tr>
<td>6.3</td>
<td>Graphical Interface for Loading Tool in Parsec</td>
<td>93</td>
</tr>
<tr>
<td>A.1</td>
<td>Component 1 (Box 1)</td>
<td>105</td>
</tr>
<tr>
<td>A.2</td>
<td>Component 0 (Box 2)</td>
<td>106</td>
</tr>
<tr>
<td>A.3</td>
<td>Component 3 (Box 3)</td>
<td>107</td>
</tr>
</tbody>
</table>
A.4 Component 2 (Box 4) ................................. 108
A.5 Component 2 (Box 5) ................................. 109
A.6 Transputer System as a 8x9 mesh ................. 110
Acknowledgment

I would like to thank both my supervisors Dr. Samuel Chanson and Dr. Alan Wagner for their support and guidance. I thank David Feldcamp, Mandeep Dhami, and H. V. Sreekantaswamy for their valuable assistance during my research work. My thanks go to Bill Gates for proofreading this thesis, Vishwa Ranjan for his continuous encouragement, and Raza Khan and Chris Romanzin for being there on those countless nights. I am grateful to Dr. Chanson and to the Department of Computer Science at the University of British Columbia for providing the financial support.

Finally, I would like to express my acknowledgments to my parents, my constant source of inspiration.
Chapter 1

Introduction

Parallelism is one way to achieve computational speedup for a large class of applications. A wide range of parallel computer architectures [59] are available to exploit parallelism. The most basic distinction between these architectures is whether the processors synchronously execute the same instruction on multiple data (SIMD) or asynchronously execute multiple instructions on multiple data (MIMD).

MIMD machines are grouped into two categories according to their address-space organization [2]: message-passing architectures and shared-address-space architectures. In a message-passing architecture, each processing element (PE) consists of a processor and its local memory and the PEs are interconnected by a high-speed network. MIMD message-passing computers are commonly referred to as multicomputers. In a shared-address-space architecture, all processors have read and write access to a shared address space. Processors interact by modifying data objects stored in a shared space. These computers are referred to as multiprocessors.

There are several factors that motivate the use multicomputers for parallel computing. A multicomputer can be built out of inexpensive and powerful microprocessors and can scale easily to accommodate a large number of PEs. Performance of these machines has improved over the years with increasing CPU speed and decreasing message-passing time. For these reasons, multicomputers are viewed as low-cost, scalable, and high-performance parallel computers [35]. Examples of multicomputers include Intel iPSC, Transputers, CM-5, and Intel Paragon. In recent years, networked workstations (workstations in a local
area network) have emerged as an effective alternative for dedicated parallel machines [37].

1.1 Problems

Unfortunately, it is difficult for a naive user to program a multicomputer. Many multicomputers provide low-level and hardware-dependent programming environments (for example, NX/2 for the iPSC and Logical C for the transputer). Programmers in these environments, must know the details of a machine architecture such as the number of processors, the network connection, the memory organization, the I/O processing, the operating system, and the communication and synchronization facilities. These programs guarantee high performance but at the cost of a complex design and development process. The hardware dependencies inherent in these programs also make them difficult to port to a different architecture. On the other hand, high-level programming environments [40, 42, 41] support common interfaces across different hardware platforms. These environments provide high-level abstractions to simplify the program development, but their performance is poor. Thus there arises a trade-off between high-level and low-level programming environments. Developing high-performance high-level programming environment is a challenging task. The primary focus of this thesis is on two performance related issues associated with high-level programming environment for multicomputers: the mapping problem and a runtime system for communication and synchronization.

Mapping problem

The first step in executing a parallel application on a multicomputer is to assign each of its processes to an individual processor. The problem of assigning these processes, in order to minimize the communication cost is known as the mapping problem [17, 29]. The
process assignment can be static or dynamic. In static mapping, processes are assigned to processors prior to any program execution and in dynamic mapping, processes are dynamically reassigned to processors in response to changing load on processors and network. The way in which a mapping algorithm allocates the processes directly affects the program execution time. For example, if two communicating processes are assigned to two non-adjacent processors, then messages are delayed because of extra copying at the intermediate nodes. Similarly assigning the processes to the same processor serializes the communication and hence slows down the execution. The problem of finding an optimal task assignment is known to be NP-complete [3]. This problem is even more difficult for reconfigurable multicomputers. In these multicomputers, the interconnection network can be changed using control switches that are configured by software. A good mapping requires not only an efficient algorithm but an elegant way to formulate the problem. The difficulty lies in formulating the dynamic characteristics of the interconnection network.

Runtime system

Multicomputers support task-level parallelism based on a message-passing paradigm. A native runtime system provides message-passing primitives for communication and synchronization which ensure high performance. Usually these primitives support only point-to-point send and receive operations. Higher level message-passing systems, built on top of the native runtime system, provide flow-control, buffer management, routing, and group communication. However with these added functionalities, the high-level primitives are difficult to use. Given a wide range of primitives, it is up to the user to select the optimal message-passing primitives.

Ideally one would like to tailor the message-passing to the type of communication so that neighbour-to-neighbour, non-multiplexed processes can use the lowest-level faster
mechanism whereas higher-level multiplexing and routing is reserved only for the messages that need it. However, this information is not available until after mapping is done. Such an optimization cannot be done in the application because the user code needs to be changed if the mapping changes. A runtime system should provide this type of optimization.

Apart from these performance-related problems, we also address portability. The tools which are hardware-dependent should be designed for all the target architectures and should present a generic user interface for application development across multiple platforms. Once an application is developed for a particular MIMD architecture, it cannot be run on other MIMD architectures because the tools and techniques used to parallelize the code are specific to a particular hardware platform. As a result, considerable effort is necessary to port code to other platforms. This will certainly discourage the user from developing applications for different parallel architectures.

1.2 Motivations

We first identify the relationships between the aforementioned problems and the target machine architecture and its native runtime system.

Target Hardware and Software Systems

Both the mapping problem and the runtime system are discussed in the context of a reconfigurable transputer-based multicomputer. The multicomputer consists of 75 INMOS T800 transputers and 10 C004 programmable crossbar switches. Each crossbar switch has 32 data links and a control link. Any pair of data links can be connected by sending an appropriate software command to a control link. The system can be reconfigured into
Chapter 1. Introduction

a variety of interconnection networks. There are two software tools available on the transputer system at UBC: Trollius [9] and Logical C [7, 8]. Logical C provides hardware-level message-passing primitives. The Trollius message-passing primitives resemble the OSI model of protocol architecture. Different primitives are available at the physical level, data-link level, network level, and transport level in order of increasing functionality and decreasing efficiency.

The mapping problem is characterized by various factors such as parallel program structure, computational cost of a task, communication cost among the tasks, number of processors, processors' connectivity, the speed of an individual processor, and communication cost between the processors. It is independent of a machine architecture and the native runtime system. The objective of the message-passing library, built on top of the native runtime system, is to provide high-level easy-to-use primitives without compromising performance. The optimization techniques employed by such a system depends on the native runtime system as well as on the machine architecture.

There are several reasons for having a mapping tool to allocate tasks in order to minimize the communication overhead.

1. Assignment of the processes to the processors should be simplified. Many programming environments like PVM [42], TOPSYS [44], and Trollius and languages like Occam and Logical C leave this up to the programmer. Although this is not difficult when there are only a few processes, it becomes more tedious as the number of processors and processes grow.

2. Different programming paradigms require different process topologies. In order to map these different topologies by hand, on a particular parallel architecture, the user must know the underlying hardware interconnection. But such topology independence is easier when the mapping is performed automatically by the software
system.

3. A reconfigurable network makes the mapping problem more difficult. One can enhance performance by using a mapping tool that can do better than a naive user.

4. A variety of runtime systems are available to support different functionalities for different types of communication. We sought to take advantage of these functionalities while at the same time providing easy to use programming interface. Different communication primitives affect the performance differently. The mapping should be done in such a way that the total communication cost will be minimum. This will allow one to take advantage of the lowest-level (and most efficient) communication primitives. However, such optimization by hand is not desirable for the programmer because every time a program structure changes, the mapping information changes, and so does the optimization code. Hence both the mapping and the optimization should be done automatically.

Example

In order to motivate the problems associated with optimizing communication, we present an example which shows how mapping affects the optimization and what overheads are associated with the Trollius communication primitives and the Logical C channel primitives. A process graph is mapped onto the transputer system, and the message-passing time between every pair of processes is measured.

Figure 1.1 shows a five node complete graph. This graph is mapped on the component 0 of our transputer system (Figure A.2) which has 25 transputers where transputer T₀ is directly connected to the host machine (Sun Sparc machine). Component 0 has two rings of 8 transputers each (T₀–T₇ and T₁₆–T₂₃) and one chain of 9 transputers (T₈–T₁₅, T₀). The
transputers within a ring or a chain are connected on hard-links whereas two transputers
across a ring and a chain can be connected over the crossbars. The mapping of the process
graph onto component 0 is shown in Figure 1.2. Dilation of an edge in the process graph
is defined as the number of physical links the edge is mapped onto. Edges $e_{sp1}$, $e_{sp2}$,
$e_{sp3}$, $e_{sp4}$, and $e_{p0p4}$ have dilation greater than one. Logical routes include all the physical
links between every pair of communicating processes. Depending on the mapping done
(which determines the dilation of edges in the process graph and the physical links that
are multiplexed), a specific Trollius function can be used. For example, two processes $P_0$
and $P_1$ are mapped on the adjacent transputers $T_a$ and $T_{15}$ respectively. However, the
physical link between $T_a$ and $T_{15}$ is multiplexed over four different edges in the process
graph. Hence the physical level or data-link level Trollius primitives or Logical C channel
primitives cannot be used for communication between $P_0$ and $P_1$. One is then limited to
the network level or the transport level Trollius primitives.

Table 1.1 shows the message-passing time between every pair of processes for the given
mapping. In this experiment 1000 messages, each of 4k bytes, were sent from the source
node to the destination node. The round-trip time is measured for message-passing
between every process pair. Two communication levels are used: network level and
hardware level. At the hardware level, Logical C primitives are used instead of physical-
level Trollius primitives to minimize the delay. (To use Logical C primitives from within
Trollius, it is required to set up virtual circuit using data-link level or network level
Trollius primitives. This is described in detail in Chapter 5).

Process pairs $P_0$-$P_1$ and $P_1$-$P_2$ communicate over the hard-links (two transputers

<table>
<thead>
<tr>
<th>Source node</th>
<th>Dest node</th>
<th>Communication level</th>
<th>Time (in sec)</th>
<th>Transmission rate (in MBytes/Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>$P_1$</td>
<td>Network</td>
<td>6.060</td>
<td>0.675</td>
</tr>
<tr>
<td>$P_1$</td>
<td>$P_2$</td>
<td>Hardware</td>
<td>4.864</td>
<td>0.842</td>
</tr>
<tr>
<td>$P_2$</td>
<td>$P_4$</td>
<td>Hardware</td>
<td>7.025</td>
<td>0.582</td>
</tr>
<tr>
<td>$P_1$</td>
<td>$P_4$</td>
<td>Network</td>
<td>7.633</td>
<td>0.536</td>
</tr>
<tr>
<td>$P_0$</td>
<td>$P_4$</td>
<td>Network</td>
<td>13.318</td>
<td>0.307</td>
</tr>
</tbody>
</table>

Table 1.1: Message Passing Timings
are directly connected to each other) whereas $P_2-P_4$ and $P_1-P_4$ communicate over the crossbars (in this case, transputer links are connected to a common crossbar switch). The transmission rate for $P_1-P_2$ is higher than that for $P_0-P_1$ because the hardware level primitives add little overhead compared to the network level primitives (network level primitives incur the cost of buffering, routing, and packetization). Also, communication on a crossbar is slower compared to one on a direct link (note the time between $P_0-P_1$ and $P_1-P_4$ or $P_1-P_2$ and $P_2-P_4$). Edge $e_{p0p4}$ is dilated and hence a message travels over the multiple links which results into poor performance (i.e., higher message passing time between $P_0-P_4$).

It is possible to use only the network level functions in the above example without any need to know about the mapping details. However to achieve better performance it is required to select the optimal communication primitives based on the configuration information (which includes dilation over the edges and the multiplexing over the physical links) available from the mapping process.

**Workstation Environment**

A network of workstations has evolved into a viable programming environment for parallel applications [37]. A variety of systems are available (§2.1) to program in a networked environment. Some of the motivations for using networked workstations for parallel and distributed computing are:

1. They provide inexpensive computing resources to develop parallel applications and are capable of delivering the performance of a parallel computer [47].

2. They are easy to use and many software tools are available in a familiar environment for program development such as optimizing compilers, debugger, and performance monitors.
3. Network resources can be easily shared. They include peripheral devices, distributed file system, local memory, and computing power of individual workstation.

4. If a workstation remains idle then its otherwise wasted CPU cycles can be harnessed to improve performance.

1.3 Approach

The main objectives of the mapping tool are:

- It should be topology independent, i.e., independent of changes in the underlying hardware and runtime system.

- It should assist the user in generating a good mapping. In an ideal mapping, every pair of communicating processes map onto adjacent processors such that the communication cost is minimum. Usually this mapping is not feasible and hence the mapping tool attempts to optimize the mapping with respect to several criteria.

- Routing determines the path (one or more physical links) between every pair of communicating processes. It is optimal if communication takes place along the shortest path and no two messages are multiplexed over the path. In the case where dilation is greater than one, the mapping tool should generate optimal routing between all non-adjacent processes so as to use the most efficient message-passing level, i.e., it should, where possible, use the lowest level of communication.

The mapping problem is formalized using a graph theoretic model [20, 21, 24, 18]. A parallel program structure is represented as a process graph where each node denotes a user process and an edge indicates that the two processes it joins communicate with each other. Thus parallelism is explicitly specified in the graph. Similarly, the hardware
architecture is represented as a processor graph where each node denotes a processing element (PE) and each edge indicates the communication link between two PEs. With this formalization, the mapping problem is reduced to mapping a process graph into a processor graph. A mapping algorithm is developed that combines heuristic search with backtracking. The mapping takes place in three steps: contraction, assignment, and routing. The mapping information is then passed to the runtime system where it is used to optimize the communication.

A high-level runtime system has been designed on top of Trollius. It includes a message passing library to provide a consistent user interface to all of the Trollius message-passing functions while maintaining the efficient use of low-level functions. The runtime system supports point-to-point and collective communication (scatter/gather operation). The mapping tool statically compiles the program structure into the configuration information. Based on this information, the runtime system selects the optimal Trollius primitives for communication. The optimization is entirely transparent to the user. However, the semantics of the message passing such as synchronous vs. buffered communication are provided by the user.

In the workstation environment, the mapping tool and the runtime system were ported to a network of Sun Sparc workstations. A static load balancing algorithm was developed to assign processes to the processors and a runtime system was designed that uses TCP communication services. Both environments (the transputer system and the networked workstations) share the same user interface and program code and thus location transparency is achieved (i.e., the user can develop an application in either environment and then transparently move the code).
Chapter 1. Introduction

1.4 Thesis Context

Developing a parallel application is a complex process. It includes several steps: program structuring, specifying parallel control and communication, mapping, program loading, runtime support, monitoring, visualization, and debugging. Ideally all of the above should be provided by one integrated environment.

Parsec [32] is one such environment for programming multicomputers which is being developed at UBC. It provides program modules (i.e., skeletons, templates) that are partially implemented virtual machines corresponding to a particular paradigm. Parsec has various constructs to define the structural information required by our optimization tools. It supports a graphical interface to create process graphs and provides three objects, process, port, and channel, to specify explicit parallelism in a program. The mapping tool and the runtime system for message passing are integrated with Parsec to optimize the communication within the environment. Parsec provides an X-window based user interface to the mapping tool and also supports a hardware database which stores information such as the number of processors, processor name, processor speed, and connectivity. In the remainder of the thesis, these tools are discussed in the context of Parsec.

1.5 Thesis Organization

The remainder of this thesis is organized as follows. In Chapter 2, the relevant literature is reviewed. We briefly survey different programming environments, message-passing systems, and mapping tools. Parsec objects which provide structural information to the mapping tool and the runtime system are also described in this chapter.

The mapping tool and the runtime system are integrated with Parsec. Although the tools differ in their implementation on each architecture, they possess the same
functionality and user interface. This integration along with the Parsec communication primitives are described in Chapter 3.

The Parsec mapping tool for a transputer system is presented in Chapter 4. The Parsec runtime system for a transputer system is described in Chapter 5. Native runtime systems (Trollius and Logical C) and optimization techniques employed by the Parsec runtime system are described in this chapter.

Chapter 6 discusses the tool with respect to the workstation environment. In order to demonstrate portability, we also present a parallel application that is developed in the Parsec environment and run on both the architectures. Conclusions are drawn in Chapter 7 followed by a brief discussion of future work.
Chapter 2

Related Work and Background

There are two distinct approaches to developing parallel programming systems. The first approach is to develop a programming environment that provides tools for different phases of program development such as defining parallel code, assigning processes to parallel architecture etc. Usually in these systems, a standard sequential language is extended by providing some message-passing libraries; examples include PVM [42], Topsys [44], Express [46], and Frameworks [47]. A second approach is to develop new parallel languages and compilers based on some high-level programming paradigms. Compilers are responsible for detecting and managing parallelism; examples include Linda [48], Jade [51], and DINO [50].

In Parsec, the parallel programming environment developed at UBC, our goal is to obtain better performance and simplify code development. A mapping tool and runtime system were developed and integrated with Parsec to achieve these goals. Like other programming environments, Parsec helps the user to specify parallel program structures, binds processes to processors and provides primitives for communication and synchronization. However, the Parsec approach differs from many tools in that the mapping tool statically compiles the program structure into information which can be used by the runtime system to select the optimal communication primitives supported by the native runtime system.

In this chapter we briefly survey the related work. In §2.1, we present various parallel programming environments with the main emphasis on communication-related issues.
A compiler-based approach is described in §2.2. The scope of this thesis is limited to a mapping tool and runtime system. In §2.3, we discuss transputer mapping tools (namely, Marc [28], Oregami [31], and Prep-P [29]) followed by message-passing systems for multicomputers such as Zipcode [53] and Via [54] in §2.4. Finally, in §2.5, we describe the three Parsec objects, process, port, and channel, which are used to define the structural information required by our tools.

2.1 Parallel Programming Environment

PVM3

Parallel Virtual Machine (PVM) [42] is a software system that allows a heterogeneous network of parallel and serial computers to appear as a single computational resource. It consists of a set of user-interface primitives, a daemon that runs on every participating machine, and a library for process creation, management, communication, synchronization along with other housekeeping facilities. PVM was primarily designed for networked workstations, however, it has also been ported to the Intel iPSC/860 multicomputer. In the latter case, the native message-passing routines are used for inter-node communication.

In PVM, a programmer is responsible for configuring a virtual machine. As yet, it does not provide load balancing, partitioning, or mapping. PVM supports dynamically reconfigurable runtime system. It includes a few simple message-passing primitives for send/receive, broadcast, and multicast operations. It is the programmer's job to structure and implement parallelism and communication types such as point-to-point communication and group communication. The user is also responsible for buffer management and message packing/unpacking. In terms of communication cost, a daemon on each machine adds to the cost as the message (destined for remote machine) is routed through
the daemons. In general, no communication optimization is provided by the runtime system. In the most recent version of PVM (PVM3.x), the runtime system supports two types of communications: daemon mediated UDP communication and direct TCP communication. Unlike PVM, Parsec runtime system utilizes statically configured structural information to perform optimization. In the workstation environment, Parsec supports only TCP communication. In Chapter 6, the performance of our system is compared to that of PVM3.

**TOPSYS**

Tools for Parallel Systems (TOPSYS) [44] is an integrated, hierarchical tool environment for distributed multiprocessors. The main design goals of Topsys are ease of use and portability. It was originally implemented on an iPSC/860 and later ported to Sun Sparc workstations. Parsec followed similar design approach in that it was originally implemented on a transputer system and later ported to networked workstations. Topsys consists of a parallel debugger, a performance analyzer, a graphical animation tool, and an interface for all of the above tools. It supports MMK (multiprocessor multitasking kernel), a parallel library for process creation and communication. Topsys provides three objects for communication and synchronization: process, mailbox and events. Once the programs are written, the user must specify the connectivity of the objects and the mapping of these objects onto the processors. This information is supplied in a special text file called the mapping file. MMK provides only point-to-point communication and no communication optimization is done in the system as the communication objects are defined statically by the user. Good monitoring support is provided with back-end tools like a debugger and a performance analyzer. However, there are no front-end tools (tools to implement parallel programming paradigm, specify parallelism, or perform task allocation and communication optimization) to help during program development.
Chapter 2. Related Work and Background

Enterprise

Enterprise [40] is a graphical programming environment for distributed systems. It provides tools to design, code, debug, monitor, and execute the parallel applications. Enterprise programs run on a network of workstations. A user attaches assets to the sequential code to specify parallelism in a program. The assets are similar to the templates or programming paradigms that represent high-level techniques for exploiting parallelism. The template-based approach was proposed in Framework [47], a precursor to Enterprise. To specify parallelism, Enterprise uses an analogy between the structure of a parallel program and the structure of a business organization. This analogy provides different types of assets (e.g., contracts, departments, individuals), each defining a different way of partitioning an operation across multiple processes. The system automatically inserts the code to handle communication and synchronization into a module based on its attached asset. It also loads the processes on machines for execution and provides limited process migration and dynamic load balancing.

PIE

The Programming and Instrumentation Environment (PIE) [45], developed at CMU, is directed towards generation of performance efficient parallel programs. Like Parsec, PIE provides a level of abstraction between the algorithm and the parallel architecture level. It is called an Implementation Machine (IM) and corresponds to a particular parallel programming paradigm like master-slave and divide and conquer. The pre-coded structures (which encapsulate communication, synchronization and data partitioning) are made available to the user in the form of templates. The user is required to provide only the sequential code. The main emphasis of PIE is on performance monitoring and debugging. PIE does not supports tools for mapping and loading, and a programmer is
Chapter 2. Related Work and Background

expected to know the details of the hardware architecture and the underlying operating system. Message passing is supported by the IPC mechanism of the operating system. Templates in Parsec are similar to the assets in Enterprise and the Implementation Machines in PIE. However, the main difference is that Parsec also supports the mapping and loading tools and the runtime system for optimized communication.

2.2 Parallel Languages and Compilers

High-level parallel languages provide programming paradigms that hide the complexity associated with explicitly parallelizing the code. It is built around a set of abstractions that express a parallel algorithm. The implementation of such a high-level language hides the low-level details such as mapping and presents an operating system and hardware-independent interface to the user. Some parallel languages, such as Fortran 90 [49] and Linda [48] are extensions to existing sequential languages while others such as Jade [51] and DINO [50] are completely new.

Like parallel languages, parallelizing compilers also free programmers from the difficult task of writing low-level parallel code. They are responsible for extracting parallelism from the code and for effectively schedule it on a target parallel machine. These compilers perform static optimization to generate efficient machine code. However, compiler-based approach is restricted to low-level optimization in that only basic send and receive operations are supported. Another problem is that the compilers are not portable across different machines.

2.3 Mapping tools

Many good mapping algorithms have been proposed in the theoretical literature to map processes onto various fixed interconnection networks. They use a variety of searching
heuristics. For example, the pairwise interchange algorithm by Bokhari [17] used probabilistic jumps to solve a mapping problem on a specific array of processors (i.e., the finite element machine). A greedy algorithm by Chen and Gehringer [21] and a mapping scheme (initial assignment followed by pairwise exchange) by Lee and Aggarwal [20] are targeted for hypercube architectures. Simulated annealing [22] and recursive divide-and-conquer strategy [23] based on Kernighan-Lin graph bisection heuristics for hypercubes were proposed by Ramanujam et al. In all these approaches; the topology of the target architecture was fixed, and the algorithms took advantage of this fact.

There is another approach where the mapping tools are designed to accommodate one or more mapping algorithms, each suitable for a particular topology type. Three such tools are described below.

**MARC**

MARC [28] is an automatic configuration tool for mapping parallel programs onto a transputer network. It includes methods for load balancing and communication-optimized process distribution onto arbitrary topologies as well as efficient and secure routing strategies. The tool accepts a program written in Occam [13] and transforms it into a configured program for a fixed network of transputers. It constructs a software-graph from the Occam program and a hardware-graph from the transputer interconnection and attempts to optimize the mapping of the software-graph onto the hardware-graph. The actual mapping takes place on the entire transputer network defined by the hardware-graph. The mapping algorithm uses a distributed method based on diffusion where neighbouring processors exchange information about their load and communication cost and processes are moved based on some optimization condition. In case of a reconfigurable transputer network, the desired topology should already be switched and thus a user is expected to know the hardware configuration. The input program is required in the Occam language
and this limits the scope of MARC to the transputer networks.

**Oregami**

Oregami [31] provides an automatic and guided mapping of parallel programs onto the parallel architectures in order to achieve portability and maximal performance. The system has two key features: (1) an ability to take the advantage of the regularity present in both the computation structure and the interconnection network, and (2) a desire to balance the user's knowledge and heuristics with efficient mapping algorithms. Oregami consists of three components:

1. LaRCS, a language that allows the user to express the input graph. The input graph specifies regularity in the communication topology (i.e. which tasks communicate with each other) and regularity in the execution and communication patterns over time (i.e. phase behaviour of an algorithm).

2. Mapper, a library of mapping algorithms that allow canned mapping and the mapping of regular (like tree, mesh, and ring) and irregular topologies.

3. Metrics, an interactive graphics tool that provides a way for the manual mapping where a user can specify particular process to processor binding.

In Oregami, the hardware is assumed to be fixed and connected by some regular network topology. In contrast, our mapping strategy is aimed for reconfigurable network (which itself is not regular (§4.1)) to map variety of topologies.

**Prep-P**

Preprocessor for Poker (Prep-P) [30] is a mapping preprocessor that takes as input, a set of user processes whose communication graph is described in a Graph Description
Chapter 2. Related Work and Background

Language (GDL). The mapping takes place in three phases: contraction, placement, and routing. A GDL and a compiler (GraphCom) facilitates representation of the input graph in a layout-independent form. The algorithm used for contraction is a local neighbourhood search and the algorithm for placement is a modified version of the Kernighan and Lin circuit placement algorithm [24]. Once contraction and placement take place, Prep-P provides a mechanism for multiplexing mapped processes over a single processor. In Prep-P, the target hardware is a CHiP machine, a configurable message-passing parallel architecture which can simulate variety of network topologies. The dependency on a compiler makes Prep-P difficult to port to other hardware architectures.

Tmap [36], a programming environment developed at UBC, uses Prep-P as a front-end tool for mapping on our transputer system. It also includes a runtime system for communication built on top of Trollius [12]. Like Prep-P, mapping in Parsec takes place using the same three phases. But unlike Prep-P and Tmap, a new technique is used to generate the mapping (§4.3) and the mapping tool is more flexibly incorporated into the system.

2.4 Message-passing Libraries

Most multicomputers have hardware support for message passing. However, it often differs from machine to machine and can sometimes be complex to use. Message-passing libraries have been designed to provide performance and convenient functionality. Given the proliferation of message-passing standards, there have been an attempt to standardize the libraries. MPI [52] (Message Passing Interface) is a standard that includes a large number of functions to support various communication types and semantics.

It is necessary to design platform-specific message-passing library which will support a few easy-to-use primitives for optimized communication. Zipcode [53] is one such system
which also influenced the MPI standard. It is a dynamically configurable runtime system which provides optimized communication on a variety of platform including nCUBE/2, iPSC/860, Delta, CM-5 scalar machine, and a network of Sun Sparc and IBM RS/6000 machines. It introduces an object called *mailer* that encapsulates groups, contexts, and virtual topologies and provides the appropriate scope for all communication operations in Zipcode. Parsec differs from Zipcode in that optimization is done statically and encoded in the form of load-time parameters to the message-passing primitives. Thus it avoids overheads associated with dynamic optimization.

Via [54], a structured message-passing system, has a hierarchy of objects: task class, task set, and task. This hierarchy is useful in many ways. First, it allows the general properties of a task to be defined in libraries. Second, a group of tasks can be renamed to avoid name collision and third, it allows programs to contain groups of tasks which have the same properties but are addressed differently. Each task has a number of named ports. This task and its ports together provide the communication context. To optimize the communication, Via makes use of the configuration information provided by the user. Parsec has similar hierarchy of objects but, unlike Via, the configuration information is automatically generated by the mapping tool and used by the runtime system to optimize communication.

In the following section, we describe Parsec objects used to specify a parallel program structure.

### 2.5 Background

In Parsec, there is a clear distinction between sequential program code and specification of parallelism. The specification captures two notions of communication: communication structure and communication semantics. Communication structure encapsulates
declarative information such as which processes communicate with each other, their connectivity, and communication intensity (number of messages transferred). The semantics specify whether communication is synchronous or buffered. The mapping tool converts communication structure into detailed configuration information needed by the runtime system. The runtime system, based on this information and the user-specified semantics, optimizes communication (this is explained in Chapter 5).

Parsec provides a graphical user interface to specify the communication structure of a parallel program. It uses three primitive types of objects: process, port, and channel. Process, a global object, is the smallest decomposable unit of computation in a parallel application. Port, a local object bound to a process, is used for communication on a channel which is uniquely defined by the pair of process-port bindings. Figure 2.1 shows the relationship between these objects. Processes A and B each have two ports (Next and Prev) associated with them. The two processes communicate on a channel defined by \{A, Next; B, Prev\} association. The port Prev of process A and port Next of process B are unbound and hence cannot be used for the communication.

**Process**

Process is the most widely used type of object that specifies the structure of a parallel program. It is an instance of a *process type*, another object provided by Parsec for
process reuse. Process type has two types of information: construction information and process structure information. Construction information includes source files, libraries, and runtime parameters that are needed to build an executable image of a process. Process structure information specifies the port name, process weight or priority. A new process is created by specifying its type and by binding its named ports to channels. This logical division makes the high-level program structure independent of the actual process instances.

**Port**

In a parallel program, processes are often replicated (multiple instances of a process type) to form a program structure. As with non-replicated processes, these replicas must be bound with explicit connections to other processes. Changing the statically configured program structure requires that these connections are often changed. However, if the process is to remain replicable, its internal specification must remain independent of its external connection. The port object is used to facilitate this isolation. Port is a named variable whose value specifies the data structure that includes configuration information generated by the mapping tool. Thus, port hides the complexities of the underlying communication medium. Ports also encapsulate communication scope. A single port can be bound to several other ports and thus provide a simple way for collective (one-to-many or many-to-one) communication.

**Channel**

Channels define the communication paths in a program. In Parsec, a channel itself has no name but a channel end is defined by the process-port binding. Thus a channel is an association of a process-port connected to another process-port. The channels possess weights which denote the traffic density (the number of messages passing through the
channel). These weights are used during mapping to determine the best possible way to map these channels over the hardware architecture. A channel implementation is hardware-dependent. In multicomputers, channels are implemented over the physical links and on workstations they are built on top of some layered protocol and use the service access points of that protocol (e.g., sockets on TCP/IP protocol).

The process-port-channel model has a close resemblance with our graph theoretic approach in the mapping tool. In a process graph, a node denotes the process-port binding and an edge denotes the channel. Similarly, port internals provide the information required by the message passing primitives for communication optimization. These objects are further discussed in that context in Chapter 4 and 5. Apart from these primitive objects, the Parsec interface has process groups, parameterized process graphs, and modules to support reusability, scalability, and ease-of-use [32] [38].
Chapter 3

System Design

The various programming systems discussed in the last chapter offer different tools and techniques for developing efficient parallel programs. However, the user is responsible for binding a parallel program structure to a parallel architecture. Also, the runtime systems provided by these programming environments do not support automatic communication optimization. In Parsec, the tools are designed to support these features. As described in §2.5, Parsec supports the port-process-channel model for specifying a parallel program.

In this chapter, we present a tool that automatically maps this program structure onto a parallel architecture. We also introduce the runtime system that provides a high-level message-passing library and uses the mapping information to optimize the communication.

In §3.1, we describe the integration of our tools with Parsec. This system presents a conceptual model that defines a generic tool interface and functionality for any type of multicomputer. The mapping tool is described in §3.2. Different phases of the mapping process are defined in this section. In §3.3, the Parsec process model is defined and its message passing primitives are described. Some of the features of the runtime system are directly influenced by the underlying operating system, the network management, and the hardware support for communication. However the primitives described here are generic and hide platform-specific dependencies.
3.1 System Overview

The mapping tool and the runtime system are integrated with Parsec. Figure 3.1 shows their interaction with the Parsec environment. A database is used for data integration between the various tools.

In Parsec, a parallel program is specified in the form of a process graph using a graphical interface. In a process graph, each node denotes a process, each edge denotes a channel, and the connection between the edge and the node specifies a port. This process graph along with the hardware resources such as the number of processors, the processor type, the number of links, and the interconnection of the network are stored in the database.

The mapping tool takes as input the process graph and the hardware resources and performs the mapping in three distinct steps: contraction, assignment, and routing.
Its output is the process-processor bindings and platform-dependent information. The platform-dependent information includes the communication paths between every pair of processes. For the transputer system, these paths specify transputer links, crossbar switches, and crossbar links. For networked workstations, the paths may specify bridges or routers. The mapping tool can be viewed as a static compilation tool that converts structural information into detailed platform-specific configuration information. This configuration information is stored into the database for further use. A loader tool provides the database interface between the mapping tool, the runtime system, and the program execution. It passes the optimized configuration information in the form of load-time parameters to the runtime system. A loader also generates the script to boot the network that is configured by the mapping tool and to load the application program onto the multicomputer.

The Parsec runtime system has a few simple message-passing primitives that are built on top of the native runtime system. These primitives hide the complexity associated with low-level communication while maintaining the efficiency of the low-level primitives. For the transputer system, the Parsec primitives select one of the many Trollius functions provided at different levels (physical, data-link, network, and transport level). In case of workstations, they use a IPC mechanism supported by the operating system. The runtime system has two distinct phases: an Initialization phase and an Optimization phase. During initialization, the configuration information is converted into the data structures corresponding to the ports (§2.5) so that the optimal primitives can communicate on those ports. During the optimization phase, the port data structure is analyzed and the most efficient low-level function is selected. As we will see in Chapter 5, this requires that the physical links are reserved for the lowest-level of message passing. As shown in Figure 3.1, the user can also specify communication semantics such as synchronous and buffered communication. A timeout mechanism is also provided that allows user
to control blocking of every send and receive operation on a per port basis. Finally, the optimized message-passing primitives are compiled and linked with the application program to form an executable image.

3.2 The Mapping Tool

Methodology

The mapping problem is related to two basic problems: task allocation and routing. Stone [25] proposed a network flow algorithm for task assignment in order to minimize interprocess communication. Various other approaches pursued are: simulated annealing [22], heuristics [26], recursive bipartitioning method [23], and initial assignment, followed by pairwise exchange [20] or boundary refinement [27]. Since obtaining optimal solutions is intractable, these algorithms are not guaranteed to produce the optimal solution.

We have proposed a graph-oriented mapping strategy that uses a greedy algorithm combined with heuristics to find a solution. Greedy algorithm were used by systems like Prep-P and in the theoretical literature on mapping [17, 21].

The mapping problem is represented as one that relates a process graph, corresponding to a parallel program structure, to a processor graph, corresponding to a parallel architecture. A node in a process graph indicates a user process and an edge indicates a communication path between the two adjacent processes. The cost of Communication between two processes is indicated by the weight associated with every edge of the process graph. In the case of non-adjacent nodes, the communication path is a sequence of such edges. Similarly in a processor graph, a node represents a PE and an edge is the physical link between two PEs. Each link of a processor graph is bidirectional so that the underlying processor graph is undirected.
Functional Overview

The mapping process has the following phases: resource allocation, contraction, assignment and routing. Dividing the mapping problem into different phases allows one to provide different solution to each individual phase. Thus it is possible to implement different algorithms, each best suited for a particular task, for different phases of the mapping process. Same approach is taken by the Prep-P [30] mapping tool. However, the difference (with respect to Parsec mapping tool) lies in the tasks performed in each of these phases. For example, the routing phase in the Parsec mapping tool analyzes the assignment and generates the configuration information which is required by the runtime system for communication optimization. The basic inputs to the mapping tool are the hardware resources and a process multigraph (multigraph is an undirected graph that can have multiple edges between the vertices and self-loops).

1. Resource Allocation

This module determines which ports (hardware ports; e.g., port that connects transputer to the host machine) and processors to use, and which processors are connected to the host machine (i.e. root processors). The following information is required by this module: different logical components of the system and how they are connected, the number of processors, the machine-type of the processors, the number of processes and their machine types. A hardware graph is built from this information and is used during assignment and routing.

2. Contraction

If the number of processes is more than the number of processors available, then more than one process will need to run on each processor. This task is performed by contraction. The process graph is contracted in such a way that the number
of processes is less than or equal to the number of processors in a target machine. This is similar to partitioning a process graph of \( n \) nodes into \( m \) groups, \( m \leq n \), such that a given cost function is minimized. An example cost function is the communication cost over the edges. Several algorithms are available to perform the contraction including: simulated annealing, local neighbourhood search and branch-and-bound. However, to keep the mapping problem simple, we ask the user to specify which processes need to be assigned to the same processor. The input to this module is a process multigraph and output is a abstract process graph where each node is a virtual process which represents one or more user processes. This is many-to-one mapping from the user multigraph to the abstract process graph.

3. Assignment

This process assigns a single virtual process to a single processor of the target machine. This is a one-to-one assignment. The assignment strategy is driven by the type of algorithm used for mapping and its main objective is to minimize the total cost of communication in the process graph.

4. Routing

Once the node assignment is done, there are two levels at which routing takes place: physical routing and logical routing.

(a) physical routing : in a dynamically reconfigurable system, the interconnection network is defined by setting the control switches between the processor links. All the physical links between every pair of processors are statically configured. This results into a fixed interconnection graph.

(b) logical routing : from a fixed interconnection graph, the communication path
between all the non-local processes are determined. The channels of the abstract process graph are then assigned to the physical links in the interconnection graph.

5. Dumping Information

This phase gathers all the information generated in previous phases and stores it into the Parsec database so that the loader tool and the runtime system can access it. This information includes physical wires, routes, type of communication, and objects for synchronization. The contracted user processes within a virtual process are invisible to the assignment and the routing phase. Hence their routes are generated separately and stored into the database.

Mapping strategy is platform-dependent. We present a mapping strategy for the transputer system in Chapter 4 and for the workstation environment in Chapter 6.

3.3 Runtime System

The Parsec runtime system is built on top of the native runtime system. The implementation details are platform-dependent but the message-passing primitives are generic. We first present the Parsec process model which specifies the communication and synchronization characteristics and then describe the Parsec primitives.

3.3.1 Parsec Process Model

As described in §2.5, a parallel program in Parsec consists of a set of processes. A process is an entity that performs internal, unobservable actions and also interacts with other processes. An interaction implies synchronization among multiple processes. Such a synchronization is an observable action and may involve data exchange. The Parsec
process model allows multiple processes to synchronize and thus support one-to-many (scatter) and many-to-one (gather) type communication. The communication can be either synchronous or buffered. In the case of multi-process synchronization where a single process can receive data from more than one processes, the synchronization behaviour is non-deterministic in that the sender’s address cannot be known in advance.

The runtime system implements this model using processes, ports, and events that correspond to processes, ports, and channels — three static objects that are defined in Parsec. The approach is similar to that of Zipcode [53] and MPI [52] where it is characterized in terms of contexts of communication and communicators. A process is defined earlier in this section. A process group is a collection of processes and defines the scope of collective (scatter/gather) communication. It also provides an efficient mechanism to define and alter the characteristics of the processes, such as port bindings and process weights. We now describe the other two objects: events and ports.

Events

Events are the identifiers used for synchronization. They are globally unique integers, generated by the mapping tool. There is at least one event associated with every channel. The events are introduced in Parsec because they are required by the Trollius functions (described in Chapter 5). The Logical C functions do not use events. They access the communication channels directly. In the workstation environment, events are mapped to the port addresses used for communication over the sockets (described in Chapter 6). The user never manipulates these events but only observes the synchronization behaviour implied by the events.
Chapter 3. System Design

Ports

Ports are similar to mailers in Zipcode and communicators in MPI and define the context of communication. The ports encapsulate the configuration information generated by the mapping tool and make it available to the runtime system for optimization. The information includes the physical addresses of the processors that embed all the members of a process group and the events for every channel accessible through the port. The physical address and the events together provide scope for all the communication operations in Parsec.

3.3.2 Message Passing Primitives

The runtime system supports two message-passing primitives: port_send and port_recv. They are used to select the optimal communication primitives provided by the native runtime system. All ports are declared and bound to a process through the Parsec's graphical interface. Before any message is sent or received, every user process must initialize all its ports. This is accomplished by the following function:

```c
init_ports(int argc, char **argv, Port *p1, Port *p2, ..... NULL);
```

where p1, p2, .. are the names of the ports that are bound to a process. The names can be either defined by the user or defined by Parsec. Function init_ports reads the configuration information passed by the loader tool and stores it into the data structures corresponding to the ports. Other initialization tasks are specific to the native runtime system. For example, in our transputer system init_ports registers a process with Trollius kernel and allocates soft channels (described in Chapter 5) for local communication. Once the communication is over, a process is exited with the following function call which essentially frees all the port structures (and in the case of transputer system it also de-registers the user process):
Processes send or receive data on a specific port by invoking two functions: `port_send` and `port_recv`, respectively. Timeout parameters are used to control the blocking of send and receive operations. Two parameters, one for the send operation and one for the receive operation, are defined for each port. The user can also specify whether the communication is buffered or synchronous. The default options are: zero timeout (i.e. blocking until the send or receive returns successfully) and buffered. Like the configuration information (generated automatically by the mapping tool), these user-controlled parameters are encapsulated in the port structures. The syntax of the port functions is as follows:

```c
port_send(Port *port, void *data, int size, int msgtype);
port_recv(Port *port, void *data, int *size, int msgtype);
```

where `port` is a pointer to the port to (or from) which process is sending (or receiving) the data. Parameter “msgtype” is either a predefined token RAW or a user-defined token for XDR data structures. The data structures can be primitive data types or user-defined structures. Parsec automatically generates XDR code to pack and unpack the data structures and perform the byte-ordering when needed. If `port_send` and `port_recv` are called with user-defined message type then the the XDR code does the appropriate data conversion. For all XDR messages, the size argument in both the functions should be NULL. The data argument is a pointer to an instance of the data structure for the send function and a pointer to a variable of the message type for the receive function. If the data structures contain the elements that are accessed through pointers, they are allocated by the XDR code if the size pointer is NULL. In the case of the RAW message type, the XDR code is bypassed. For these messages, the data argument is a pointer to the existing buffer for both send and receive function. The `size` argument is the actual size of the buffer in the case of send and in the case of receive, it is set to the number of
bytes that are received.

Features of the runtime system are summarized as follows:

1. The communication primitives are easy-to-use and portable across different hardware platforms.

2. They select the optimal communication primitives depending on the configuration information generated by the mapping tool. This type of optimization is platform-dependent and is described in §5.3.

3. They support point-to-point communication as well as scatter and gather type of communication.

4. They support data conversion for heterogeneous networks. The basic data types as well as the user-defined complex data structures can be used.

5. Messages of arbitrary length are allowed, and the packetization is done internally.

In Chapter 5, the runtime system for the transputer system is described. This includes the underlying communication primitives and the optimization techniques employed by the runtime system. Finally, the Unix implementation of the runtime system is discussed in Chapter 6.
Chapter 4

Mapping Strategy on Multicomputers

Following the general design of the tool given in §3.2, we describe our mapping strategy on multicomputers. The mapping algorithm has been implemented on our configurable transputer system.

A brief overview of the transputer system in Department of Computer Science at the University of British Columbia is presented in §4.1. In §4.2, we describe the structural design of the mapping tool. The structural design shows different components of the mapping tool. These components provide a layer of abstraction on top of the hardware architecture and the parallel program structure. This allows the mapping to be done in a hardware-independent way. The mapping strategy is described in §4.3. We discuss various cost metrics, an objective function, a mapping algorithm, and the experimental results of our mapping tool.

4.1 Hardware System

The hardware system consists of 75 INMOS T800 transputers and 10 C004 programmable crossbar switches. Each transputer has a 32 bit, 10 MIPS CPU, a 64 bit FPU, 4KB of on-chip RAM, and four bit-serial communication links. Each node is a single transputer with external (off-chip) memory: 64 nodes have one MByte, 10 nodes have two MByte and there is one node with 16 MByte. Each crossbar switch has 32 data links and a control link. Any pair of data links can be connected by sending the appropriate software command to the control link. Crossbars can have up to 16 bidirectional links.
The system is reconfigurable and supports a variety of interconnection networks.

The transputer system is accessed by a Sun Sparc2 workstation through a S-bus card with four data ports. The nodes are partitioned into four reset components. Each component has one transputer directly connected to the host through a data port. Trollius, a single-user, multitasking operating system runs on each node. This allows only one user to access any node at a given time. However, the logical partitioning allows up to four users to concurrently use the system. Once a particular component is allocated, the user has complete control over all the transputers in it.

The entire transputer network resides in five physical boxes\(^1\). Links 0 and 1 of most transputers are hardwired to one another whereas links 2 and 3 are connected to the crossbar links. Some transputers in box 4 have all the links connected to the crossbars.

There are six 8-node rings and one 8-node chain. The remaining transputers are either connected to the host machine or to the crossbars. Thus, the network is not regular. Figure 4.1 shows the part of our transputer network which is configured as described above.

\(^1\text{There are four components partitioned into five physical boxes as shown in Appendix A.}\)
4.2 Structural Overview

The different components of the mapping tool are shown in Figure 4.2. The components are connected by directed edges that show the dependency between them. In the rest of the chapter we will use the terminology defined in this section to describe the various graph structures used in the mapping process.

1. Hardware Database (HWDB)

The hardware database includes the machine dependent information on the nodes and their connectivity. A node is either a processor, a crossbar, or a host machine. The node information includes the logical name of the node, its architecture type (either transputer, crossbar, or host machine), the component to which it belongs, the number of communication links, memory size, processor speed. A transputer link is either hardwired to another transputer link or connected to a crossbar link.
In the latter case, connections between the transputers depend on the crossbar settings. Thus, multiple physical links (Figure 4.3) are possible between two transputers depending on the number of hardwired links and the number of crossbar switches. All such transputer interconnections are stored in the database. Given two transputers and their links, a database query can determine whether there exists a path between them.

2. Resource Allocation Module (Resources)

This module retrieves information from the hardware database. The module allocates the resources from the component requested by the user. The resources include the number of wires (network connections), a data structure for each wire (which includes transputer names, transputer links, current crossbar settings, and weight on each wire), the names of all the processors (i.e., the transputers and the host machine), and the names of all the transputers that are connected to the host machine. Only these resources are used by the remaining phases of the mapping tool.

The four hardware components of our system are shown in Appendix A. As an example consider component 0 which has 25 transputers \((T_0-T_{24})\) and \(T_a\). In Figure 4.3, we have shown the connections between \(T_a\), \(T_{15}\), and Xanadu (the host machine). The data structures for the wires, retrieved by the resource allocation
module, are given in Table 4.1.

<table>
<thead>
<tr>
<th>Wire #</th>
<th>PE₁</th>
<th>Lk₁</th>
<th>PE₂</th>
<th>Lk₂</th>
<th>Wt</th>
<th>Xlen</th>
<th>Xname</th>
<th>Xlk₁</th>
<th>Xlk₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>w₁</td>
<td>Tₐ</td>
<td>1</td>
<td>T₁₅</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>w₂</td>
<td>Tₐ</td>
<td>2</td>
<td>T₁₅</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>xb₇</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td>w₃</td>
<td>Tₐ</td>
<td>3</td>
<td>T₁₅</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>xb₇</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>w₄</td>
<td>Xanadu</td>
<td>0</td>
<td>Tₐ</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>w₁'</td>
<td>T₁₅</td>
<td>0</td>
<td>Tₐ</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>w₂'</td>
<td>T₁₅</td>
<td>3</td>
<td>Tₐ</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>xb₇</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>w₃'</td>
<td>T₁₅</td>
<td>3</td>
<td>Tₐ</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>xb₇</td>
<td>15</td>
<td>17</td>
</tr>
<tr>
<td>w₄'</td>
<td>Tₐ</td>
<td>0</td>
<td>Xanadu</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4.1: Wire Structures

Each wire is unidirectional. If two transputers are hardwired then xlen is 0 indicating that there are no crossbar connections. In the case of positive xlen (i.e., wire has an intermediate crossbar links), the name of the crossbar switch and its links are stored into the data structure. Wt indicates the communication cost on a wire. For the crossbar links this cost is double the cost on a direct link. The processes running on Xanadu and Tₐ are specified by the user and are assigned to the processors during initial phase of the mapping algorithm. Hence the communication cost between Xanadu and Tₐ is not considered for the mapping.

3. System Graph

A system graph is a multigraph representing the interconnections of the transputer network. The system graph for the processors in Figure 4.3, minus Xanadu, is shown in Figure 4.4. Each node represents a transputer, and an edge between two nodes represents a physical link between two transputers. Each transputer has four links. If all four links of the two transputers are connected to the same crossbar, then there are 16 different ways to connect them. Therefore, the system graph has up
to 16 edges between two nodes. Each edge is associated with two data structures (as shown in Table 4.1) for two unidirectional wires.

\[
\begin{align*}
e_1 &= \{w_1, w'_1\} \\
e_2 &= \{w_2, w'_2\} \\
e_3 &= \{w_3, w'_3\}
\end{align*}
\]

Figure 4.4: System graph

The main advantage of using a system graph is that it hides the hardware-specific information. This graph can be used as an input graph to any mapping algorithm. This is evident from Figure 4.2, in that there is no direct edge from hardware database to an algorithm module. A system graph is independent of a particular mapping instance (i.e. process graph). It can be constructed once and stored in the database for reuse, unless the hardware configuration or the requested resources change.

4. Conflict-Graph

The system graph stores the communication paths between the nodes. There are, however, the potential for conflicts. A conflict graph is constructed from the system graph to encode the information about which pairs of system edges can not be mapped simultaneously. In Figure 4.4, there are three edges between \(T_a\) and \(T_{15}\): \(e_1\), \(e_2\), and \(e_3\). Since \(e_2\) and \(e_3\) share link 3 of \(T_{15}\), only one of them can be used for the mapping. In general, two edges

\[
e = < n_1, l_1k_1, n_2, l_2k_2 > \quad \text{and} \quad e' = < n'_1, l'_1k'_1, n'_2, l'_2k'_2 >
\]

conflict if and only if
Chapter 4. Mapping Strategy on Multicomputers

\[(n_1 = n'_1) \lor (n_2 = n'_2) \land (lk_1 = lk'_1) \lor (lk_2 = lk'_2)\]

A conflict graph for Figure 4.4 and its matrix representation are shown in Figure 4.5. The graph shows a conflict between \(e_2\) and \(e_3\) and adjacency matrix has

\[
\begin{array}{ccc}
  e_1 & e_2 & e_3 \\
  e_1 & 0 & 0 & 0 \\
  e_2 & 0 & 0 & 1 \\
  e_3 & 0 & 1 & 0 \\
\end{array}
\]

Figure 4.5: Conflict-graph for Figure 4.3

value one in the corresponding row and column. Like a system graph, a conflict graph depends only on the hardware configuration and is independent of the mapping instance.

5. Abstract Processor Graph

The abstract processor graph or Host graph is a subgraph of the system graph. Different host graphs are constructed for different mapping instances. Before the mapping starts, there are as many nodes in the host graph as the transputers in a component requested by the user. Since the crossbar switches are reconfigurable, their link information is not available until the mapping is finished. Hence only hardwired links (i.e. wires between two PEs without any intermediate crossbar switches) are added to the host graph initially. Once the mapping is done, physical routing takes place on paths between every pair of transputers that are statically configured. Configuration information such as transputer links, crossbar switches and their links is stored during routing. The transputers that are not mapped and are not connected to any mapped transputers are removed from the host graph. Thus, after the mapping is done, it is guaranteed that the host graph is a connected graph that embeds the process graph.
6. Process Graph

A parallel application is specified in the form of a process multigraph where each node represents a process and an edge between two nodes indicates a communication channel. It is assumed that communication channels are unidirectional, one-to-one, and named on each end. This is called the "port name" of the channel. Thus a channel endpoint is defined by a tuple <process_name, port_name> and a channel is an association of two such tuples. The type of communication (e.g., one-to-one, one-to-many, and many-to-one) is also characterized in this graph. An example of a process graph is shown in Figure 4.6.

The above process graph has four processes and each process has two ports bound to it. These processes communicate on eight different channels: 

- {<A, east>, <B, west>}
- {<B, south>, <D, north>}
- {<D, west>, <C, east>}
- {<C, north>, <A, south>}
- {<B, west>, <A, east>}
- {<D, north>, <B, south>}
- {<C, east>, <D, west>}
- {<A, south>, <C, north>}

There is only one-to-one communication since no two channels have the same end-points (i.e., process/port pair).

7. Abstract Process Graph

The abstract process graph or Guest graph is the same as a process graph shown in Figure 4.6 except that if contraction occurs, each node of the guest graph denotes
a virtual process (a virtual process is a process with more than one user process mapped onto it). An edge stores (logical) routing information. Once the mapping is done, logical routing determines the paths of any non-local communication wherein the channels as given by the process graph are assigned to the physical links in the host graph.

The contracted process graph for Figure 4.6 is shown in Figure 4.7. X and Y denote the virtual processes. Once the node assignment is done, processes A and C will be mapped to the same processor and the same is true for processes B and D.

8. Mapping Algorithm

Given a guest graph and a system graph, the mapping problem is reduced to embedding a guest graph onto a system graph. A one-to-one embedding of a guest graph onto a system graph is desirable. However, designing a generic algorithm for various topologies is intractable. Instead, different algorithms and heuristics for different topologies can generate good mappings in many cases [31]. The graph structures, described above, provide basic objects for any mapping algorithm. These graphs hide the internal details of the parallel architecture and the parallel program structure. In §4.3.3, we present a greedy algorithm that employs heuristics to map several process graphs onto a configurable transputer system (§4.1).

9. Mapping State
Once the mapping is done, the information about embedding, logical routes between all pairs of guest nodes, and wires connecting mapped processors is stored in the database. The loader tool extracts this information from the database, configures the network of the host graph, and generates a script to boot the network and to load the application programs onto the transputers. The mapping information is also used to measure congestion\(^2\) on the hardware channels, to identify the communication type for every edge in the guest graph (i.e., whether the communication occurs on local, neighbouring or remote transputers), and to generate events for synchronization between the processes. This configuration information is then passed to the runtime system where it is used to select the optimal Trollius functions (§5.3).

4.3 Mapping Strategy

The mapping strategy is guided by constraints such as dilation, congestion, load, and expansion and aimed at minimizing the cost of communication. One can formulate these constraints in the form of objective functions. The mapping strategy then attempts to optimize these objective functions.

We first present the mathematical formulation of a mapping problem, followed by the mapping constraints and the objective function, and finally the algorithm.

4.3.1 Mathematical Formulation

Let \( G = (V_G, E_G) \) be a guest graph to be mapped onto a target machine, where \( V_G = \{ v_i \mid v_i \in G \} \) is a set of nodes representing the processes and \( E_G = \{ e_{ij} \mid e_{ij} = (v_i, v_j) \) and \( v_i, v_j \in V_G \} \) is a set of bidirectional edges. The weight of an edge, \( w_{ij} \), denotes the

\(^2\)defined in §4.3.2
traffic intensity. The adjacency matrix for G is denoted by $A_G$.

Let $S = (V_S, E_S)$ be a system graph representing a target architecture, where $V_S = \{ v_i \mid v_i \in S \}$ is a set of processors and $E_S = \{ e_{ij} \mid e_{ij} = (v_i, v_j) \text{ and } v_i, v_j \in V_S \}$ is a set of physical links of the hardware. Let $w_{ij}$ be the weight on edge $e_{ij} \in E_S$. It denotes the communication cost associated with the edge. $A_S$ denotes the adjacency matrix for the system graph.

A system path $p_{ij}$ corresponds to the sequence of system edges when it is defined for any two system nodes. It is denoted as $p_{ij} = (e_{i_k}, e_{k_2}, ..., e_{k_n})$ where $e_{i_k}, e_{k_2}, ..., e_{k_n} \in E_S$. As explained in §4.2, there are conflicts, as given by the conflict graph, between many edges in the system graph. $T$ denotes this conflict graph.

We assume that $|V_G| \leq |V_S|$. $G$ is a abstract process graph defined in §4.2. If there are more processes than processors then the contraction is specified by the user and $G$ is a contracted graph.

The mapping problem has been reduced to an embedding of $G$ into $S$. An embedding maps each vertex in $V_G$ onto a vertex in $V_S$ and each edge in $E_G$ onto an edge (or edges) of $E_S$. The corresponding mapping function is denoted as $f: V_G \rightarrow V_S$.

### 4.3.2 Mapping Constraints

The following cost metrics are considered to measure the quality of the mapping: load, congestion, dilation, and expansion. They are discussed in detail in [4]. The objective function for our optimization is derived from these measures.

- **Load**: Load is the number of nodes in a process graph that are embedded in a single node in the system graph. In our case, contraction is done manually through the Parsec interface and we assume that only one guest node (virtual process representing one or more user processes) is embedded on a single system.
node.

- **Congestion:** Edge congestion is the congestion over a system edge and is defined as the number of guest edges mapped onto it. The congestion of an embedding is the maximum edge congestion over all the system edges. When congestion is greater than one, the system edge is required by more than one guest edge. Hence communication on these edges is delayed unless there are multiple system edges between the two nodes. Thus communication cost increases with the congestion. The congestion over \( e_{ij} \in E_S \) is given as,

\[
C(e_{ij}) = |E'_G|
\]  

(4.1)

where \( E'_G = \{ e_{mn} | p_{mn} = (e_{mi}, ..., e_{ij}, ..., e_{jn}) \text{ for } e_{mn} \in E_G \text{ and } e_{mi}, e_{ij}, e_{jn} \in E_S \} \) such that \( f(v_m) \) and \( f(v_n) \in V_S \forall v_m, v_n \in V_G \).

- **Dilation:** Dilation of a guest edge is the length of a path (a path consists of one or more system edges) that a guest edge is mapped onto. The dilation of an embedding is the maximum dilation over all the guest edges. When dilation is greater than one, communication over the guest edge may get delayed because the message has to traverse over multiple system edges. In addition there may be congestion over some of the system edges. Thus, like congestion, dilation also adds to the communication cost. The dilation of a guest edge \( e_{ij} \) is given as the distance between \( f(v_i) \) and \( f(v_j) \). We denote this distance as \( D_G (f(v_i), f(v_j)) \).

- **Expansion:** Expansion is the ratio of the number of nodes in a system graph to the number of nodes in a guest graph. The expansion is related to the dilation of an embedding, and optimizing one factor can adversely affect the other [4]. The expansion of the embedding does not directly affect the cost of communication. Our mapping strategy attempts to minimize only the dilation and the congestion.
In Parsec, the expansion is determined by the user. The number of nodes in a guest graph depends on the contraction of a process graph which is specified by the user whereas the nodes in the system graph are limited by the size of the component. However, once a component is allocated, the mapping tool will use as many transputers from that component as required.

Given that the entire communication takes place in a single phase (a phase is the time interval for the communication to take place over a single guest edge), if the edges in a system graph and a guest graph have same weights then the total communication cost is same as the maximum dilation or congestion. However, if the weights on the edges are different then these weights should be taken into account while measuring the communication overhead. The weight on a guest edge indicates the traffic intensity. For example, if one edge is used more frequently than another edge, then it carries more weight. In the system graph, the edges representing crossbar links and direct links have different weights, reflecting the different bandwidth of the two.

With these different weights, the congestion over a system edge, \( e_{ij} \), is measured as the sum of the weights, \( w_{mn} \), of all the guest edges that are mapped on it:

\[
C(e_{ij}) = \sum_{e_{mn} \in E'_{G}} w_{mn}
\]  

(4.2)

where \( E'_{G} \) is defined in equation 4.1. Similarly, the weights on the system edges (instead of number of system edges) should be taken into account while measuring communication cost due to dilation.

The dilation of an embedding is minimum if each of the guest edges maps to a single system edge. Let “dil-one” be the number of guest edges mapped onto one and only one system edge. We call it an “embedding amount” of the mapping function \( f \) and define
it as
\[
\text{dil-one} = \frac{1}{2} \sum_{v_i, v_j \in V_G} A_G(v_i, v_j) \times A_S(f(v_i), f(v_j))
\]
(4.3)

This equation counts the number of times an adjacent pair of guest nodes, \(v_i\) and \(v_j\), remain adjacent when mapped to \(S\), the system graph.

Depending on the guest graph, one of the following two cases arises with any mapping strategy:

**Case 1 :** \(\text{dil-one} = |E_G|\)

**Lemma :** If the embedding amount, dil-one, is equal to the number of edges in a guest graph then the dilation of an embedding is 1.

**Proof :** We are given that \(\text{dil-one} = |E_G|\). Hence from equation 4.3, it is clear that for every edge \(e_{ij} = (v_i, v_j) \in E_G\) (i.e., for every pair of nodes, \(v_i\) and \(v_j\), adjacent in \(G\)) there exists \(f(v_i)\) and \(f(v_j)\) adjacent in \(S\).

We know that the mapping distance \(D_G(f(v_i), f(v_j)) = 1\) if and only if \(f(v_i)\) and \(f(v_j)\) are adjacent in \(S\), and the dilation of an embedding is defined as

\[
\max_{v_i, v_j \in V_G} \{D_G(f(v_i), f(v_j))\}.
\]

In this case \(D_G(f(v_i), f(v_j)) = 1\) \(\forall v_i, v_j \in V_G\).

This proves that the dilation of an embedding is 1 when the dil-one is same as number of edges in a guest graph. \(\Box\)

If there are multiple edges between the same pair of guest nodes then the congestion can be greater than one while the dilation is equal to one. The mapping algorithm will map multiple guest edges on the multi-edges of a system graph in order to minimize the congestion. Since the system edges have different weights, these weights must also be considered while mapping in order to minimize the communication overhead.
**Case 2**: $\text{dil-one} < |E_G |

If $\text{dil-one}$ is less than number of edges in the guest graph, then the dilation and hence the congestion are greater than one. We assume that entire communication takes place in a single phase in that all guest edges are required simultaneously. Hence the total cost of communication is the maximum of the communication overhead on any guest edge. The communication overhead on a guest edge depends on the congestion, dilation, and the weights on the system edges and the guest edges. The mapping strategy should take into account all these factors in order to minimize the communication overhead. The communication overhead, $CO$, on the guest edge $e_{ij}$ is given as

$$CO(e_{ij}) = \max_{e_{mn} \in p_{ij}} \{C(e_{mn}) \times w_{mn}\} \quad (4.4)$$

where $C$ is defined in equation 4.2, $e_{ij}$ is dilated over the system path $p_{ij}$, and $w_{mn}$ is the weight on the system edge $e_{mn}$. The expression in braces indicates the communication cost associated with an individual edge in the system path. The maximum cost over all the edges in the system path gives the communication overhead on the corresponding guest edge.

As an illustration of the mapping strategy, consider the following example. Figure 4.8

![Figure 4.8: (a) Guest graph, (b) Host graph for case 1, (c) Host graph for case 2](image-url)
(a) shows the guest graph to be mapped. The number on each edge indicates the weight associated with it. A system graph represents a dynamically reconfigurable machine. Hence it is possible that different host graphs are generated depending on the resources requested or the guest graphs. Figure 4.8 (b) and (c) show two such host graphs. The numbers on the system edges represent the weights. The guest nodes, shown as circles, get mapped onto the adjacent system nodes, shown as squares.

The mapping shown in (Figure 4.8 (b)) is the best mapping. Here the dil-one is same as the number of guest edges which is equal to six. Hence the dilation of the embedding is one and the congestion of the embedding is also equal to one since there are no multiple edges. The total communication cost is the communication overhead on the guest edge $e_{02}$ or $e_{13}$, which is equal to two. There is one more embedding possible ($s_0[g_0]$, $s_1[g_1]$, $s_2[g_2]$, and $s_3[g_3]$) which also results in minimum dilation and congestion. However in this case, the communication cost is four.

If dil-one is less than the number of edges in guest graph then the dilation and the congestion are greater than one. One such possible mapping is shown in Figure 4.8 (c). Only four out of six guest edges are directly mapped onto a system graph and guest edges $e_{03}$ and $e_{12}$ are dilated. To minimize the total cost, the mapping strategy tries not to dilate the guest edges with higher weights. In this example, there are three edges at guest node $g_0$ and only two edges at system node $s_0$ on which $g_0$ is mapped. Hence one of the three guest edges must be dilated. However, the guest edge $e_{02}$ has more weight and hence is mapped on system edge $e_{01}$ and guest edge $e_{03}$ with less weight is dilated. Having done this, routes for the dilated guest edges $e_{03}$ and $e_{12}$ must be determined.

The guest edge $e_{03}$ can be dilated over system path $(e_{01}, e_{12})$ or $(e_{03}, e_{32})$. Given that $e_{03} \in E_G$ is dilated over $(e_{01}, e_{12})$ where $e_{01}, e_{12} \in E_S$, from equation 4.4, communication overhead on $e_{03}$ is given as,
CO (e_{03}) = \max \{ C(e_{01}) \times w_{01}, C(e_{12}) \times w_{12} \} \\
= \max \{ (3 \times 1), (2 \times 2) \} \\
= 4

Similarly, CO(e_{03}) is 4 if the guest edge e_{03} is dilated over (e_{03}, e_{32}) where e_{03}, e_{32} \in E_S.

In the case of equal overhead, the first route is selected. Assume the guest edge e_{03} is dilated over (e_{01}, e_{12}). While mapping guest edge e_{12}, one should take into consideration the guest edge e_{03} that is already mapped. It can be easily verified that for the guest edge e_{12}, communication cost over system path (e_{30}, e_{01}) is four and over (e_{32}, e_{21}) is six. Hence the guest edge e_{12} is dilated over (e_{30}, e_{01}).

In this section we discussed various constraints and how they affect the quality of mapping. In the next section the mapping algorithm is presented that implements the strategy described here.

### 4.3.3 Algorithm

Both the guest graph and the system graph are multigraphs and have different weights on their edges. It is not sufficient to optimize any one cost metric. We try to optimize the objective functions, dil-one, defined by equation 4.3. The heuristics are combined with a greedy algorithm to maximize the dil-one and to minimize the communication overhead by considering the actual weights on the edges. The algorithm employs recursion and backtracking techniques for an efficient search through the system graph.

The input to the algorithm is G (Guest graph), S (System graph), and T (Conflict graph) and the output is the physical routes, stored in H (Host graph) and logical routes, stored in G. The algorithm carries out two steps: node assignment and routing. During the node assignment, dil-one is maximized in that a one-to-one edge mapping is sought between the guest edges and the system edges. If there are multi-edges in the guest graph
then the algorithm seeks multi-edges in the system graph. To minimize the overhead, the
guest edges with more weights are mapped onto the system edges having smaller weights
(i.e., the direct links).

If the dilation of an embedding is greater than one then during the routing phase,
the logical routing is done for dilated edges such that the communication overhead, CO
(equation 4.4), is minimum. The physical and logical routing information is then stored
in the host and the guest graph respectively.

The algorithm defines the "first guest node" and the "first system node" as follows:
The "first guest node" can be specified by the user. It is either the root of the guest graph
or the host process (the host process runs on the host machine and must be specified
by the user). If neither is specified then the algorithm selects the guest node having
minimum degree. In the case of the system graph, either the host machine or the root
transputer, retrieved by the resource allocation module, is selected as the "first system
node". The mapping starts by assigning the "first guest node" to the "first system node".
Thereafter at every level of recursion, algorithm finds the number of to-be-mapped guest
edges at the last mapped guest node and seeks that many non-conflicting, unmapped-
edges at the corresponding system node.

The system graph represents the entire search-space and the algorithm attempts to
find a graph which is isomorphic to the guest graph (i.e., to maximize the dil-one). As
we will see in Chapter 5, this allows the runtime system to make use of the lowest-level
communication primitives to minimize the communication cost. The pseudo-code for the
algorithm is presented below.

Input : G, S, T, H
Output : node embedding, H and G with physical and logical routing
begin /* main */
1. LastMappedVg := Vg := { First node in G };
Chapter 4. Mapping Strategy on Multicomputers

2 \[ \text{UnMappedV}_g := \{ \text{all nodes in G} \} - \text{V}_g; \]
3 \[ \text{LastUsedVs} := \text{Vs} := \{ \text{First node in s} \}; \]
4 \[ \text{UnUsedVs} := \{ \text{all nodes in s} \} - \text{Vs}; \]
5 \[ \text{level} := 1; \]
6 \[ \text{map}[\text{V}_g] := \text{Vs}; \]
7 \[ \text{DoEmbedding(level, LastMappedV}_g, \text{LastUsedVs}); \]
8 \[ \text{SetPhysicalRoutes(G, H, map}); \]
9 \[ \text{DoLogicalRouting(G, H, map}); \]
end

\begin{verbatim}
begin /* DoEmbedding */
7a \ for \text{V}_g \text{ in LastMappedV}_g \text{ do} \\
7b \ \text{Get EdgesToMap at V}_g; \\
7c \ \text{Get adjEdges at V}_s (= \text{map}[\text{V}_g]); \\
7d \ \text{Sort adjEdges in ascending order of their weights}; \\
7e \ \text{Check whether adjEdges provides at least one set of |EdgesToMap| non-conflicting system edges to map EdgesToMap}; \\
7f \ \text{EdgeList} = \text{GetTuples(EdgesToMap, adjEdges)}; \\
7g \ \text{if EdgeList} = \{ \text{return(failure)}; \\
7i \ \text{for tuple in EdgeList do} \\
7j \ \text{for edge in tuple do} \\
7k \ \text{Insert NewV}_g \text{ in NowMappedV}_g; \\
7l \ \text{Insert NewVs in NowUsedVs}; \\
7m \ \text{Delete NewV}_g \text{ from UnMappedV}_g; \\
7n \ \text{Delete NewVs from UnUsedVs}; \\
7o \ \text{map}[\text{NewV}_g] = \text{NewVs}; \\
7p \ \text{Update S}; \\
7q \ \text{endfor} \\
7r \ \text{if UnMapped} = \{ \text{return(success)}; \\
7s \ \text{if (DoEmbedding(level+1, NowMappedV}_g, NowUsedVs) = success) \\
7u \ \text{break}; \\
7v \ \text{endfor} \\
7w \ \text{Update S}; \\
7x \ \text{endfor} \\
end
\end{verbatim}

Several aspects of the algorithm are discussed below. The numbers correspond to the line numbers shown in the pseudo-code.
- **Lines 1-6:** As previously discussed, the initial node assignment is done.

- **Line 7:** DoEmbedding function is called recursively to maximize dil-one. For each mapped guest node, the following procedure (Lines 7a-7x is repeated. At any given time, the edges in a system graph are in one of the three states: Avail, CantUse or Used, meaning a particular system edge is available (Avail) for mapping, or a guest edge is mapped on it (Used), or it cannot be used (CantUse) for mapping because of the conflict with already Used system edge.

Initially the state of all the system edges is set to Avail.

  - **Line 7b:** EdgesToMap, a set of unmapped guest edges at the last mapped guest node, is derived from the guest graph. |EdgesToMap| indicates the desired size of dil-one at the guest node.

  - **Line 7c:** Similarly, adjEdges, a set of Avail system edges at the corresponding system node, is constructed.

  - **Line 7d:** The set adjEdges is sorted in ascending order of weights so that the edges with less weight get mapped first. In particular all the direct links of a transputer are mapped before the crossbar links.

  - **Line 7e:** Check is made whether adjEdges has at least one set of non-conflicting system edges. This feasibility check detects the possible failure before any mapping is done and thus backtracks at early stage of recursion.

  - **Lines 7f-7h:** GetTuple function finds all possible sets (EdgeList) of non-conflicting system edges from adjEdges, each of size |EdgesToMap| or less. EdgeList is a list of tuples each having |EdgesToMap| or less system edges. All the edges in each tuple are Avail and non-conflicting. This list is a search-space at a particular level of recursion. EdgeList is sorted in descending
order of tuple size. Thus the tuples with |EdgesToMap| system edges are tried (to maximize dil-one), before the smaller tuples (which results into greater dilation).

- **Lines 7i-7q:** The algorithm then maps EdgesToMap guest edges onto the first tuple from EdgeList. The system graph is then updated. The edges in this tuple are marked as Used and all the conflicting edges are marked as CantUse.

- In the case of backtracking, the status on every system edge that is unmapped, is restored, at the previous level the next tuple from EdgeList is mapped and the process continues.

- **Line 8:** Once the node assignment is done, all the system edges with Used status are copied to the host graph. All the paths between every pair of transputers in the host graph are statically configured (i.e., the transputer links, crossbar switches, and crossbar links are stored on the edges in the host graph).

- **Line 9:** During logical routing, the routes between every pair of guest nodes are constructed. When the dilation is greater than one, the paths between non-local nodes are constructed so as to minimize the communication overhead (§4.3.2).

**Mapping Analysis**

The node assignment and the routing specify the configuration information required by the loading tool and the runtime system (§3.1). This information includes the process-to-processor bindings, physical links between the processors including current crossbar settings, and the routes for all the communication bindings. The mapping tool analyzes the logical routes to identify their communication characteristics. The characteristics include multiplexing over the physical links (measure of congestion) and the scope of
each communication binding (i.e. whether the communication is local, neighbour-to-neighbour, or non-neighbour). On the transputer system, the Parsec runtime system uses Trollius events as synchronization objects (described in Chapter 5). The mapping tool assigns these events to each route characterizing point-to-point and collective communication. In the case of user-specified contraction, the mapping tool also generates events for all the intra-processor communication.

4.3.4 Experimentation

As described in §4.1, our transputer network is not regular. A large number of crossbar connections allows the system to be configured for different network topologies. We tested numerous regular topologies (i.e. the guest graphs) including tree (binary, ternary, binomial), mesh, ring, chain, torus, butterfly, and hypercube.

The performance of the algorithm is measured in terms of dil-one, the embedding amount. In general, finding the best mapping for a particular guest graph is difficult. However if dil-one is equal to the number of edges in the guest graph, then the dilation of an embedding is one and the solution is definitely the optimal solution.

The mapping tool can be run in two modes: normal and quick. In normal mode, it backtracks upon failure and continues to search for an optimal solution. In quick mode, backtracking is disabled to guarantee a solution for any type of guest graph. The GUI for the mapping tool is described in Chapter 6. Table 4.2 shows the mapping results for various topologies of different sizes. The measurements are taken on a Sun Sparc10 machine. The time taken for the mapping and the value of dil-one are shown for both modes. A column $|E_G|$ lists the total number of bidirectional edges for each guest graph and size shows the cardinality of the guest graph.

The algorithm neither assumes any hardware interconnection pattern nor does it take advantage of the topology type (tree, mesh, hypercube etc.) of the guest graph.
However, the following heuristic (which assumes that the transputer system resides in different components) is employed to expedite the search in the system graph: at any given mapped guest node, there are $|\text{EdgesToMap}|$ guest edges to be mapped. $\text{EdgeList}$ is a list of tuples each having $|\text{EdgesToMap}|$ or less system edges. In any tuple, all the system edges have one common node that is already mapped and the other node is unmapped. Tuples having their unmapped nodes in different components get higher priority during mapping.

| Topology | Size | $|E_G|$ | Normal Time (in Sec) | Normal dil-one | Quick Time (in Sec) | Quick dil-one |
|----------|------|--------|----------------------|----------------|---------------------|--------------|
| Chain    | 32   | 31     | 1.84                 | 31             | 1.68                | 31           |
|          | 64   | 63     | 9.85                 | 63             | 9.86                | 63           |
| Binary   | 8    | 7      | 0.54                 | 7              | 0.58                | 7            |
| Tree     | 16   | 15     | 0.81                 | 15             | 0.94                | 14           |
|          | 32   | 31     | 5.84                 | 31             | 3.33                | 27           |
|          | 64   | 63     | 72.56                | 63             | 15.32               | 44           |
| Ternary  | 14   | 13     | 12.14                | 13             | 0.84                | 11           |
| Tree     | 41   | 40     | -                    | -              | 10.58               | 22           |
| Mesh     | 3x3  | 12     | 1.00                 | 12             | 0.63                | 11           |
|          | 4x4  | 24     | 16.44                | 24             | 0.90                | 20           |
|          | 5x5  | 40     | 7563.59              | 40             | 6.76                | 31           |
|          | 6x6  | 60     | 222230.0             | 60             | 9.78                | 36           |
|          | 8x9  | 127    | -                    | -              | 19.96               | 90           |
| Butterfly| 12   | 16     | 2.55                 | 16             | 2.54                | 16           |
|          | 32   | 48     | -                    | -              | 5.93                | 35           |
| Hypercube| 8    | 12     | 4.48                 | 12             | 0.67                | 11           |
|          | 16   | 32     | 1.23                 | 22             | 1.28                | 22           |
| Shuffle  | 8    | 10     | 4.13                 | 10             | 0.65                | 9            |

Table 4.2: Mapping Statistics

As shown in Table 4.2, the algorithm performed reasonably well for the trees but
not for meshes. As graph size increases, the time required in the normal mode grows exponentially. The mapping tool could not map a 8x9 mesh, a ternary tree of depth four and a three dimensional butterfly. Our heuristic worked efficiently for those topology types and sizes for which there were more than one solution. In quick mode, the solution is found for all the topologies, but at the expense of smaller dil-one (i.e., greater dilation of an embedding). Apart from these topologies, the mapping was tested for different variations of regular topologies (e.g., complete graph, binary tree with leaves connected to its root) suitable for various parallel programming paradigms.
Chapter 5

Runtime System for Transputers

The runtime system, described in Chapter 3, consists of a high-level message-passing library. It selects the optimal communication primitive from a set of primitives supported by the underlying message-passing system. The process model, described in §3.3.1, and the port primitives, described in §3.3.2, are common to any type of multicomputer. However, their implementation-specific features are influenced by the underlying hardware architecture, the operating system and the communication support. The runtime system must hide these system level details and provide simple primitives without compromising performance. In this chapter, we describe the Parsec runtime system developed for the transputer-based multicomputer.

The underlying process model, supported by a transputer, is described in §5.1. The Parsec runtime system is implemented on top of Logical C [7, 8] and Trollius [9]. These low level software tools are presented in §5.2. In §5.3, we describe the Parsec runtime system. The optimization techniques and implementation level details of different communication types are described. Finally, the performance of the Parsec primitives is measured in §5.4.

5.1 Transputer Programming Model

A transputer has four inter-processor communication links to support parallel processing at the hardware level. Two transputers can be connected by their communication links and thus can be used as the basic building blocks for constructing complex parallel
architectures. Communication within a single transputer occurs through the internal channels, implemented as words of memory. Each transputer has a built-in process scheduler with a process context switching time of less than one microsecond (for the INMOS T800-25 model). The communication links operate concurrently with CPU and thus data transfer on those links takes place with little effect on CPU performance.

Occam [14] is a parallel programming language that reflects the transputer architecture. The Occam programming model consists of a set of parallel processes communicating on the channels. Communication between the processes is point-to-point and synchronous. A channel provides a one-way communication path between two processes. At any time, a process may be ready to communicate on one or more of its channels. Channels between two processes on the same transputer are implemented using shared memory, and channels between two processes on different transputers are implemented by the communication links. Non-blocking communication can be implemented by introducing buffer processes on the links.

5.2 Low Level Software Tools

The Parsec runtime system is implemented on top of two software environments: Logical C and Trollius. It uses the communication functions provided by Logical C and Trollius. We describe below the key features influencing our runtime system.

5.2.1 Logical C

Logical C [7, 8] is a transputer toolset consisting of standard C library routines and transputer specific routines. The transputer specific routines are written in transputer assembly language and are very efficient. Functions are provided for process creation, process communication through message passing or shared data and semaphores, and
determining the status of the channels which are essential to implement one-to-many and many-to-one communication. The features that are used in our runtime system are discussed below.

1. Logical C identifies two types of channels: soft channel and hard channel. The interprocess communication between two processes on different transputers takes place through a hard channel, formed by connecting communication links of the two transputers. Communication between two processes on same transputer takes place through a soft channel, which is basically a word of memory. However the routines used for input and output to the channel are the same for both types (e.g., ChanIn/ChanOut, ChanInChar/ChanOutChar, ChanInInt/ChanOutInt etc.).

2. A soft channel is created using the ChanAlloc function and hard channels are pre-defined by the system. There are eight hard channels, one for input and one for output on every physical link of a transputer. They are LINK0IN, LINK0OUT, LINK1IN, LINK1OUT, etc.

3. During a channel operation, two processes must perform operations on the same data size. Thus, if one process writes 2 bytes to a channel on one end and another process attempts to read 1 byte on the other end then one overwrites the other. The runtime system should guard against this type of mismatch.

4. At any given time, a process can be ready to read data from one or more of its channels. A set of function calls are provided to determine the status of the channels and to wait until a channel is ready for input. These functions are ProcAltList, ProcSkipAlt, ProcSkipAltList, and ProcTimerAltList.
The basic channel communication functions (ChanIn, ChanOut etc.) consist of only a few transputer assembly instructions and hence add little overhead. The type of channel (hard channel vs. soft channel) used by these functions depends upon whether the communication is intra-processor or inter-processor. Unlike hard channels, soft channels must be declared and defined before they are used for communication. Hence the user must know the process-processor bindings (mapping information) and the channel information must be hand-coded into a program depending on these bindings. This makes the program development mapping-specific. Also the channel functions support only synchronous, point-to-point, and non-multiplexed type of communication.

5.2.2 Trollius

Trollius [9] is an operating system for multicomputers that provides the user with a wide range of functionality in a hierarchical form. Each trollius node consists of a multi-tasking local environment upon which the rest of the operating system is built. There is a local kernel at each node that synchronizes any number of local messages. Trollius provides three basic objects for communication and synchronization: node identifier, process, and event. The node identifier (nodeid) is a globally unique identifier chosen by the user and represents a single trollius node. The other two objects, process and event, are local to the node. Events are chosen by the user and are used to rendezvous (two processes with identical events always synchronize).

Trollius adds a buffering mechanism to the transputer programming model (§5.1) It supports a wide range of communication services such as synchronization, buffering, flow control, packetization, routing, and data conversion. There are two levels of message passing in Trollius: the kernel level and the network level. The kernel level allows communication between two processes on the same node whereas the network level is used for communication between two processes on different nodes. The kernel level function ksend
attempts to send a message to another process on the same node and \texttt{krecv} attempts to receive a message from another process on the same node. The Trollius kernel provides rendezvous services for the sender and receiver. The network level message passing has four sub-levels representing different functionality and overhead. They are the physical level, the data-link level, the network level, and the transport level in order of increasing functionality and decreasing efficiency. These levels are described below in more detail.

1. **Transport Level**: The transport level functions \texttt{tsend} and \texttt{trecv} provide flow control and process synchronization. The sending process blocks until it receives a "request to send" message from the receiving process. This pre-synchronization is followed by the actual message transfer using network level functions. Non-blocking functions are not available at this level.

2. **Network Level**: The network level functions \texttt{nsend} and \texttt{nrecv} do routing and packetization but do not provide flow control. The sending process returns immediately once the message has left the local node. If the receiving process is not ready, the message is buffered at the receiving side. In that sense \texttt{nsend} is loosely blocking. The receiving process will block if there is no message with the same event in the local buffer. There are two non-blocking variants called \texttt{ntry\_send} and \texttt{ntry\_recv} where the message is either immediately transferred or an error is returned.

**Virtual Circuits**

Each Trollius node runs multiple processes. They are responsible for buffer management, memory management, file management, routing, and communication on the links. When a sending process sends a message at the transport level or at the network level, it passes through the buffer and link processes on both the source
and the destination node, before being delivered to the receiving process. This extra message copying incurs extra overhead. In order to obtain better performance, the trollius processes and other user processes can be bypassed by establishing *virtual circuits* between sending and receiving processes. A virtual circuit reserves the links between any two user processes on any two processors. No Trollius process or other user processes can access the reserved links. As a result, the processes on the intermediate nodes cannot communicate once a virtual circuit is set up between two non-adjacent nodes.

Virtual circuits are also available locally as well. The kernel level functions can bypass the local kernel once an initial rendezvous has been made.

3. **Data-link Level**: The data-link level functions `dsend` and `drecv` are similar to network level functions when used for non-neighbour message passing except that they do not support packetization and routing. Hence the user must know information about the forwarding link process and perform packetization in the code. They are more efficient than the network level functions when used only for nearest-neighbour message passing using the virtual circuits.

4. **Physical Level**: The physical level message passing functions `psend` and `precv` directly use physical links between the transputers (i.e., they are used only for nearest-neighbour communication). These functions can be used only after the virtual circuit is established using the data-link or network level functions. There is no protection against the sender/receiver length mismatch.

The transport and the network level functions are simple yet powerful and together support several communication services (flow control, packetization, routing, etc., as mentioned above). However, performance requires functions at the data-link or physical
level which in turn requires detailed understanding of the operating system and the hardware.

Trollius and Logical C together provide a rich set of message passing primitives with varying semantics and efficiency. The transport level and network level functions are easy to use and their behaviour is deterministic. Multiple processes can access the transputer links simultaneously and hence debugging is possible with these functions. However they introduce considerable overhead. We refer to these primitives as the high-level primitives. On the other hand, the data-link level and physical level functions are efficient but difficulty lies in setting up the virtual circuits. We refer to them as the low-level primitives.

5.3 Runtime System

The Parsec runtime system is built on top of Trollius operating system. It is possible to include Logical C library with Trollius to make channel functions available from within the Trollius environment. In the remainder of this chapter, we refer to the channel functions as a subset of the Trollius functions.

The Parsec message passing primitives, `port.send` and `port.recv` (§3.3.2), select the optimal communication functions from among the Trollius functions described in the previous section. The selection criteria depends on the following factors:

- Communication scope (local/neighbor/non-neighbor)
- Semantics (buffered/synchronous)
- Debugging switch

The mapping tool analyzes the routing information to identify the scope of each communication binding. Local communication (i.e., all the processes running on same node) is
specified by the user. The congestion over the physical links (i.e., the multiplexing over the links) and the process-to-processor assignment define the scope of non-local communication. If the congestion is equal to one then the virtual circuit can be set up on the link. This allows one to use the low-level communication primitives to improve performance. If the congestion is greater than one (meaning more than one user processes read and/or write over the physical link) then the virtual circuits can not be set up even if the processes lie on the adjacent processors. In such case, the high-level primitives must be used. This is exemplified in §1.2.

The semantics of a program are controlled by the user. The user can specify, through the Parsec graphical interface, whether the communication is buffered or synchronous. It is specified either at the port level or at the application level (i.e., specified for all the ports). The blocking time for every send and receive can be controlled by the timeout mechanism. A timeout of zero indicates that the process will be blocked until the sending process or the receiving process returns successfully, and a positive timeout (t) indicates that process will be blocked for t micro seconds and return either with a success or an error.

With the low-level primitives the debugging is difficult as the virtual circuit allows only two user processes (that established the virtual circuit) to access the link. Hence for debugging purposes one can use the high-level communication primitives. This is facilitated by Force_Debug switch. Thus the user can debug a program using high-level primitives and later remove the switch to use low-level primitives where possible.

In addition to the routing information, the runtime system uses the following information generated by the mapping tool: the physical address of each transputer (nodeid), the event (positive integer), and the forwarding link number. For point-to-point communication, the mapping tool generates one event per port. For collective communication, there is one event for the port and one event for each route accessible from that port.
These events are assigned to the Trollius events and thus used for rendezvous. Once an appropriate Trollius function is selected, the events, the nodeids, and the link numbers are required by an individual function to configure internal control structures. For example, network level functions require nodeid and event whereas hardware level functions require the address of the forwarding link.

**Decision Tree**

Figure 5.1 shows the mapping between `port_send/port_recv` and the Trollius functions.

```
port_send / port_recv
     \____ Force_Debug  \____ Optimized
          \____ Buffered  \____ Synchronous
                           \____ Buffered
                               \____ Synchronous
                                    \____ nsend / nrecv
                                    \____ tsend / trecv
                                    \____ nsend / nrecv
                                                  \____ (ChanIn/ChanOut,
                                                  ksend/krecv,
                                                  dsend/drecv)
                                                  or
                                                  nsend / nrecv
```

Figure 5.1: Decision Tree

With the buffered semantics, only network level functions can be used irrespective of whether the communication is optimized or not. For the optimized and synchronized communication, type of the Trollius function depends on the communication scope. It uses both the high-level (nsend/nrecv) primitives and low-level (ChanIn/ChanOut, ksend/krecv, or dsend/drecv) primitives. The following section describes the implementation details of point-to-point and collective communication.
5.3.1 Implementation Details

The configuration information and the user-specified semantics information are made available to the runtime system by the loader tool as command-line parameters. During the initialization phase, this information is stored into the data structures corresponding to the ports. The process is attached to the Trollius kernel (using \textit{kinit} function), and the communication channels (hard channels and soft channels) are set up for neighbour-to-neighbour and local communication. The optimization is performed at runtime. The Parsec primitives, \textit{port\_send} and \textit{port\_recv}, access the information through the port data structures on which they communicate. The port information is then used to select the Trollius primitives.

The runtime system supports point-to-point and collective (one-to-many and many-to-one) communication. In either case, two communicating user processes can reside on the same node, on adjacent nodes, or on remote nodes. Similarly the processes, all participating in collective communication, can be mapped onto one or more of these three locations. The configuration information, required by the Trollius functions, is different in each of these cases. The high-level primitives are easy-to-use. \textit{Port\_send} and \textit{port\_recv} directly call network level and transport level Trollius functions. However, to use the channel primitives, it is required to set up virtual circuit (for neighbour-to-neighbour communication) and to broadcast the channel addresses to all processes (for local communication). The design decisions in the case of local and neighbour-to-neighbour communication are influenced by the following Trollius-specific features.

- For maximum efficiency, \textit{ChanOut}/\textit{ChanIn} are used instead of \textit{psend}/\textit{precv} (for neighbour-to-neighbour communication) and \textit{ksend}/\textit{krecv} (for local communication). At the hardware level, there is no guard against message length mismatch and hence the length of the message is sent first followed by the message. Thus
The virtual circuit must be set up using data-link level or network level functions. \texttt{PortSend}/\texttt{PortRecv} uses data-link level functions to set up virtual circuit. The first message transfer takes place using \texttt{dSend/dRecv}. The \texttt{KHOLD} flag is set during the first message transfer. This is the notification to the Trollius kernel to bypass all system processes and reserve the physical link for the user processes. Once the virtual circuit is set up between neighbouring nodes, subsequent messages are transferred using \texttt{ChanIn/ChanOut} on hard channels. There is no limit on the length of message that is transferred using \texttt{ChanIn/ChanOut}. However, data-link level functions do not provide packetization and the message length is limited to 4k bytes. (The message larger than 4k bytes is lost). Hence the data-link level functions are modified to allow packetization. These modified \texttt{dSend/dRecv} are used to set up virtual circuits.

- The virtual circuits between any two nodes are unidirectional. The runtime system sets them in one direction. This allows two pair of processes to communicate, one pair communicating in either direction, over the same physical link.

- The local communication takes place using soft channels. A soft channel is implemented as a word of memory. All the processes reading from and writing to any channel must know the address of that channel. This is accomplished as follows: Initially each process sends the addresses of all its soft channels (created by \texttt{ChanAlloc} during the initialization phase) to all the communicating processes.
using \texttt{ksend}. Then each process receives, using \texttt{krecv}, the addresses of the channels where the messages are sent. After this initial address transfer, the message transfer takes place using \texttt{ChanIn}/\texttt{ChanOut} on the soft channels.

- The status of the channels is determined using \texttt{ProcAlt*} functions (§5.2.1). A process can be ready for input from one or more channels. These functions block until one of the channels is ready for the input or a specified time is elapsed (in later case, -1 is returned). In local communication, two soft channels are defined between every pair of communicating processes: one for the input and one for the output. Since they are implemented as shared memory, the input channel for one process is the output channel for the other. Each process creates the channel and sends its address to its peer. It is required that the process creates the input channel and receives the address of the output channel from its peer. This is because only those channels that are created locally can be used in \texttt{ProcAlt*} functions.

**Point-to-point Communication**

Table 5.3.1 shows the various Trollius functions used by \texttt{port_send} and \texttt{port_recv}.

<table>
<thead>
<tr>
<th>Communication Type/Scope</th>
<th>Parsec Primitives</th>
<th>Synchronization Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>\texttt{port_send}</td>
<td>\texttt{port_recv}</td>
</tr>
<tr>
<td>Synchronous</td>
<td>\texttt{tsend}</td>
<td>\texttt{trecv}</td>
</tr>
<tr>
<td>Buffered</td>
<td>\texttt{nsend}</td>
<td>\texttt{nrecv}</td>
</tr>
<tr>
<td>Neighbour only</td>
<td>\texttt{dsend/}</td>
<td>\texttt{drecv/}</td>
</tr>
<tr>
<td>Local only</td>
<td>{\texttt{krecv, ksend, ChanOut}}/\texttt{ChanOut}</td>
<td>{\texttt{krecv, ksend, ChanIn}}/\texttt{ChanIn}</td>
</tr>
</tbody>
</table>

Table 5.1: Trollius Function Invocation
The high-level primitives use nodeid and event for rendezvous. For neighbour-to-
neighbour communication, the first message is transferred using dsend/drecv (modified
to allow packetization) to set up the virtual circuit. Subsequent message transfer takes
place by using channel primitives only. For local communication, ksend and krecv are
used to broadcast the addresses of the soft channels. Following this, the communication
takes place using channel primitives.

Collective Communication

Collective communication is implemented using basic point-to-point communication. It
supports one-to-many (scatter) and many-to-one (gather) type of communication. The
port data structure stores the nodeids and the forwarding links of all the destination nodes
(transputers). It is possible that some of the processes, participating in the collective
communication, run on the local nodes while some run on the adjacent and/or remote
nodes. Each port send and port recv calls different Trollius functions depending on the
location of the destination processes. Based on these locations (with respect to the given
process) the following communication types are defined:

- **Local only**: All the processes are on the same node. Like point-to-point commu-
unication, initially the addresses of the soft channels are exchanged between every pair
of communicating processes using kernel-level functions followed by data transfer
using channel primitives. However the major difference lies in the events used by
the Trollius kernel for rendezvous. With multiple local processes, different events
are required by every pair of processes. Therefore in the case of collective com-
unication, the mapping tool generates one event for each port and one event for
each route accessible from that port. We refer to them as “port event” and “route
event”.
• **Neighbour only:** All the destination processes are on the adjacent nodes. First the virtual circuits are set up between all the neighbouring nodes and then subsequent communication takes place using channel primitives. As for the “local only” case, it is required that “route events” are used in the data-link level functions for every adjacent node.

• **Local and neighbour only:** The destination processes are on the local and adjacent nodes (no process on the remote node). As with “neighbour only”, the virtual circuits are set up with the adjacent nodes, and the hard channels are used for communication. However, local communication uses kernel level functions only (no soft channels are used). Both the events are used for local communication. The messages are sent to all the local processes on the “route events”, and they are received from the local processes on the “port event”.

• **Non-neighbour:** Some destination processes are on the remote nodes while some are on the local and/or adjacent nodes. The local communication is similar to the local communication in “local and neighbour only”. The non-local communication takes place using network level functions in that “port event” is used for rendezvous. It is important to note that Trollius does not allow the use of the same “port event” in the network level and the kernel level functions. For this reason, the value one is added to “port event” (a positive integer) used for local communication.

The first two types (“local only” and “neighbour only”) use low-level primitives. In both cases, an initial handshake is required between every pair of communicating processes. From the user’s perspective this means that all the processes should call `port_send` or `port_recv` at least once before any data transfer takes place. The same is true for “local and neighbour only” since it involves setting up virtual circuits with the neighbours.
This limitation is due to the handshake performed during first invocation of \texttt{port\_send} and \texttt{port\_recv} calls for each communication binding.

5.4 Performance Analysis

We now measure the performance of the Parsec primitives and compare it with that of the Trollius functions. Depending on the mapping done, the \texttt{port\_send} or \texttt{port\_recv} invokes one of the Trollius functions described earlier. In this experiment the port primitives are forced to use different Trollius functions for communication between the same pair of processes. This is accomplished by changing the configuration information by hand (in that only the communication flag is changed which selects the Trollius functions according to the decision tree given in Figure 5.1).

Two processes are mapped onto two adjacent transputers which are connected by a direct link (i.e., no crossbar connection). In each case, 1000 messages of the desired size are sent from one process to another and received back. The RAW message type is used in the Parsec primitives and hence the cost associated with data conversion is avoided. The message-passing delays for the Trollius functions are calculated using the statistics presented in [12]. In both the cases the round-trip timings are measured for different communication levels and are shown in Table 5.4.

At the network and transport levels, both Trollius and Parsec use packets of 4K bytes each. Messages of two different sizes (4096 bytes and 4097 bytes) are sent to measure the cost of packetization. Communication cost of the Parsec primitives using hard channels is compared with the cost associated with physical level Trollius functions. The Parsec primitives add considerable overhead because they use data-link level functions for the first message transfer to set up virtual circuits and channel functions for subsequent message passing. This cost (associated with data-link functions) is included in the
Table 5.2: Parsec: Message Passing timings

<table>
<thead>
<tr>
<th>message size (in bytes)</th>
<th>Parsec Primitives</th>
<th>Trollius Functions</th>
<th>time (in Sec)</th>
<th>time (in Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>Transport Level</td>
<td>Transport Level</td>
<td>7.850</td>
<td>Not Available</td>
</tr>
<tr>
<td>4097</td>
<td></td>
<td></td>
<td>8.834</td>
<td></td>
</tr>
<tr>
<td>4096</td>
<td>Network Level</td>
<td>Network Level</td>
<td>6.007</td>
<td>5.968</td>
</tr>
<tr>
<td>4097</td>
<td></td>
<td></td>
<td>6.959</td>
<td>6.520</td>
</tr>
<tr>
<td>4096</td>
<td>Hard Channels</td>
<td>Physical Level</td>
<td>5.135</td>
<td>2.230</td>
</tr>
<tr>
<td>4097</td>
<td></td>
<td></td>
<td>5.153</td>
<td>2.231</td>
</tr>
<tr>
<td>4096</td>
<td>Soft Channels</td>
<td>Kernel Level</td>
<td>0.798</td>
<td>0.640</td>
</tr>
<tr>
<td>4097</td>
<td></td>
<td></td>
<td>0.799</td>
<td>0.640</td>
</tr>
</tbody>
</table>

measurement. The second reason is that the Parsec primitives send the length of the message prior to the message itself (meaning there are two ChanIn and two ChanOut for each message) since the channel functions do not provide packetization.

Finally, the two processes are run on the same transputer to measure the communication cost on the soft channels. This cost is compared with the kernel level Trollius functions. For the Parsec primitives the cost includes the delay caused due to initial channel-address transfer between two processes.

The Parsec primitives are built on top of Trollius and are bound to add extra overhead. Performance can be improved using efficient implementation mechanism. Our primitives use if-then-else construct to select the optimal Trollius functions. In another implementation of the Parsec runtime system [33] the function pointers are used with our optimization techniques to achieve better performance.
Programming A Network of Workstations

In previous chapters we have seen the Parsec tools developed for the transputer-based multicomputer. Providing parallel programming tools for any one type of multicomputer restricts its wide-spread use as no one type is suitable for all kinds of parallel and distributed applications. Considerable effort is necessary to port these applications to another multicomputer. Hence it is desirable for a programming environment to support different hardware platforms and make the application-development process independent of the underlying architecture and its supporting operating systems.

In this chapter, we describe the mapping tool and the runtime system implemented for a network of Sun Sparc workstations running the SunOS operating system.

6.1 Workstation Environment

A workstation environment consists of a number of workstations, each with its own processor, local memory, and I/O device, connected on a local area network. It provides inexpensive shared computing resources to develop parallel and distributed applications. Networked workstations are easy to use and there are many software tools available for program development such as compilers, debuggers, and profilers.

In recent years, many software environments have been developed to exploit these features [42, 41, 44, 40]. However, writing a parallel application in a shared and multiuser environment is still not a easy task. Parsec currently supports two different hardware platforms: a transputer system and networked workstations. Our main objective is to
provide location transparency so that the user is able to run the same parallel application in either environment. Performance will, however, vary on different hardware architectures as each type is best suited for only a certain class of applications. By confining our scope to providing location transparency, the task of selecting the best hardware platform for a particular application is delegated to the user. A similar approach is pursued in Topsys [44] in that the programming environment, developed for iPSC multicomputer, has been ported to the Sun workstations.

The design strategy and heuristics employed in the mapping tool and the runtime systems are hardware-dependent. We discuss below the characteristics of a workstation environment that influence the design of these tools.

**Processor Allocation**

It is frequently observed that, with a number of hosts connected by a network, there is high probability that some hosts are heavily loaded while others are almost idle. This suggests that if the computation uses the available hosts in such a way that the load on a network is evenly balanced (i.e. the job is executed on a lightly loaded machine rather than on a heavily loaded one) then the system performance can be improved. This form of resource sharing by distributing the workload among the available hosts is known as *load balancing* or *load sharing* [62, 63, 64]. In a workstation environment, the mapping problem can be viewed as a processor allocation problem that tries to allocate processing power in an intelligent way.

Networked workstations provide a multiuser, multitasking environment. Sharing the resources should not restrict other users in the environment. Therefore, the number of processors and the processors themselves may vary from one execution to another. This requires that the mapping tool keep statistics of the current load in the network and distribute the newly arriving jobs to optimize the processor utilization.
Idle workstations

The computing power of the network can be fully exploited by utilizing idle workstations. It is believed that the majority of resources, including powerful workstations, in current networks may lie idle for as much as 95% of the time [37]. The mapping tool should primarily use the idle workstations.

Interprocess Communication

Communication in a workstation environment is managed by the networking software which is a part of the network operating system. In the case of Sun workstations, SunOS provides remote procedure calls (RPC), and socket-based and stream-based interprocess communication (IPC). These services follow well-defined, standardized protocols for different communication characteristics such as flow control, retransmission, error recovery, and routing [16]. These are the lowest level (and hence the most efficient) services available to the user. However, their primitives require a good understanding of the operating system. Therefore, they are not useful to the common user unless an easy-to-use and efficient mechanism is provided.

In general, to develop a parallel application in a workstation environment, the user should address the above requirements. The mapping tool and the runtime system for IPC are designed to meet these requirements and are described in the next section.

6.2 Mapping Tool

The mapping problem in a workstation environment differs from the one in a transputer-based multicomputer. In a transputer-based multicomputer, processes are assigned to processors in such a way that the total cost of communication is minimum. In a workstation environment, the problem is to assign processes to processors in such a way that
the load in the network is evenly balanced.

In terms of graph theory, a network of workstations can be viewed as a complete graph where each node represents a workstation and each edge represents a communication path between two workstations. Thus any two processes on different machines can communicate directly. The user perceives it as a neighbour-to-neighbour communication. In practice, the topology depends on how the network is configured. Typically, there are one or more subnets, each consisting of a few workstations connected by an ethernet, interconnected using gateways or routers. The gateways route data packets from one subnet to the other and communication protocols such as TCP/IP are responsible for the routing. The gateways should be included in the graph (they are like the other nodes except that no user process runs on them) as the processor allocation depends on the load on the subnets as well as on the gateways. However, to keep the allocation problem simple, gateways are not included in the graph which consists of the workstations only.

The mapping tool is designed to assign processes to the workstations, based on the following assumptions:

- In the case where the number of processes are greater than the number of processors, contraction is specified by the user.

- Only one process is mapped onto each processor unless contraction occurs.

- Only the load on individual workstation is taken into account while allocating the processes. The cost of communication between two processes is not considered. This is because it is not possible to determine the communication cost statically. It depends on various factors such as physical characteristics of a machine (e.g. CPU speed), load on each machine (which changes dynamically in the multiuser environment), location of the processes (i.e. either they are on the same machine
or on different machines), and location of the machines (i.e. whether two machines are on the same subnet or different subnet).

A graph-oriented mapping strategy, similar to the one for a transputer system (§4.2), is adopted. A hardware database is maintained that stores the static characteristics of each workstation. These include CPU speed, memory, and internet address. This information is made available to the mapping tool by the resource allocation module. There were three phases in the transputer mapping tool: contraction, assignment, and routing. The mapping in a workstation environment takes place in two phases: contraction and assignment. Routing is done by the underlying software. The assignment module allocates each process to a processor in such a way that the load in the network is distributed.

A simple greedy algorithm is presented below that maps the guest graph onto the system graph. For each node in the guest graph, the algorithm selects the workstation with the least load and assigns the process to it.

```plaintext
begin /* mapping algorithm */
1 G := { Guest Graph };  
2 S := { System Graph };  
3 UnMappedVg := { All vertices in G };  
4 UnUsedVs := { All vertices in S };  
5 while UnMappedVg <> {} do
6   for Vg in UnMappedVg do
7     for Vs in UnUsedVs do
8       Load := GetLoad(Vs);
9       end do
10     BestVh = GetBestNode(Load, UnUsedVs);
11     Delete Vg from UnMappedVg;
12     Delete Vs from UnUsedVs;
13   end do
14 end do
end
```
The various aspects of this algorithm are discussed below:

1. This is a static load balancing algorithm. All the processes are assigned to the processors before the job is started. The assignment decision is made deterministically and takes into account the average behaviour of the system.

2. `GetLoad` function (Line 8) computes the load on each machine. The load index used to measure the load is the *load average* which gives the average number of processes in a ready queue. The RPC function `rstat` is used to get the kernel statistics. The statistics include the number of processes in the ready queue averaged over 1, 5 and 15 minutes. The weighted average of these three averages is used as the load index.

3. One cannot isolate the physical characteristics of a machine (e.g., CPU speed, memory size) with its dynamically changing load. For example, when the load on two machines varies only slightly, a machine with higher CPU speed may process the job faster even if it has a higher load. In general, this is true but it is difficult to identify the acceptable load variations in order to apply this heuristic. It largely depends on the processor speed of the individual machine as well as the job itself. However, we have divided the load into seven different ranges. The load thresholds for each range are: 
   
   - 
   - 
   - 
   - 
   - 
   - 
   - 
   
   If two machines with different load fall within the same range then the machine with higher CPU speed and large memory is selected for the node assignment. Given a set of machines and load on each machine, `GetBestNode` function (Line 10) finds the suitable machine as described above.

In summary, the Parsec mapping tool for the workstation environment does a simple
processor allocation according to the load status. It differs from the programming environments which are primarily designed for networked workstations. In Framework [47], the user can specify the mapping constraints completely or partially using a graphical interface or may leave processor allocation entirely up to the system. An extensive load sharing facility is provided by Utopia [41] for a large heterogeneous distributed systems. Different algorithms are implemented for managing resource load information, for task placement, and for remote task execution. Some of the load indices used by Utopia are CPU speed, memory, disk I/O, and login sessions. A number of applications have been developed such as a load sharing command interpreter, a distributed batch facility, and a parallel make. The main emphasis of Utopia is on distributed computing, and it uses general purpose communication libraries (such as PVM) for message passing between the processes in a parallel application.

In other programming environments like PVM and Topsys, processor allocation is left to the user. This requires a detailed knowledge of the available resources, and their loads. Parsec provides a simple tool to relieve the user from this difficult task.

6.3 Runtime System

All participating Sun Sparc machines run under SunOS release 4.1.x. SunOS supports two types of IPC mechanisms: Socket-based and Remote Procedure Calls (RPC). Socket-based IPC provides the lowest level communication primitives. A socket is an endpoint for communication (at transport level) and provides a descriptor for I/O over the network [6]. There are several types of sockets such as stream, datagram, raw, and sequenced packet sockets and each provides different services. For example, the stream sockets provide sequenced, reliable, two-way connection based on byte streams. An explicit connection is set up between two processes prior to any message passing. The datagram
sockets support connectionless, unreliable communication. The user should check for flow control, sequencing, and length mismatch. Since it is a connectionless mechanism, the peer's address is required for all send and receive operations. System calls are provided to control the socket operations for buffering, blocking/non-blocking, out-of-band message passing etc.

RPCs are a communication paradigm built on top of low-level networking mechanism like TCP/IP and UDP/IP. They present a client-server model. A client makes a procedure call to send a data packet to the server, when the packet arrives, the server calls a dispatch routine, performs the requested service, and sends back the response after which the procedure call returns to the client. RPC services can be accessed at three different layers with the highest layer entirely transparent to the operating system and the lowest layer directly using the network services. However, RPC is not an attractive choice for group communication as it can not handle communication from one sender to multiple receivers [15]. We have chosen the socket primitives over the RPC mechanism as they are the most efficient and they are available in all Unix systems. This decision is critical to future development of Parsec on heterogeneous machines.

6.3.1 Internal Features

As discussed in §2.5, the user specifies parallelism explicitly by the process, port, and channel objects. A channel specifies a communication path between any two processes. Ports are bound to each process and thus define the endpoint of a channel. Thus ports are high-level abstractions for the sockets and encapsulates the configuration information generated by the mapping tool. The configuration information includes event and physical address of a machine. The port primitives are implemented for reliable communication using TCP sockets.

For TCP sockets, two-way connection must be established between every pair of
communicating processes. The initialization phase sets up the connection between two processes $A$ and $B$ as follows: One process initiates a connection to another on its known network port address. It is required for both the processes $A$ and $B$ to know the network port address in advance. On the transputer system, the events (positive integers) generated by the mapping tool are mapped onto the Trollius events. Similarly, in the workstation environment, the events are mapped onto the network port addresses, used for naming the sockets. However, it is mandatory that only process $A(B)$ will initiate a connection and process $B(A)$ will accept it. Failing this requirement results in a deadlock since both $A$ and $B$ are waiting to accept a connection from each other. The runtime system avoids a deadlock by having the mapping tool generate a unique event (network port address) for every process. Process $A$ will either initiate a connection or accept it with its neighbours depending on whether its event is less than or greater to its neighbour’s event. In the case of more than one neighbour, they are sorted in the ascending order of their events and then a connection is established with each neighbour sequentially in ascending order. Function $\text{init\_port}$ performs this task. It must be called by each process prior to any message passing. Since system calls $\text{accept}$ and $\text{connect}$, used by $\text{init\_port}$, are blocking, $\text{init\_port}$ returns successfully when the TCP connection is established.

Once the initialization phase is over, $\text{port\_send}$ (write system call) and $\text{port\_recv}$ (read system call) are used for message passing. Currently, sockets operate only in blocking mode and thus support only synchronous communication. With TCP implementation, the kernel tries to batch-up the requests by waiting for some minimum amount of data to communicate. We have used the $\text{TCP\_NODELAY}$ feature to avoid this stream-oriented batching and thus reduce the overheads. A scatter and gather type of communication is simulated on top of basic point-to-point communication. Once the communication is over, all the socket descriptors are closed by calling $\text{exit\_port}$ before exiting the process.

In the next section the performance of the runtime system is measured and compared
Chapter 6. Programming A Network of Workstations

with PVM3 [42].

6.4 Performance Analysis

Two Node test

This example measures the communication time between two Sun Sparc workstations in a local area network using Parsec. Both the machines run a network file system and are part of the Department's shared computing environment. A message of desired size is sent from one node to the other and received back. The round trip communication time is measured for a large number of iterations.

The same experiment is repeated using PVM3, and the communication time of Parsec and PVM3 are compared. PVM3 (release 3.2.6) supports two types of communication: TCP sockets between every two communicating processes and local daemons which communicate using UDP sockets. The default is daemon-mediated communication. The TCP communication is set up by using a `pvm_advise(PvmRouteDirect)` call prior to communicating. Both the types are used in the experiment. The following tables show round trip communication times for different message sizes ranging from 100 bytes to 1 megabyte, using 100 iterations.

Parsec's communication times are consistently faster than both the PVM3 options for all message sizes and for the same subnet. In the case of daemon-mediated UDP communication in PVM3, each message gets copied, first at a daemon on the source node and second at a daemon at the destination node, before it is delivered to the destination process. This overhead is considerable for a communication-bound example like the one presented here. For the TCP sockets, there is the cost of establishing a TCP connection between every pair of processes. However, this cost is amortized when the total number of communications is large. In PVM3, two processes establish a TCP connection when
### Table 6.1: Parsec: TCP communication

<table>
<thead>
<tr>
<th>Msg. Size (Bytes)</th>
<th>Same Subnet</th>
<th>Diff Subnet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Msg. Passing Time (in Sec)</td>
<td>Trans. Rate (Mbytes/Sec)</td>
</tr>
<tr>
<td>100</td>
<td>0.208076</td>
<td>0.048059</td>
</tr>
<tr>
<td>400</td>
<td>0.279717</td>
<td>0.143002</td>
</tr>
<tr>
<td>1000</td>
<td>0.382999</td>
<td>0.261097</td>
</tr>
<tr>
<td>4000</td>
<td>0.948864</td>
<td>0.421557</td>
</tr>
<tr>
<td>10000</td>
<td>2.369537</td>
<td>0.422024</td>
</tr>
<tr>
<td>40000</td>
<td>8.879566</td>
<td>0.450472</td>
</tr>
<tr>
<td>100000</td>
<td>22.012445</td>
<td>0.454288</td>
</tr>
<tr>
<td>400000</td>
<td>88.589821</td>
<td>0.451519</td>
</tr>
<tr>
<td>1000000</td>
<td>222.976151</td>
<td>0.448478</td>
</tr>
</tbody>
</table>

### Table 6.2: PVM3: daemon mediated UDP communication

<table>
<thead>
<tr>
<th>Msg. Size (Bytes)</th>
<th>Same Subnet</th>
<th>Diff Subnet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Msg. Passing Time (in Sec)</td>
<td>Trans. Rate (Mbytes/Sec)</td>
</tr>
<tr>
<td>100</td>
<td>1.265465</td>
<td>0.007902</td>
</tr>
<tr>
<td>400</td>
<td>1.434604</td>
<td>0.027882</td>
</tr>
<tr>
<td>1000</td>
<td>1.572177</td>
<td>0.063606</td>
</tr>
<tr>
<td>4000</td>
<td>2.998988</td>
<td>0.133378</td>
</tr>
<tr>
<td>10000</td>
<td>6.421205</td>
<td>0.155734</td>
</tr>
<tr>
<td>40000</td>
<td>20.890930</td>
<td>0.191471</td>
</tr>
<tr>
<td>100000</td>
<td>50.802238</td>
<td>0.196842</td>
</tr>
<tr>
<td>400000</td>
<td>200.089066</td>
<td>0.199911</td>
</tr>
<tr>
<td>1000000</td>
<td>493.379272</td>
<td>0.202684</td>
</tr>
</tbody>
</table>
they first communicate. This makes the first communication significantly costlier than subsequent communications. In Parsec, the start-up cost is hidden in port initialization as all the TCP connections are established prior to any communication. PVM3 differs from Parsec in that the communication buffers must be explicitly packed and unpacked. This cost is also included in the round trip communication time. In all of the above cases, the transfer rate does not increase for large messages (of the order of hundreds of kilobyte). This is attributed to greater packetization cost and the kernel overheads. For different subnets, Parsec is slower than PVM3 (TCP version) when the message size is greater than 10000 bytes.

In the following section an example is presented to show how program development is done in Parsec using the mapping tool and the runtime system.

### 6.5 Program Development in Parsec

This section describes the Parsec GUI required to create a parallel program structure and to run the mapping tool and the loader tool. The application, called *Spring*, takes
as input any graph and springs it to output a new graph (hence the name *Spring*) which retains the node connectivity of the input graph. The example runs on a ring of eight nodes. One of the nodes, called the root node, is connected to a node, called the host node, outside the ring. The *worker process* runs on each node in the ring and the *host process* runs on the host node. Different phases in the processing are:

- Initially, the *host process* reads a graph (its vertices and edges) from a file and displays it on the GUI. It assigns the vertices to each node in the ring. The graph is then sent to the *worker process* on the root node. The other information, sent to the root node, includes the simulation parameters, number of iterations (N) for simulation and communication, and the vertices assignments. The *Host process* then waits for the result.

- The root node passes the entire data to the next node in the ring and it is then propagated to all the nodes.

- Upon receiving the data, each node carries out simulation, followed by communication (data exchange with the neighbours). During simulation, each node computes new positions of the vertices (based on simulation parameters) assigned to it. The new positions are sent to the adjacent nodes during communication. Both the steps are repeated N times.

- Finally, the new position of each vertex is sent to the host node where the new graph is displayed in the GUI.

Figure 6.1 shows the process graph (eight node ring) for the *Spring* application. A label associated with each node (or process) indicates the process type. All nodes of type *Worker* and *Last Worker* have two ports bound to them: “Prev” and “Next”. The type *First Worker* has an extra port “Host” bound to port “Root” of the Interface
process Type. The details about how to create this parameterized process graph are beyond the scope of the discussion and are contained in [32]. The software and hardware environments can be set up as shown in the Project Information window. Once the

Figure 6.1: Process Graph for Spring in Parsec

process graph is defined and code is linked with each process type, the following steps are carried out:

1. The code is compiled and executable images are produced using the Build Files and Make Binaries options under Tasks choice in the Application menu (refer to Project Tasks window in Figure 6.1).
2. The process graph is mapped onto the transputer network by invoking the mapping tool (Figure 6.2) from the Map Structure option in the Project Tasks window.

(a) As previously discussed in Chapter 4, the mapping tool can be run in two modes: Normal and No Backtracking (or Quick). For some process graphs, the time required for the mapping to complete can be unpredictable. Therefore, the user can set time limit (in seconds) for the mapping tool to run before it terminates with or without the solution. The default time is zero and is ignored by the mapping tool. In this case, the mapping tool continues to run until it finds a solution. In the No Backtracking mode, the time limit is ignored and a solution (not necessarily optimal though) is guaranteed.

(b) The Network Root process is the process that is connected to the Host process. If this option is not set, the mapping tool will select the node (having minimum degree) from the process graph.

(c) The contraction is done manually and the the mapping tool is notified by setting the flag permit contraction. (this is necessary since the configuration information for the contracted processes is generated by the tool).

(d) The transputer network is divided into four components with each component having one port connected to the host machine. The number beneath each component shows the total number of transputers in the component. In this example, component 1 is selected.

(e) Once these options are set, the mapping process is started by pressing Map button. The mapping results (node assignments and edge assignment) are available at the output window.
Figure 6.2: Graphical Interface for Mapping Tool in Parsec.
3. Once the mapping is done, the loader tool generates a file to configure the transputer network (this is called the boot schema file) and to load Trollius and also generates a script file to load user processes onto the hardware and to pass the configuration information to the runtime system. The loader tool (Figure 6.3) is invoked from Load option in Figure 6.1.

Figure 6.3: Graphical Interface for Loading Tool in Parsec
Building makefiles and executable files (Step 1) is independent of the mapping and loading process (Steps 2 and 3). However, the loader must be invoked after the mapping is done as the configuration information, generated by the mapping tool, is utilized by the loader.

The code fragments for each process type are shown below. The main objective is to show how the Parsec primitives are used in the program and hence the computation part is omitted in these fragments.

The following code which runs on each node in the ring uses the conditional macros **ROOT** and **LAST** to differentiate between three process types: **Worker**, **First Worker**, and **Last Worker**.

```c
Param P; /* data structure to send information between nodes */
Port Next, Prev;
#ifdef ROOT /* macro for process type First Worker */
Port Host;
#define in Host;
#else
#define in Prev
#endif

main ()
{
#ifdef ROOT
    init_ports (argc, argv, &Prev, &Next, &Host, NULL);
#else
    init_ports (argc, argv, &Prev, &Next, NULL);
#endif
while (TRUE) {
    set_initial_data (); /* receive parameters from host/previous node */
    for (i=0; i<n; i++) {
        simulate ();
        communicate (); /* send/receive data from adjacent nodes */
    }
#ifdef ROOT
    send_final_position (); /* send result to the host node */
#endif
}
}
```
void set_initial_data()
{
    port_recv (&in, &p, NULL, PARAM);
    ifndef LAST /* macro for process type Last Worker */
    port_send (&Next, &p, NULL, PARAM);
    endif /* initialize the adjacency matrix */
}

void communicate()
{
    pmsg sp; /* data structure to send vertices */
    if (myid % 2) {
        port_recv (&Prev, &sp, 0, POS);
        port_send (&Next, &sp, 0, POS);
    } else {
        port_send (&Next, &sp, 0, POS);
        port_recv (&Prev, &sp, 0, POS);
    }
}

The code for the process type Interface is given below. This process runs on the host node.

Param P;
Port Root;
main()
{
    /* GUI part to display input graph */
    /* read edges and vertices from a file */
    send_parameter (); /* send data to the root node */
    receive_final_position (); /* receive new graph info from root */
    /* send new data to GUI to display new graph */
}

void send_parameter ()
{
    /* initialize p */
    port_send (&Root, &p, NULL, PARAM);
}

void receive_final_position ()
{
    pmsg sp;
    port_recv (&Root, &sp, NULL, PARAM);
}

In order to run the same application on a network of Sun Sparc workstations, the hardware and software environment are changed from the Project Information window (Figure 6.1). Then the application is built again. Depending on the target environment,
Parsec uses different system libraries and macros during compilation and linking. In Figure 6.2, the Host system is the workstation which is assigned as standard I/O for the application. All other options remain the same. The mapping tool assigns processes to different workstations and the loader tool then generates a script to load processes on those workstations. The application was tested in both the environments. The performance of the application in the two environments is shown in Table 6.4.

<table>
<thead>
<tr>
<th>Hardware Platform</th>
<th>Programming Environment</th>
<th>Time (in Sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transputers</td>
<td>Parsec</td>
<td>0.8</td>
</tr>
<tr>
<td>Sun (Sparc)</td>
<td>Parsec</td>
<td>5.2</td>
</tr>
<tr>
<td>Sun (Sparc)</td>
<td>PVM3</td>
<td>20.8</td>
</tr>
</tbody>
</table>

Table 6.4: Spring: Performance
Chapter 7

Conclusions

In this thesis, we examined two major issues influencing communication in multicomputers:

- a mapping of a parallel program onto a parallel architecture
- a runtime system for optimized communication.

A mapping tool and runtime system were developed for a transputer system and for networks of Sun Sparc workstations. These tools are integrated with Parsec, a parallel programming environment for multicomputers being developed at UBC.

The mapping problem in a reconfigurable transputer network is a challenging problem. The mapping tool has been implemented on the 75 node transputer system which supports a wide range of network topologies including binary tree, ternary tree, mesh, hypercube, and butterfly. A graph-oriented mapping strategy is proposed in that a heuristics controlled greedy algorithm is used to map various process graphs. The algorithm attempts to maximize the number of dilation one edges. The mapping tool is a static compilation tool that converts a parallel program structure into detailed platform-specific configuration information. Once the mapping is done, the logical routes are analyzed to identify the communication characteristics such as the multiplexing over physical links and the scope of each communication binding. The configuration information is then made available to the runtime system.
Chapter 7. Conclusions

The Parsec runtime system on the transputer system is built on top of Trollius. Trollius offers numerous communication primitives of varying functionality and performance. Depending on the configuration information, the runtime system selects the most optimal Trollius primitives. If the communication is neighbour-to-neighbour or local and if the physical links are non-multiplexed, then the runtime system uses the most efficient hardware-level communication primitives. Parsec provides two easy-to-use communication primitives: `port_send` and `port_recv`. They support point-to-point and collective communication, employing both synchronous and buffered semantics.

The Parsec primitives are compared with raw Trollius functions. At the network level their performance is comparable to that of Trollius functions. However, at hardware-level the Parsec primitives add extra overhead but relieve the user from setting up virtual circuits.

Both the mapping tool and the runtime system were ported to a network of Sun Sparc workstations to provide location transparency. A static load balancing algorithm has been implemented to allocate processes to processors such that the load is evenly balanced. The runtime system uses reliable TCP communication. The Parsec primitives are compared with the PVM3 primitives. The Parsec timings were consistently faster (15% to 100% faster depending on the message size) than the PVM3 timings except when the machines were on different subnets and the messages were large (of the order of tens of kilobyte). Finally, an example is presented to demonstrate how a parallel application is developed on a transputer network and moved to a workstation environment without changing the code.
Chapter 7. Conclusions

7.1 Future Work

The work can be extended in each of the three areas: mapping problem, runtime system, and tools for different hardware-platforms.

It is unlikely that an efficient algorithm will be found for the general mapping problem. Hence, different algorithms and heuristics can be employed for different topologies. A basic framework for the mapping process has been provided. Various algorithms can be implemented using this framework. By giving different choices for mapping, users can select the best achievable mapping for a wider class of topologies. Our transputer system has programmable crossbars switches in that multiple crossbar can be connected to one another giving better connectivity and thus increasing the range of topologies that can be mapped onto the system. The graph structures presented in Chapter 4 are used for a single crossbar connection between any two transputers. The algorithm can be modified to accommodate multiple crossbar architecture. This requires reconfiguration of the transputer network but has a potential for better mapping results.

Our work can be extended to provide heterogeneity at an architectural level in a workstation environment. Currently, our tools are ported to Sun Sparc workstations and can easily accommodate different platforms like HPs, DECs, SGIs and RS6000s. Since different machines are best suited for different types of applications, incorporating them in Parsec can be an attractive choice to application programmers in various areas including graphics, image processing, numerical analysis and scientific computing. Developing parallel applications in these areas will justify the usefulness of our system.
Bibliography


Bibliography


Mapping related papers


Appendix A

Transputer System Layout

The transputer system in the Department of Computer Science at UBC is shown below. There are four components residing in five physical boxes. Each component has one transputer connected to Xanadu, the host machine, on a data port. The transputers are either hardwired with one another or are connected to the crossbars.

Figure A.6 shows the entire system as a 8x9 mesh with two extra nodes (T_a and T_b). Solid lines are hard-links between two transputers and dotted lines indicate intermediate crossbar connections.

Figure A.1: Component 1 (Box 1)
Figure A.2: Component 0 (Box 2)
Appendix A. Transputer System Layout

Figure A.3: Component 3 (Box 3)
Figure A.4: Component 2 (Box 4)
Figure A.5: Component 2 (Box 5)
Figure A.6: Transputer System as a 8x9 mesh