



THE HONG KONG
POLYTECHNIC UNIVERSITY

香港理工大學

Pao Yue-kong Library

包玉剛圖書館

Copyright Undertaking

This thesis is protected by copyright, with all rights reserved.

By reading and using the thesis, the reader understands and agrees to the following terms:

1. The reader will abide by the rules and legal ordinances governing copyright regarding the use of the thesis.
2. The reader will use the thesis for the purpose of research or private study only and not for distribution or further reproduction or any other purpose.
3. The reader agrees to indemnify and hold the University harmless from and against any loss, damage, cost, liability or expenses arising from copyright infringement or unauthorized usage.

If you have reasons to believe that any materials in this thesis are deemed not suitable to be distributed in this form, or a copyright owner having difficulty with the material being included in our database, please contact lbsys@polyu.edu.hk providing details. The Library will look into your claim and consider taking remedial action upon receipt of the written requests.

AN ENVIRONMENT FOR HIGH-LEVEL, GRAPH
ORIENTED PARALLEL PROGRAMMING

By
Fan Chan

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF PHILOSOPHY
AT
THE HONG KONG POLYTECHNIC UNIVERSITY
HUNG HOM, KOWLOON, HONG KONG
JANUARY 2004



CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written nor material which has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

(Signed)

(Name of Student)

Fan Chan

Abstract

Parallel computing has been used as an important technique to speed up the computations in many different application areas. It also brings benefits in other aspects of performance, such as scalability and fault tolerance. However, programming with parallelized programs is much harder than writing sequential programs. A parallel program consists of multiple processes that cooperate to execute the program. There are many issues that need to address, e.g. communication, synchronization, and load balancing. Also, large-scale parallel applications are not easy to maintain. The ability to develop parallel programs quickly and easily is becoming increasingly important to many scientists and engineers. Therefore, there is a need for a high-level programming models and tools to support the building of parallel applications.

In this thesis, a project is described which aims to provide support for the design and programming of parallel applications in a multiprocessor and cluster environment. The project investigates the Graph-Oriented Programming (GOP) Model to provide high-level abstractions for configuring and programming cooperative parallel processes. Based on GOP, a software environment with various tools has been

developed.

Many parallel programs can be modelled as a group of tasks performing local operations and coordinating with one another over a logical graph, which depicts the architectural configuration and inter-task communication pattern of the application. Most of the graphs are regular ones such as tree and mesh. Using a message-passing library, such as PVM and MPI, the programmer needs to manually translate the design-level graph model into its implementation using low-level primitives. With the GOP model, such a graph metaphor is made explicit in the programming levels because GOP directly supports the graph construct. The programmer can configure the structure of a parallel/distributed program by using a user-defined logical graph and write the code for communication and synchronization using primitives defined in terms of the graph. The GOP runtime has been implemented on MPI with the enhancement on the communication support and high-level programming with the Multiple Program Multiple Data (MPMD) model. It provides a high-level programming abstraction (GOP library) for building parallel applications. Graph-oriented primitives for communications, synchronization and configuration are perceived at the programming-level and their implementation hides the programmer from the underlying programming activities associated with accessing services through MPI. The programmer can thus concentrate on the logical design of an application, ignoring unnecessary low-level details.

We have also built a visual programming interface, called VisualGOP, for the design, coding, and running of GOP programs. VisualGOP applies visual techniques to provide the programmer with automated and intelligent assistance throughout the program design and construction process. It provides a visual, graphical interface with support for interactive graph drawing and editing, visual programming functions and automation facilities for program mapping and execution. VisualGOP is a generic programming environment independent of programming languages and platforms. VisualGOP also addresses the issues on providing graph scalability and support for interoperability by using XML representations of GOP entities and primitives so that it can support the deployment and execution in different platforms.

Example applications have been developed with the support of our GOP environment. It has been observed that the environment eases the expression of parallelism, configuration, communication and coordination in building parallel applications. Sequential programming constructs blend smoothly and easily with parallel programming constructs in GOP. Using the examples, we have conducted evaluations on how GOP performs in comparison with the traditional MPI parallel programming model. The results showed that GOP programs are as efficient as MPI programs.

Publications Arising from the Thesis

1. F. Chan, J. Cao, and Y. Sun. Graph scaling: A technique for automating program construction and deployment in ClusterGOP. In The Fifth International Workshop on Advanced Parallel Processing Technologies (APPT03), pages 254-264, September 2003.
2. F. Chan, J. Cao, and Y. Sun. High-level abstractions for message-passing parallel programming. *Parallel Computing*, 29(11-12):1589-1621, 2003.
3. Fan Chan, J. Cao, A. T.S. Chan, and Minyi Guo. Programming Support for MPMD Parallel Computing in ClusterGOP. *IEICE Transactions on Information and Systems* (Oxford University Press, U.K.), E87-D(7), 2004.
4. Fan Chan, J. Cao, A. T.S. Chan, and Kang Zhang. Visual Programming Support for Graph-Oriented Parallel/Distributed Processing. Revised version submitted to *Software: Practice and Experience*, 2004.

5. Fan Chan, J. Cao, M. Guo, ClusterGOP: A High-Level Programming Environment for Clusters, to appear as a chapter in High Performance Computing: Paradigm and Infrastructure (John Wiley & Sons, Inc), 2004.

Acknowledgements

I take this opportunity to express my sincere and deep felt gratitude towards my supervisor Dr. Jiannong Cao for his invaluable guidance and constant encouragement throughout this work. I would like to thank him for giving me the opportunity to undertake this challenging topic. It was his wonderful association that has enabled me achieve the objectives of this work. The amount of knowledge I have gained from him has been immense.

My parents and my brother Johnson have been a constant source of love and affection throughout. I am eternally grateful to them for always being with me whenever I needed them.

Finally, I wish to thank the following friends: Sandy Wong, Mark So, Ng Kwan Tak, Iris Chow, Jerry Yau, Nash Tseng, Liu Zhen Pong, Edward Lo, Choi Kwok Ho and Edward Tsui.

Hong Kong

Fan Chan

December 31, 2003

Table of Contents

Abstract	iii
Publications Arising from the Thesis	vi
Acknowledgements	viii
Table of Contents	ix
List of Figures	xii
1 Introduction	1
1.1 Problem	3
1.2 Contribution	5
1.3 Organization of the Thesis	8
2 Background and Related Work	9
2.1 Parallel Computing	9
2.1.1 Parallel Computing Applications	10
2.1.2 Parallel Programming Models	13
2.1.3 Cluster Computing	17
2.2 High-Level Programming Support	21
2.2.1 High-Level Programming Tools	21
2.2.2 XML Support for High-level Programming Environment	26
2.3 Visual Programming Environment	27
2.3.1 Graph-Based Visual Languages	27
2.4 Summary	30
3 Graph-Oriented Model and ClusterGOP	31
3.1 The Graph-Oriented Programming Model	31

3.2	ClusterGOP System Structure	37
3.3	The ClusterGOP Library	39
3.4	Programming in the ClusterGOP Environment	44
3.4.1	SPMD Programming Example	44
3.4.2	MPMD Programming Example	50
3.5	Summary	54
4	VisualGOP	55
4.1	The Architecture and Framework	56
4.2	Program Construction	61
4.3	Two-Step Mappings	67
4.3.1	An Example of Mapping	68
4.4	Remote Compilation and Execution	72
4.5	Consistency Check	74
4.6	Graph Scaling	78
4.7	Automatic Mapping	82
4.8	Interpretability	84
4.8.1	Mediator	86
4.8.2	Wrapper	86
4.9	Summary	90
5	Implementation of ClusterGOP	92
5.1	System Architecture	92
5.2	ClusterGOP Runtime Library	95
5.2.1	Basic Library Structure	95
5.2.2	ClusterGOP Primitives	97
5.3	Implementing the MPMD model in ClusterGOP	100
5.3.1	NodeGroup Implementation	100
5.3.2	Support for Data Distribution	103
5.3.3	Automatic compilation and execution support	105
5.4	ClusterGOP Daemon and Runtime Configuration	106
5.5	Summary	110
6	Example Applications	112
6.1	Finite Difference Method	112
6.2	Parallel Matrix Multiplication	117
6.3	Two-Dimensional Fast Fourier Transform	119
6.4	Summary of Results	123

7 Conclusions and Future Work	124
Bibliography	126

List of Figures

2.1	Typical architecture of a cluster of multiple computers	19
3.1	The GOP conceptual model	32
3.2	The ClusterGOP Framework	38
3.3	Grid partitions for 4 processors	46
3.4	Parallel matrix multiplication structure	50
4.1	The VisualGOP architecture	57
4.2	The main screen of VisualGOP	59
4.3	Logical graph for 4 processors in VisualGOP	62
4.4	Editing the LP	64
4.5	Loading and Creating LPs	65
4.6	The GOP Primitive Menu	66
4.7	Manipulating graph nodes directly	67
4.8	Diagram for the parallel matrix multiplication	68
4.9	LP-to-Node and Node-to-Processor mapping	70

4.10	Processor configuration	71
4.11	Node-to-Processor mapping	71
4.12	Automatic code generation framework	72
4.13	Dialog for choosing the host machine to compile	73
4.14	Dialog for entering the compile argument	73
4.15	Dialog for returning the remote command result	74
4.16	Flow chart of automatic consistency check	76
4.17	Participant program code	77
4.18	Validation in a participant program	77
4.19	Updating GOP primitive parameter through the Graph Editing Panel	78
4.20	Graph expansion for mesh structure	80
4.21	Graph Template for the Basic Graph Diagram	80
4.22	Graph expansion for tree structure	81
4.23	Array assignment program in VisualGOP	81
4.24	Mapping Index Table	83
4.25	Layered Architecture or GOP-XML	85
4.26	Program deployment process using GOP-XML	87
4.27	Platform independent structure in ClusterGOP	90
5.1	The ClusterGOP Program Communication Implementation	93
5.2	Choosing Memory Distribution Type	104

6.1	Time required by MPI and GOP programs	114
6.2	Speedups achieved by MPI and GOP programs	115
6.3	Message-passing overhead on GOP	116
6.4	Graph initialization time	117
6.5	Execution time per input array for the parallel matrix multiplication application	118
6.6	Two implementations of a 2-D FFT, the shading area indicates the elements of the array that are mapped to one processor	119
6.7	Diagram for the 2-D FFT program in VisualGOP	121
6.8	Execution time per input array for the 2-D FFT application	122

Chapter 1

Introduction

Traditional sequential programming techniques involve a single computer consisting of a memory connected to a processor via a datapath. This kind of system has performance issues when the program grows large and complex. Over the years a number of architectural and programming innovations have addressed these issues. One of the most important innovations is parallelism. Parallelism is a strategy for more quickly performing the larger and complex tasks in parallel programs. It does this by breaking tasks up into several smaller tasks, assigning these tasks to several workers/processors to work on simultaneously, and coordinating the workers/processors. Parallelism can be used in large applications such as building construction, large organizations, and automobile manufacturing plants, but the creation of software for parallel and distributed systems is difficult and expensive [21], while existing software engineering methods and tools are becoming less effective in the context of new distributed and dynamic hardware features [31]. As a result, systems often contain faults which make

programs difficult to maintain and enhance, and leads to systems which fail to scale as workloads increase.

Many efforts have been made to make programming easier by providing high-level abstractions and programming tools. One such tool supporting visual programming of parallel and distributed systems is provided by graph-based programming environment. In graph-based programming environments a program is defined as a directed graph where nodes denote computations and links denote communication and synchronization between nodes [4]. Providing structured, high-level abstractions can greatly simplify the programming task and reduce development times.

This thesis proposes a graph-oriented programming model (GOP), which is used for the configuration and high-level programming of modular parallel/distributed systems [6, 9]. With built-in support for a language-level, logical graph construct and various operations on the graph, GOP provides an integrated approach to designing and programming parallel and distributed systems. Using GOP, it is possible to represent the configuration of the interacting processes of a parallel/distributed program, as well as the physical network topology, as a user-specified logical graph mapped onto the physical network topology. The programming of inter-process communication and synchronization is supported with the built-in primitives of graph-based operations [7]. Through its mapping to the user-specified logical graph, GOP provides

a high-level view of the message-passing nature of the underlying hardware. This thesis uses the GOP model to address the problem of creating high-level programming environments.

Subsequent sections will define some of the problems in developing parallel applications, discuss the approach that this research will take, and describe the organization of this thesis.

1.1 Problem

The computing model supported by Message-Passing (MP) environments is both simple and very general, and accommodates a wide variety of application program structures. The programming interfaces are deliberately straightforward, thus permitting simple program structures to be implemented in an intuitive manner, yet programmers have encountered three major difficulties when using MP facilities to develop parallel programs, especially for large-scale scientific and engineering computing applications. First, although the concept of the message-passing is quite simple, communication and synchronization of the process of developing parallel applications is not. MPI [45], PVM [24] and other MP interfaces are low-level but nonetheless problematic programming tools. Their interfaces are simple but they require the programmer to deal with low-level details. On the other hand, for non-professional

programmers their functions are complicated and unwieldy. For example, when computing solutions to problems whose data forms a two dimensional matrix, or mesh, it is natural for programmers to arrange each node to communicate with its immediate neighbours in its row or column. When these problems are formulated as algorithms, it is natural to specify the destination of each message in terms of the primary compass directions, rather than using an abstract identifier bearing no logical resemblance to the problem being solved. Yet low-level parallel primitives require such identifiers and this makes the writing of real-world parallel applications tedious and error-prone. A second difficulty when using MP facilities to develop parallel programs is the fact that some traditional MP tools (i.e. MPI) does not allow problems to be expressed in terms of the logical communication patterns [34] of a number of well-identified and standard process topologies [36], including meshes, hypercubes, and trees. A third problem is that MP applications often call for the process management of MP facilities. This is a difficult task requiring special techniques and skills which, moreover, differ between MP facilities. This requires programmers to rely on their experience, quite often in an ad-hoc way.

Visual programming environments, i.e., systems with graphical user interfaces which support a user in developing parallel programs, suffer from a problem which is common to all large software systems; they are complex and time consuming to develop. This is because parallel/distributed programming is much more complex than

programming in a sequential paradigm. Many functions such as parallel execution, task mapping, interprocess communication, synchronization and reconfiguration are quite hard to program [5, 50] and even where the programmer can produce a complete application, the source code is subsequently difficult to maintain. Current supports are provided with visual programming environment, but the environment still lack of integrated programming and runtime support.

1.2 Contribution

This thesis describes a high-level programming methodology. It is based on the graph-orient programming (GOP) model, which was originally proposed as an abstract model for distributed programming [6, 9]. In applying GOP to parallel programming, we have observed that many parallel programs can be modeled as a group of tasks performing local operations and coordinating with one another over a logical graph. The logical graph depicts the architectural configuration and inter-task communication pattern of the application, and most of them are of a regular pattern, e.g. tree and mesh. Using a message-passing library, such as PVM and MPI, the programmer should use low-level primitives to manually translate the design-level graph model into its implementation. In the GOP model, such a graph metaphor is made explicit in the programming levels because GOP directly supports the graph construct. By directly using the logical graph construct, the task of a parallel program is configured

as a logical graph and implemented by using a set of high-level operations defined over the graph.

The results of this work are a set of tools, library and an environment for programming in parallel systems. The tools allow a unified view of the graphical/textual aspects of parallel programs and also a unified view of all the activity associated with them at the user interface: program editing, mapping, compilation and execution. More concretely, it is described as follows:

- *ClusterGOP*. This provides a high-level programming abstraction (GOP library) for building parallel applications on clusters. Graph-oriented primitives for communications, synchronization and configuration are perceived at the programming-level and their implementations hide the programmer from the underlying programming activities associated with accessing services through MPI. The programmer can thus concentrate on the logical design of an application, ignoring unnecessary low-level details. ClusterGOP can also support parallel software architecture. In ClusterGOP, the concept of software architecture is reified as an explicit graph object, which provides a locus for addressing architectural issues, separated from programming of the parallel tasks. Furthermore, the system architecture design can be simplified with the graph abstraction and predefined graph types. ClusterGOP system is portable to a variety of operating systems as its implementation is based almost exclusively on calls

to MPI, a portable message-passing standard, and the ClusterGOP library, a user-level library implemented on top of MPI.

- *VisualGOP*. This provides integrated graphical tools for creating, compiling, and executing programs. Because it is by nature graphical, VisualGOP is at a higher level of abstraction than GOP. Using the visual components provided by VisualGOP, a programmer creates a graph-oriented parallel/distributed program by visually constructing a graph, binding and specifying the local programs (LPs), and writing the code of the LPs for inter-process communication and synchronization. To run the constructed program, the programmer can compile the LPs and visually map them onto the physical processors in the network environment. In this way, the programmer does not need to code the textual specification of the graph construct and manage the mappings between logical processes to the physical processors.

In this thesis, we focus on GOP's abstraction in message-passing programming. Readers are referred to [8] for GOP's support for software architecture. Also, the system developed and described in this thesis is not a compiler for a GOP program. The system does, however, allow the programmer to specify the execution of programs in a visual way, so that programmers can write, compile and execute parallel programs within the visual tool. The program design and execution results will be shown on the visual tools. The programmer does not,

therefore, need to know the details of underlying system structure.

1.3 Organization of the Thesis

This thesis is divided into six chapters. Chapter 2, *Background and Related Work*, shows that many high-level programming models and visual programming environments exist, and that these benefit from the ability of the tool to quickly produce high-level programming support for them, making it much easier to develop software and maintain parallel applications. This thesis also identifies previous research that has addressed the problem of visual programming environments and previous work in developing message-passing parallel programs. Chapter 3, *The Graph-Oriented Model and ClusterGOP*, provides a detailed explanation of the GOP model and the way in which parallel programs are programmed with GOP. Chapter 4 describes *VisualGOP*, a visual programming tool developed for supporting the design and coding of graph-oriented parallel/distributed programs. Chapter 5, *Implementation of ClusterGOP*, describes the design and the implementation of ClusterGOP, a high-level parallel programming environment on clusters. Chapter 6, *Example Applications*, provides two detailed examples to show how ClusterGOP works with real-world applications and their performance results. Finally in *Conclusions and Future Work*, we conclude our project and outline the future work to enhance the construct.

Chapter 2

Background and Related Work

This chapter reviews the background and previous work germane to the system described in this thesis. It first describes the background of parallel computing, the most basic concept for designing and executing programs on parallel systems and follows that with a description of the high-level programming model for supporting the programming environment. Finally, we will examine some work that has been done on graph-based visual languages and discuss different approaches.

2.1 Parallel Computing

There have been considerable advances in processor technology in the past decade. Processor speeds have greatly increased and processors are now capable of executing multiple instructions in the same cycle. Over the same period, a variety of other issues have also become important. Concurrency provides multiplicity of datapaths, increased access to storage elements, scalable performance, and lower the cost in the

wide variety of parallel computing. Applications requiring high availability rely on parallel and distributed platforms for redundancy. Desktop machines, workstations and servers are now often equipped with several processors, and are connected to create a common platform for design applications. Large scale applications rely on larger configurations of parallel computers, often consisting of hundreds of processors. Database or web servers often use clusters of workstations that provide high disk bandwidth. Graphic and visualization applications use multiple rendering pipes and processing elements to compute and render realistic environments in real time. It is very important, from the point of view of cost, performance, and application requirements, to understand the principles, tools, and techniques for programming the wide variety of parallel platforms currently available.

2.1.1 Parallel Computing Applications

Parallel computing has made a large impact on areas from computational simulations for scientific and engineering applications to commercial applications in data mining and transaction processing. In this section, we classify the parallel applications in different categories and present some examples.

Applications in Engineering and Design

Parallel computing has traditionally been employed with great success in the design of airfoils, internal combustion engines, high-speed circuits, and structures. Recently,

its use in the design of microelectromechanical and nanoelectromechanical systems has attracted significant attention. Most of the applications in engineering and design solve some problems related to physical phenomena that creates great challenges for geometric modeling, mathematical modeling, and algorithm development, all of them in the context of parallel computers.

Other applications in the engineering and design focus on optimization of a variety of processors. Parallel computers have been used to solve a variety of discrete and continuous optimization problems. Algorithms such as Simplex, Interior Point Method and Genetic programming have been efficiently parallelized and are frequently used.

Scientific Applications

The past few years have seen a revolution in high performance scientific computing applications. The sequencing of the human genome by the International Human Genome Sequencing Consortium and Celera, Inc. has opened exciting new frontiers in bioinformatics. Analyzing biological sequences with a view to developing new drugs and cures for diseases and medical conditions requires innovative algorithms as well as large-scale computational power.

Advances in computational physics and chemistry have focused on understanding processes ranging in scale from quantum phenomena to macromolecular structures. These advances have resulted in design of new materials, understanding of chemical pathways, and more efficient processes. Applications in astrophysics have explored

the evolution of galaxies, thermonuclear processes, and the analysis of extremely large datasets from telescopes. Weather modeling, mineral prospecting, flood prediction, etc., rely heavily on parallel computers and have very significant impact on day-to-day life.

Commercial Applications

With the widespread use of the web and associated static and dynamic content, there is an increasing emphasis on cost-effective servers capable of providing scalable performance. Parallel platforms ranging from multiprocessors to Linux clusters are frequently used as web and database servers. For instance, on heavy volume days, large brokerage houses on Wall Street handle hundreds of thousands of simultaneous user sessions and millions of orders. Platforms such as IBMs SP supercomputers and Sun Ultra HPC servers power these business-critical sites. While not highly visible, some of the largest supercomputing networks are housed on Wall Street.

Applications in Computer Systems

As computer systems become more pervasive and computation spreads over the network, parallel processing issues become engrained in a variety of applications. In computer security, intrusion detection is an outstanding challenge. In the case of network intrusion detection, data is collected at distributed sites and must be analyzed rapidly for signaling intrusion. The infeasibility of collecting this data at a central

location for analysis requires effective parallel and distributed algorithms. In the area of cryptography, some of the most spectacular applications of Internet-based parallel computing have focused on factoring extremely large integers.

While parallel computing has traditionally confined itself to platforms with well behaved computer and network elements in which faults and errors do not play a significant role, there are valuable lessons that extend to computations in ad-hoc, mobile, or faulty environments.

2.1.2 Parallel Programming Models

When discussing parallel programming models, the parallel computing community usually considers two models: message-passing and shared memory. In this section we examine the features of these two models, and how these features affect the way programs are written in parallel environments.

Message-Passing Paradigm

The message-passing paradigm is one of the oldest and most widely used approaches for programming parallel computers. Its root can be traced back in the early days of parallel processing and its widespread adoption can be attributed to the fact that it imposes minimal requirements on the underlying hardware. Five attributes characterize the message-passing programming paradigm:

Multithreading: A message-passing program consists of multiple processes, each of

which has its own thread of control and may execute a different program code. Both control parallelism (MPMD, Multiple Program Multiple Data) and data parallelism (SPMD, Single Program Multiple Data) are supported.

Asynchronous Parallelism: The processes of a message-passing program execute asynchronously. Special operations, such as barrier and blocking communication, are used to synchronize processes.

Separate Address Spaces: The processes of a parallel program reside in different address spaces. Data variables in one process are not visible to other processes. Thus, a process cannot read from or write to the variable of another process variables. The processes interact by executing special message-passing operations.

Explicit Interactions: The programmer must resolve all the interaction issues, including data mapping, communication, synchronization, and aggregation. The workload allocation is usually done through the owner-computer rule; i.e., the process that owns a piece of data performs the computations associated with it.

Explicit Allocation: Both workload and data are explicitly allocated to the processes by the user. To reduce design and coding complexity, the user often realizes applications by writing SPMD programs using the single-code approach.

The following paragraphs describe the two public domain message-passing systems: the MPI and PVM. These two widely accepted systems are the product of many years of work by researchers and users in academia, industry, and government

laboratories. Both systems have incorporated many useful features from previous programming systems.

Message-Passing Interface (MPI): MPI is a standard specification for a library of message-passing functions. MPI was developed by the MPI Forum, a broadly based consortium of parallel computer vendors, library writers, and application specialists. MPI achieves portability by providing a public-domain, platform-independent standard of message-passing library. MPI specifies this library in a language-independent form, and provides Fortran C and C++ bindings. This specification does not contain any feature that is specific to any particular vendor, operating system, or hardware. For these reasons, MPI has gained wide acceptance in the parallel computing community. MPI has been implemented on IBM PCs on Windows, all main Unix workstations, Linux and all major parallel computers. This means that a parallel program written in standard C or Fortran, using MPI for message-passing, could run in cross platform.

Parallel Virtual Machine (PVM): PVM is a self-contained, public-domain software system that was originally designed to enable a network of heterogeneous Unix computers to be used as a large-scale, message-passing parallel computer. As its popularity grows, it has been ported to many parallel systems. The programming languages supported include C, Fortran, and Java. With PVM, a user can construct a virtual machine, a set of fully connected nodes. Each node can be any Unix computer, such

as any sequential, vector, or parallel computer. The user can then dynamically create and manage a number of processes to run on this virtual machine. PVM provides library routines to support interprocess communication and other functions.

Comparison between MPI and PVM The main difference between PVM and MPI is that PVM is a self-contained system while MPI (or more specifically, MPI-1) is not. MPI relies on the underlying platform to provide process management and I/O functions. These functions are included in PVM. On the other hand, MPI has more powerful support for message-passing. MPI and PVM are evolving towards each other. For instance, MPI-2 added process management functions, while PVM now has more collective communication functions. It will be beneficial to the parallel processing community if PVM and MPI eventually merge into a single, standard library. Moreover, a nice feature of the MPI design is that MPI provides powerful functionality based on four orthogonal concepts. One can learn these concepts individually, knowing their combinations follow well-defined semantic patterns. MPI provides more than 200 functions, much more than PVM. Normally, MPI is easier to learn and use than PVM.

Shared Memory

In shared memory architectures, communication is implicitly specified since some (or all) of the memory is accessible to all the processors. Consequently, programming

paradigms for shared memory machines focus on constructs for expressing concurrency and synchronization along with techniques for minimizing associated overheads.

However, parallel programming based on the shared memory model has not progressed as much as message-passing parallel programming. An indicator is the lack of a widely accepted standard such as MPI or PVM for message-passing. The current situation is that many shared memory programs are written in a platform-specific language for multiprocessors. Such programs are not portable even among multiprocessors. Fortunately, there are also some platform-independent shared memory programming models, such as Pthreads and OpenMP.

2.1.3 Cluster Computing

In the past, supercomputers were the only machines powerful enough to solve hard problems in fields such as medical science, biology, space exploitation, natural science, robotics, financial modeling, and the design of human-machine interfaces. The progress of work will depend on the magnitude of computational performance the systems are able to engage. Unfortunately, the market value of such extreme performance systems appears to be, in the short term at least, well below that required to justify industry investment in the development of these specialty-class supercomputer architectures.

Fortunately, this conflict between the requirements for high performance and the availability of resources needed to provide it is being addressed through an innovative

synergy of some old ideas from the parallel computing community and some new low-cost technologies from the consumer digital electronics industry. The legacy concepts are the physical clustering of general purpose hardware resources and the abstract message-passing model of distributed computing. The low cost computing capability is derived from the mass market Commodity Off The Shelf (COTS) PC and local networking industries. Together, these basic capabilities and founding principles, which originally serving for very different purposes, have emerged as the new domain of high performance: commodity cluster computing.

A cluster is a collection of complete computers (nodes) that are physically interconnected using a high-performance network or a Local Area Network (LAN). Typically, each computer node is an Symmetrical Multiprocessor (SMP) server, or a workstation, or a personal computer. All cluster nodes must be able to work together collectively as a single, integrated computing resource in addition to filling the conventional role in which each node is used interactively by individual users.

Five architectural concepts are merged into a cluster as an interconnected set of whole computers (nodes) that work collectively as a single system to provide uninterrupted (availability) and efficient (performance) services. A conceptual architecture of a cluster is shown in Figure 2.1.

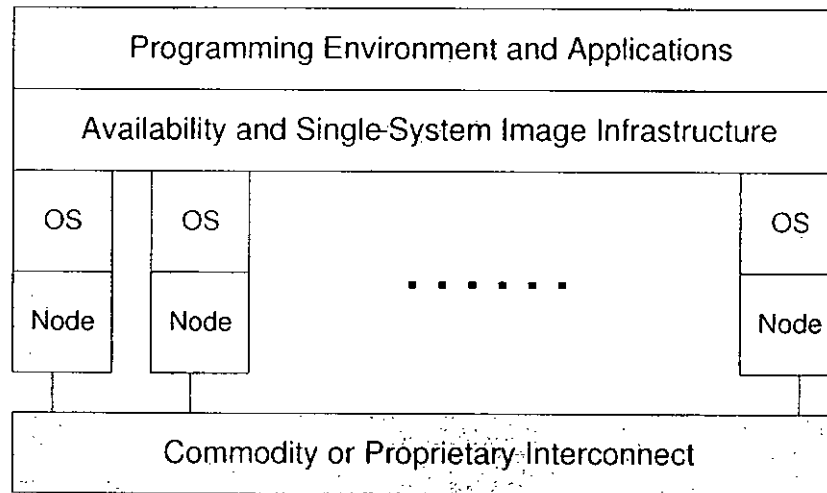


Figure 2.1: Typical architecture of a cluster of multiple computers

Benefits of Clusters

Clusters allow recycle use of computer: hardware and software technologies that were developed for broad application to mainstream commercial and consumer markets can also operate in the clustering environment. Both networks of workstations and Beowulf-class PC clusters were possible because they required no expensive or long-term development projects prior to their initial end use. Such early systems were far from perfect but they were usable. The cluster systems have price-performance advantages over contemporary supercomputers. More than that, the rapid rate of improvement in PC microprocessor performance and advances in local area networks has led to systems capable of tens or even hundreds of Gigaflops performance while retaining exceptional price-performance benefits.

Commodity clusters permit a flexibility of configuration not ordinarily encountered through conventional Massively Parallel Processor (MPP) systems. Number of nodes, memory capacity per node, number of processors per node, and interconnect topology are all parameters of system structure that, due to custom configurability, may be specified in fine detail on a per system basis without incurring additional cost. Further, the system structure may easily be modified or augmented over time as need and opportunity dictates without the loss of prior investment. This expanded control over the system structure not only benefits the end user but the system vendor as well, yielding a wide array of system capabilities and cost tradeoffs to better meet customer demands. Commodity clusters also permit rapid response to technology improvements. As new devices including processors, memory, disks, and networks become available, they are most likely to be integrated into desktop or server nodes most quickly allowing clusters to be the first class of parallel systems to benefit from such advances. The same is true of benefits incurred through constantly improving price-performance trends in delivered technology. Commodity clusters are best able to track technology improvements and respond most rapidly to new component offerings.

2.2 High-Level Programming Support

Writing parallel/distributed programs overwhelms many programmers due to the difficulty of explicitly expressing communication and synchronization among the computations. The ability to develop parallel programs quickly and easily is becoming increasingly important to many programmers. A high-level programming model facilitates the building of large-scale applications, and bridges the semantic gap between the application and the parallel machines. It also facilitates code reuse, reduces code complexity, and abstracts away low-level details necessary to achieve performance on a particular architecture. The current models for message-passing are too low-level to achieve this objective. Although we cannot expect parallel programming to become as easy as sequential programming, we can avoid unnecessary difficulties by using appropriate tools. Many efforts have been made to make programming easier by providing high-level abstractions, programming tools, etc. In the next section, we will describe the models which use high-level approaches.

2.2.1 High-Level Programming Tools

Ensemble [16, 17, 18] supports the design and implementation of message-passing applications (applied to MPI and PVM), particularly MPMD and those demanding irregular or partially regular process topologies. Also, the applications are built by composing modular message-passing components. Ensemble divided the software

architecture into two layers: the Abstract Design and Implementation (AD&I), which is the responsibility of the programmer, and the Architecture Specific Implementation (ASI), e.g. MPI implementation, which is generated from the AD&I and transparent to the developer. AD&I consists of three well-separated implementation parts: virtual components, symbolic topologies, and resource allocation. Virtual components are the implementation abstractions of a MP program. For example, they provide abstract names and abstract roots for collective calls and abstract point-to-point interaction. They also use "ports" to replace the MPI argument types (context, rank and message tag). Symbolic topology is an abstraction of a process topology, which specifies the number of processes required from each component, each process's interface and its interaction with other processes. Resource allocation is the mapping of processes, as well as the location of source, executable, input and output files in the underlying environment.

We compare Ensemble and GOP in four aspects. Firstly, the programming model of Ensemble and GOP are different. Ensemble is mainly for abstract programming design of the application. Instead of compiling the programs directly, Ensemble uses a tool to generate the abstract parallel programs into pure MPI code, and then compiles the code into modular MPI components. In GOP, the programmer can use the high-level GOP API to develop parallel applications. Then the parallel programs and the GOP library will be compiled together to form executable programs and run. GOP

also includes a runtime system to help the graph update and synchronization, so that the graph topology for communication can be changed during runtime. The program is static, so the topology for communication cannot be changed easily. Secondly, GOP and Ensemble have similar features in the application programming. Abstraction is used for referring the node names, node groups and communicating edges. Thirdly, GOP has a flexible mapping strategy which provides automatic and manual mapping, but in Ensemble mapping can be done only manually. The last difference is that the GOP syntax is independent of MPI, which provides a high-level abstraction library for programming parallel applications. The Programmer can design the GOP application without need to know any low-level syntax of MPI. However, the programming structure of Ensemble is a mixture of MPI and its syntax. Therefore, the programmer needs to learn the new architecture design of Ensemble and also the usage of MPI communication routines.

Some systems integrate message passing with other parallel paradigms, such as the data parallel approach, to enhance the programming support and take advantages of different paradigms. The Nanothreads Programming Model (NPM) [26] is a programming model for shared memory multiprocessors. The NPM can integrate with MPI, used on distributed memory systems. The runtime system is based on a multilevel design that supports both the models (NPM and MPI) individually but offers the capability to combine their advantages. Existing MPI codes can be executed

without any changes and codes for shared memory machines can be used directly, while the concurrent use of both models is easy. The major feature of the NPM runtime system is portability, as it is based exclusively on calls to MPI and Nthlib, a user-level threads library that has been ported to several operating systems. The runtime system supports the hybrid-programming model (MPI + OpenMP) [29]. Moreover, it extends the API and the multiprogramming functionality of the NPM on clusters of multiprocessors and can support an extension of the OpenMP standard on distributed memory multiprocessors.

There are similarities and differences between NPM and GOP. The two models are similar in that they both have their own API to support high-level programming design. This overcomes the inadequate support of the low-level library routines that are included in the MPI, providing a more efficient and easy way to design and manage the application. The two models differ in that although they can both separate the application into a number of smaller parts, they have different programming structures. NPM decompose the application into fine-grain tasks and executed in a dynamic multi-programmed environment. The parallelizing compiler analyzes the source program in order to produce an intermediate representation, called the Hierarchical Task Graph. The graph is used for the mapping of user tasks to the physical processors on runtime. The model allows one or several user-level ready queues to contain the ready tasks that are waiting for execution. When a processor finishes its

current task, it picks up the next task from the ready queue. Processors continuously pick up tasks from the ready queue until the program terminates. In GOP, the graph is used for several purposes, not only for mapping, and nodes in the user-defined graph are representing different processes. Programmers can write graph-oriented programs to be bound to the nodes and then map the nodes into the physical processors. Since the mapping is one-to-one, programmers can map manually or let the system do it automatically. Before the application executes, they can review their design and the mapping information.

Another example is Global Arrays (GA) [41] which allows programmers to easily express data parallelism in a single, global address space. GA provides an efficient and portable shared-memory programming interface for parallel computers. The use of GA allows the programmer to program as if all the processors have access to the same data in shared memory.

There are some high-level MPMD languages (e.g. Mentat [25], C++ [12] and Fortran-M [22]) and runtime systems (e.g. Nexus [23]), which support combination of dynamic task creation, load balancing, global name space, concurrency, and heterogeneity. Due to the need for crossing program domains, for asynchronously detecting incoming communication, and for potentially spawning new threads, the communication overheads in these systems are often prohibitively high for a multi-computer system. In these systems, programming platforms based on an SPMD model (e.g.

Split-C [19] and CRL [32]) have a significant performance advantage over MPMD-based ones and are preferred for parallel application development. However, there exists MPMD systems (e.g. MPRC [14, 13]) using RPC as the primary communication abstraction which produce good results compared with the SPMD model.

2.2.2 XML Support for High-level Programming Environment

XML (eXtended Markup Language) [3] is a popular and a widely accepted standard description language for data interchange. The central idea of XML is to allow users to create any documents of their preferred structures and share with other people. This opens a way for different types of document structures to be created to facilitate communications for various professional domains.

There are some research for graph description languages, such as GraphXML [27] and GXL [28], based on the rich features of XML. Both GraphXML and GXL provide interchange formats for graph drawing and information visualization applications, with provisions for extension. We propose an extension to GraphXML to suit the systems like VisualGOP for graph-based parallel and distributed programming, and call the GOP-XML. GOP-XML includes the specification for processor configuration, LP definition, various edge and node attributes, etc. Each graph in GOP-XML can be represented as an XML document.

2.3 Visual Programming Environment

Support for visual programming of parallel and distributed systems is provided by graph-based programming environments, in which a program is defined as a directed graph where nodes denote computations and links denote communication and synchronization between nodes [4]. Directed-graph program representations are used to specify multiple threads of control, to display and expose parallel structure, and to express communication and synchronization which are separated from the specification of sequential computations in individual LPs.

2.3.1 Graph-Based Visual Languages

In the past decade, a number of visual programming environments have been developed [49, 33, 39, 40, 44, 48, 50, 51]. HeNCE [1, 51] is an X-windows based software environment for developing parallel programs that run on a network of computers. To develop a HeNCE application, a programmer first expresses the sequential computations in a standard language and then specifies how they are to be decomposed into a parallel program. Based on a parallel programming paradigm where an application program is described on a graph, HeNCE provides the programmer with a high-level abstraction for specifying parallelism. CODE [33, 38] is a visual environment similar to HeNCE. It is a graphical, re-targetable parallel programming system. The programmer writes parallel programs by drawing a graph which represents the

relationships between the various units of computation. The structure of the graph captures major elements of the parallel structure of the program. The graph serves as a template which is used as a framework for creating dynamic structures at runtime. Both HeNCE and CODE facilitate a compositional approach to programming. Sequential units of computation are composed to form a parallel program where dependencies are specified by means of arcs of the directed graph [5]. VDCE [48] is based on a dataflow programming paradigm. The software architecture consists of three parts: Application Editor, Application Scheduler, and VDCE Runtime System. VDCE provides an efficient mechanism to execute large-scale applications on distributed and diverse platforms. Importantly, in HeNCE, CODE and VDCE, the nodes represent sequential computations in which communications with other nodes occur only at the beginning and end of the computation [4]. The dependencies are mainly based on data-flow, so the computations are forced to be split into separate processes when communications occur. Overcoming this problem, Phred [2] makes an attempt to combine dataflow and control flow at the same level of abstraction as HeNCE and CODE. Neither of these formalisms provides sufficient information on the spatial distribution of processes. By explicitly specifying processes with both dataflow and control flow, PCG (Process Communication Graph) [46] used in the Visper distributed programming environment [47], which adapts the Space-Time Diagram to visualize the design phase of message-passing programs. Its levels of abstraction range

from groups of processes, processes communication and synchronization, to sequential program blocks. The PCG environment, however, lacks the support for graph-based operations and validation capability. In terms of the programming model and functionality, systems closer to VisualGOP include VERDI [44], the Software Architect's Assistant [40], and GRADE [33]. VERDI is a visual environment for Raddle [20]. It provides a visual language used for describing the system control flow. VERDI has a graphical editor that allows a designer to represent the design graphically using icons. It can also execute a design using data that are either supplied by the programmer or generated internally. The designer can examine the functionality of the design during execution through VERDI's animation feature or through the data produced, or even through a combination of both features. The Software Architect's Assistant is a visual programming environment for the design and development of Regis [37]. Facilities provided include the integrated graphical and textual views, a flexible mechanism for recording design information and the automatic generation of program code and formatted reports from design diagrams. The focus of the Assistant is on program design and construction. There is no management of network resources such as processor mapping. GRADE is a high-level, integrated programming environment for PVM based program development. It provides a graphical user interface through which the programmer can access tools to construct, execute, debug, monitor, and visualize PVM parallel programs. GRADE also provides high-level graphical programming

abstraction mechanisms to construct parallel applications. Same as the Assistant, GRADE does not provide support for mapping of parallel processes to processors.

2.4 Summary

This chapter describes three threads of research towards developing parallel programming environments. Some previous works has been done on supporting parallel programming and graph visualization. The next chapter describes the graph-oriented programming approach and the high-level programming support in clusters. Chapter 4 provides a detailed description of the high-level supporting tool, VisualGOP, for programmer to program in the parallel environment.

Chapter 3

Graph-Oriented Model and ClusterGOP

In this chapter, we first introduce the GOP model for high-level programming of parallel applications. We then describe the ClusterGOP system, including the ClusterGOP API and discuss the enhancement to MPI provided by GOP. Finally, we use two examples to illustrate how ClusterGOP enhances the high-level programming support in parallel environment.

3.1 The Graph-Oriented Programming Model

In the GOP model, a parallel program is defined as a collection of LPs that may execute on several processors. Parallelism is expressed through explicit creation of LPs and communication between LPs is solely via message-passing. GOP allows programmers to write parallel programs based on user-specified graphs, which can

serve the purpose of naming, grouping and configuring LPs. It can also be used as the underlying structure for implementing uniform message-passing and LP coordination mechanisms.

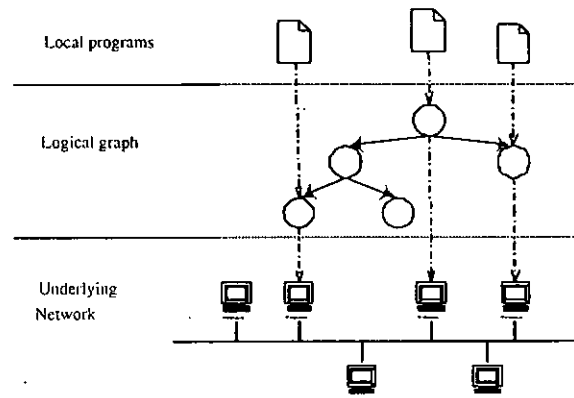


Figure 3.1: The GOP conceptual model

The key elements of GOP are a logical graph construct to be associated with the LPs of a parallel program and their relationships, and a collection of functions defined in terms of the graph and invoked by messages traversing the graph. As shown in Figure 3.1, the GOP model consists of the following:

- A *logical graph* whose nodes are associated with LPs, and whose edges define the relationships between the LPs.
- An *LP-to-Node mapping*, which allows the programmer to bind LPs to specific nodes.
- An optional *nodes-to-processors mapping*, which allows the programmer to explicitly specify the mapping of the logical graph to the underlying network of

processors. When the mapping specification is omitted, a default mapping will be performed.

- A library of language-level graph-oriented programming primitives.

GOP programs are conceptually sequential but are augmented with primitives for binding LPs to nodes in a graph, with the implementation of graph-oriented inter-node communications completely hidden from the programmer. The programmer defines variables of the graph construct in a program and then creates an instance of the construct. Once the local context for the graph instance is set up, communication and coordination of LP's can be implemented by invoking operations defined on the specified graph. The sequential code of LPs can be written using any programming language such as C, C++ and Java.

A graph-oriented parallel program is defined as a logical graph, $G(N, E)$, where N is a finite set of nodes and E is a finite set of edges. Each edge of the graph links a pair of nodes in N . A graph is directed if each edge is unidirectional. A graph is labeled if every edge is associated with a label [9]. A graph is associated with a parallel program, which consists of a collection of LPs bound to the nodes with the messages that pass along the edges of the graph. Edges denote the interaction relationship between LPs. The graph can represent a logical structure that is independent of the real structure of a parallel system. Such a parallel system can be used to reflect the properties of the underlying system. For example, the label on each edge may denote

the cost or delay in sending a message from one site to another site within the system.

A GOP program consists of a collection of LPs, which are built using the graph construct, and a main program. A graph construct consists of a directed conceptual graph, an LP-to-Node mapping, and an optional Node-to-Processor mapping. The programmer can create an instance of a graph construct using the following three steps:

- Step 1: **Graph** template declaration and instantiation

A **Graph** is a template for a logical graph describing the logical relationships between LPs. It instantiates a graph instance and associates a name with the instance. The structure of a **Graph** is a general type of logical graph, which is described as a list of nodes connected with edges. It is defined as follows in Backus-Naur Form.

$$\langle \textit{Graph-template} \rangle ::= \mathbf{Graph} \ \textit{Graph-name} \ '=' \ \{ \{ \langle \textit{set-of-nodes} \rangle \}, \{ \langle \textit{list-of-edges} \rangle \} \}$$

$$\langle \textit{set-of-nodes} \rangle ::= \langle \textit{range-of-nodes} \rangle \mid \langle \textit{node-list} \rangle$$

$$\langle \textit{range-of-nodes} \rangle ::= \langle \textit{node_no} \rangle .. \langle \textit{node_no} \rangle$$

$$\langle \textit{node-list} \rangle ::= \langle \textit{node-list} \rangle, \langle \textit{node_no} \rangle \mid \langle \textit{node_no} \rangle$$

$$\langle \textit{list-of-edges} \rangle ::= \langle \textit{list-of-edges} \rangle, \{ \textit{node_no}, \textit{node_no} \} \mid e$$

A **Graph** is the type identifier denoting the definition of a graph construct. The *Graph-name* is an identifier of a graph construct. The *node_no* is an integer identifier of a node.

- Step 2: Mapping in GOP

Map LPs to the conceptual nodes of a graph and the nodes to the underlying processors. The LP-to-Node mapping is defined as a set of $(node_no, LP_no)$ pairs.

$$\langle LN_mapping \rangle ::= \mathbf{LNMAP} \ LN_map_name \ '=' \ \{ \langle node_lp_pair \rangle \}$$

$$\langle node_lp_pair \rangle ::= \langle node_lp_pair \rangle, \{ \langle node_no \rangle, \langle LP_no \rangle \} \mid e$$

LNMAP is the type identifier of an LP-to-Node mapping. *LN-map-name* is the name of a mapping instance. The LP named in *LP_no* is mapped to the node identified by *node_no*.

The Node-to-Processor mapping is optional in the graph construct, which is specified by a set of $(node_no, processor_no)$ pairs, in a similar form to the LP-to-Node mapping.

$$\langle NP_mapping \rangle ::= \mathbf{NPMAP} \ NP_map_name \ '=' \ \{ \langle node_processor_pair \rangle \}$$

$$\langle node_processor_pair \rangle ::= \langle node_processor_pair \rangle, \{ \langle node_no \rangle, \langle processor_no \rangle \} \mid e$$

NPMAP is the type identifier of a Node-to-Processor mapping. *NP-map-name* is the name of a mapping instance. The node named in *node_no* is mapped to the processor identified by *processor_no*.

- Step 3: Graph construct binding

Given the declaration of a graph construct and its mappings, a graph instance can be

created by binding the mappings to the graph. The routine `CreateGraph` is used for this purpose.

```
CreateGraph (Graph-name, LN-map-name, NP-map-name, Graph-instance-name);
```

Graph-instance-name is the identifier of a file that specifies all of the information about the newly created instance. This information is useful in establishing the operating context for the LPs. *LN-map-name* is the name of an LP-to-Node mapping and *NP-map-name* is the name of a Node-to-Processor mapping.

Programming based on a graph-oriented model includes creating the graph construct and writing program codes for the LPs using the *graph primitives*. GOP allows the programmer to exploit the semantics of the graph construct to deal with various aspects of parallel programming. The graph primitives define operations on a user-specified graph including communication and synchronization. These operations can be used to pass messages from one node to other nodes in the graph without knowing the low-level details such as absolute naming, addressing and routing. In this way, the programmer is saved from the burden of writing dedicated program codes for implementing task mapping and message-passing, and can concentrate on designing the structure and the logic of the parallel program instead.

3.2 ClusterGOP System Structure

It is important to note that the GOP model is independent of any particular language and platform. It can be implemented as library routines incorporated in familiar sequential languages and integrated with different programming platforms. In this thesis, however we present an implementation of the GOP framework on MPI, called ClusterGOP.

The ClusterGOP software environment is illustrated in Figure 3.2. The top layer is a visual programming environment. It supports the design and construction of parallel program. It has a highly visual and interactive user interface, and provides a framework in which the design and coding of ClusterGOP program, and the associated information can be viewed and modified easily and quickly. It also facilitates the compilation, mapping, and execution of programs (see Chapter 4).

A set of ClusterGOP API is provided for the programmer to use in parallel programming, so that the programmer can build application based on the ClusterGOP model, ignoring the details of low-level operations and concentrating on the logic of the parallel program. The ClusterGOP library provides a collection of routines implementing the ClusterGOP API. The goal in the ClusterGOP library implementation is to minimize the package overhead by introducing a minimum number of services with a very simple functionality.

The runtime system is responsible for compiling the application, maintaining

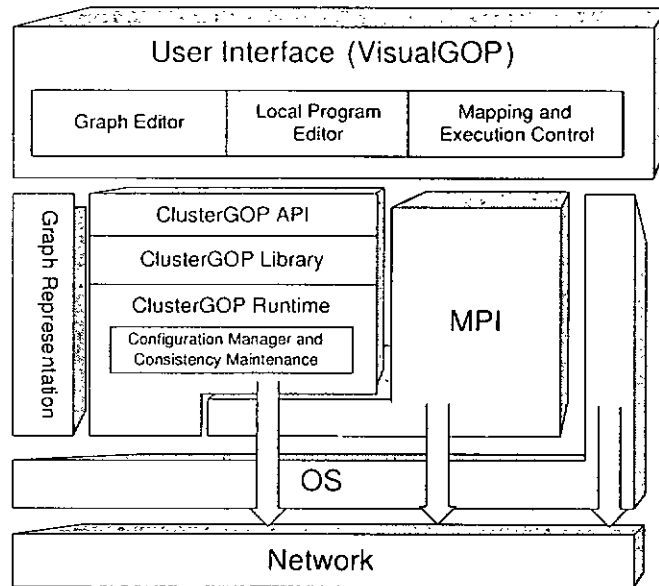


Figure 3.2: The ClusterGOP Framework

structure, and executing the application. The target machine contains two runtimes. The first runtime is the ClusterGOP runtime, a background process that provides graph deployment, update, query and synchronization. When deploying and updating the graph, it will block other machines in order to further update the graph and synchronize the graph update on all machines. The second runtime is the MPI runtime, which provides a complete set of parallel programming library for the ClusterGOP implementation. ClusterGOP uses MPI as the low-level parallel programming facility so that processes can communicate efficiently.

3.3 The ClusterGOP Library

The implementation of ClusterGOP applications is through wrapping functions (ClusterGOP library) to native MPI software. In general, ClusterGOP API adheres closely to MPI standards. However, the ClusterGOP library simplifies the API by providing operations that automatically perform some MPI routines. It also allows argument list to be simplified relative to the MPI programs.

For message passing, ClusterGOP provides a set of routines to enable graph-oriented point-to-point and collective communications (in both blocking and non-blocking modes). In this layout, the ClusterGOP system follows the MPI standard, but it is simpler and the implementation is specifically designed for the graph-oriented framework. For example, we use node ID instead of process ID to represent different processes, so the LP bound to a node can be replaced without affecting other LPs. ClusterGOP also hides the complexity of low-level addressing, communication, as well as initializing processes from the programmer.

ClusterGOP provides programmers with three types of communication and synchronization primitives:

- Point-to-Point Communication
- Collective Communication
- Synchronization

Point-to-point Communication consists of a process that sends a message and another process which receives the message - a send/receive pair. These two operations are very important as well as being the most frequently used operations. To implement optimal parallel applications, it is essential to have a model that accurately reflects the characteristics of these basic operations. *Collective Communication* refers to message passing routines involving a group (collection) of processes. Sometimes, one wishes to gather data from one or more processes and share this data among all participating processes. At other times, one may wish to distribute data from one or more processes to a specific group of processes. In ClusterGOP API, there is another type of collection communication primitive which provides the operations using the parent and child relationships. It is used for finding the parent or child nodes, and then broadcast the data to all the corresponding nodes. Finally, *Synchronization operations* are provided to support the synchronization of processes.

The following is the list of ClusterGOP communication and synchronization primitives:

Point-to-point Communication primitives:

```

/* sending unicast message */
MsgHandle Usend(Graph g, Node n, Msg msg, CommMode m)

/* receiving unicast message */
MsgHandle Urecv(Graph g, Node n, Msg msg)

/* send message to parent nodes */
MsgHandle SendToParent(Graph g, Msg msg, CommMode m)

/* receive message from parent nodes */

```

```

MsgHandle RecvFromParent(Graph g, Msg msg)

/* send message to children nodes */
MsgHandle SendToChildren(Graph g, Msg msg, CommMode m)

/* receive message from children nodes */
MsgHandle RecvFromChildren(Graph g, Msg msg)

```

Collective Communication primitives:

```

/* sending multicast message */
MsgHandle Msend(Graph g, NodeGroup ng, Msg msg, CommMode m);

/* receiving multicast message */
MsgHandle Mrecv(Graph g, NodeGroup ng, Msg msg);

/* s collect data from all nodes in the NodeGroup */
MsgHandle Gather(Graph g, NodeGroup ng, Msg msg, Node s);

/* s distribute data to all nodes in the NodeGroup */
MsgHandle Scatter(Graph g, NodeGroup ng, Msg msg, Node s);

/* data collection in all nodes in the NodeGroup */
MsgHandle Allgather(Graph g, NodeGroup ng, Msg msg);

/* data distribution in all nodes in the NodeGroup*/
MsgHandle Alltoall(Graph g, NodeGroup ng, Msg msg);

/* reduce data to s from all nodes in the NodeGroup */
MsgHandle Reduce(Graph g, NodeGroup ng, Msg msg, Node s);

/* data reduction in all nodes in the NodeGroup */
MsgHandle Allreduce(Graph g, NodeGroup ng, Msg msg);

```

Synchronization:

```

/* Synchronize all nodes in the graph */
void barrier(Graph g);

/* Synchronize this node with all (other) nodes in the NodeGroup. */
void barrier(Graph g, NodeGroup ng);

/* Check if the msg(s) arrived. */
Boolean isArrived(Graph, MsgHandle handle);

```


The signatures of the above methods all follow one pattern, in which the first argument represents the specified logical graph, used for defining the scope of process communication. The communication target can be either a single node (`Node`) or a group of nodes (`NodeGroup`). Unlike MPI, ClusterGOP hides the message contents, type and tag by embedding them inside the `Msg` datatype. ClusterGOP API thus uses fewer arguments than MPI. This reduces the complexity of using the API routines and makes program maintenance easier.

ClusterGOP also provides a set of operations to query the information about nodes and edges in the graph. The corresponding ClusterGOP API is listed below:

Query primitives:

```
/* get the edge from start node and end node */
Edge GetEdge(Graph, Node start, Node end);

/* get the end node from edge */
Node GetNode(Graph, Edge edge);
```

This query information can be generated during the running of the application. Programmers can use the query information in communication. For example, when programmers want to find the neighbor node connected with the current node, they can use the routine `GetNode` to retrieve the node name of a connected edge. Programming in this way helps programmers dynamically assign the node names in the communication procedures, without specifying static node names in the LP code. Therefore it helps programmers design the LP structure freely, and produces a more readable code for software maintenance.

In ClusterGOP, we use a NodeGroup to represent a group of processes, providing the same functionality as the MPI communicator. NodeGroup helps the programmer to write code for group communications. A NodeGroup consists of the member processes. Each process in the group can invoke group communication operations such as collective communications (gather, scatter, etc). NodeGroup hides the programming details that are used for constructing the NodeGroup in the MPMD programs so that the programmer can concentrate on programming the nodes' tasks. As a result, the program is easier to understand.

As shown below, NodeGroup has simple APIs that are easy to use. In forming a group, the programmer first forms a task group, assigns a name to the group, and then adds nodes to or removes nodes from the NodeGroup.

```
/* create the NodeGroup from a list of nodes */
NodeGroup InitNodeGroup (Graph gname, char* group_name, Node[] nodelist);

/* add one node to the NodeGroup */
int AddNode (NodeGroup *ng, Node s);

/* remove one node from the NodeGroup */
int RemoveNode (NodeGroup *ng, Node s);

/* clear all nodes in the NodeGroup */
void ClearNodeGroup (NodeGroup *ng);
```

3.4 Programming in the ClusterGOP Environment

This section shows the advantage of using the ClusterGOP environment to develop parallel applications. We demonstrate this with two examples, comparing the differences between programming in ClusterGOP and MPI. ClusterGOP can support both SPMD and MPMD program structures. In our examples, we first use an SPMD program, Finite Difference Method (FDM), to compare more clearly the differences between ClusterGOP and MPI programming. Then, we use another programming example, parallel matrix multiplication, to show that ClusterGOP supports high-level MPMD programming.

3.4.1 SPMD Programming Example

In our first example, we use a popular scientific program to illustrate how ClusterGOP supports higher level message-passing parallel programming. The FDM is an approach to obtaining an approximate solution to a partial differential equation governing the behavior of a physical system. The method imposes a regular grid on the physical domain. It then approximates the derivative of an unknown value u at a grid point (x, y) using the values of adjacent grid points. Consider a specific partial differential equation- Laplace equation, the approximation u_{ij} to the exact solution $u(x_i, y_j)$ on the grid satisfies the equation

$$u_{ij} = \frac{1}{4}(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}), \quad i, j = 0, \dots, N-1. \quad (3.4.1)$$

Equation (3.4.1) shows that the value of u at any point is affected by four adjacent elements. Given the initial value, the value of u at any point can be approximated by iteration

$$u_{ij}^{(k)} = \frac{1}{4}(u_{i+1,j}^{(k-1)} + u_{i-1,j}^{(k-1)} + u_{i,j+1}^{(k-1)} + u_{i,j-1}^{(k-1)})$$

until the predefined accuracy is reached.

When solving the problem on p processors, the grid will be partitioned into p sections. The grid can be decomposed in different ways. Here, a two-dimensional partition is used that generates a coarser grid. Figure 3.3 shows the grid partitions for 4 processors. Figure 3.3(a) is the partition on 4 processors. Figure 3.3(b) is the program graph of the FDM derived from this partition. The program graph has the coarse-grid topology correspondent to the physical topology of the grid. The entry and exit nodes are omitted in the program graph because they are dispensable to expressing the problem.

When programming in ClusterGOP, LPs can be implemented using common programming languages such as C, C++ or Java. In this example, we use the C language to write the application. The LP structure is similar to that of a MPI program, but ClusterGOP provides a simple statement to hide the details from the programmer. In each ClusterGOP program, the code starts with the routine `Init` and ends with the routine `Finalize`:

```
Init(argc, argv); ... .. Finalize();
```

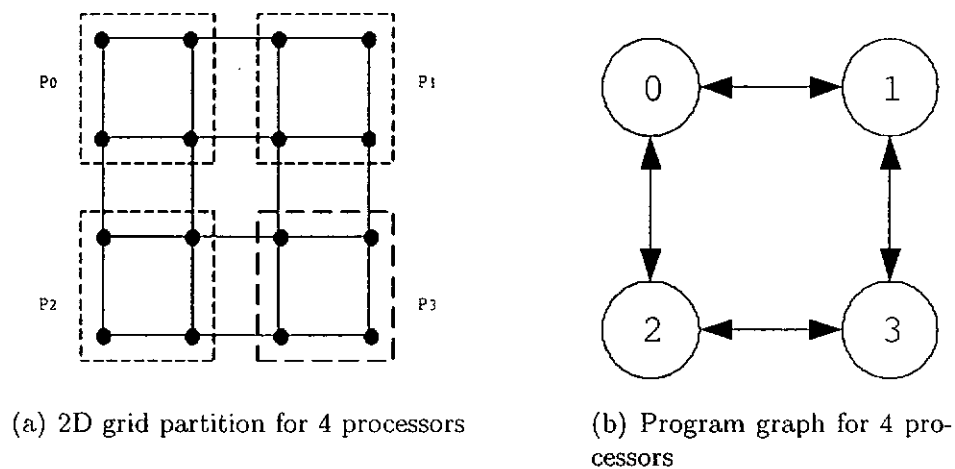


Figure 3.3: Grid partitions for 4 processors

Between routines `Init` and `Finalize`, the programmer writes code for the application. ClusterGOP communication primitives are used for processes to communicate with each other in the parallel environment. The routine `Usend` is used for communication in the example. It defines a unicast message-passing primitive, delivering a message from the current node to another node in synchronous or asynchronous sending mode. In calling the sending and receiving routines, ClusterGOP runtime resolves the node's process ID automatically. Here is the example of the message-passing routine `Update_Boundary_Condition` in the FDM:

```
Update_Boundary_Condition( double **solution_array, int mrow, int mcol, int k) {
    int i;
    Msg msg1, msg2, msg3, msg4;
    if ( k % 2 == 0 ) { /* even numbered processes */
        /* -- The message template is generated by VisualGOP -- */
        msg1 = createMsg(&(solution_array[mrow][1]), mcol, DOUBLE, 0);
        msg2 = createMsg(&(solution_array [0][1]), mcol, DOUBLE, 0);
        msg3 = createMsg(&(solution_array [1][1]), mcol, DOUBLE, 1);
        msg4 = createMsg(&(solution_array [mrow+1][1]), mcol, DOUBLE, 1);
```

```

/* ----- */
Usend(ggraph, GetNode(ggraph, "right"), msg1, ASYN);
Urecv(ggraph, GetNode(ggraph, "left"), msg2);
Usend(ggraph, GetNode(ggraph, "left"), msg3, ASYN);
Urecv(ggraph, GetNode(ggraph, "right"), msg4);
}
else {          /* odd numbered processes */
    Urecv(ggraph, GetNode(ggraph, "left"), msg2);
    Usend(ggraph, GetNode(ggraph, "right"), msg1, ASYN);
    Urecv(ggraph, GetNode(ggraph, "right"), msg4);
    Usend(ggraph, GetNode(ggraph, "left"), msg3, ASYN);
}
...
/* -- The message template is generated by VisualGOP -- */
msg1 = createMsg(&sbuf_up,  mrow, DOUBLE, 2);
msg2 = createMsg(&rbuf_down, mrow, DOUBLE, 2);
msg3 = createMsg(&sbuf_down, mrow, DOUBLE, 3);
msg4 = createMsg(&rbuf_up,  mrow, DOUBLE, 3);
/* ----- */
if ( k % 2 == 0 )      /* even numbered processes */
    Usend(ggraph, GetNode(ggraph, "up"),  msg1, ASYN);
    Urecv(ggraph, GetNode(ggraph, "down"), msg2);
    Usend(ggraph, GetNode(ggraph, "down"), msg3, ASYN);
    Urecv(ggraph, GetNode(ggraph, "up"),  msg4);
}
else          /* odd numbered processes */
{
    Urecv(ggraph, GetNode(ggraph, "down"), msg2);
    Usend(ggraph, GetNode(ggraph, "up"),  msg1, ASYN);
    Urecv(ggraph, GetNode(ggraph, "up"),  msg4);
    Usend(ggraph, GetNode(ggraph, "down"), msg3, ASYN);
}
...
}

```

In contrast, in MPI, each process must provide the static process ID for communication during the program compilation. As a result, the programmer is required to write the extra routine `neighbors` to calculate the processor IDs of neighbors for each node:

```
neighbors(int pid, int *left, int *right, int *up, int *down, int
```

```

total_p_num) {
    int q, r, c, proc_col, i=0, j;

    ...

    proc_col = 1;
    for ( j=1; j<=c; j++)      /* calculate the column size of the grid */
        proc_col *= 2;

    if ( pid%proc_col == 0 ) /* the first column in the grid */
    {
        *left = -1;          /* tells MPI not to perform send/rcv */
        *right = pid+1;
    }
    else if ( pid%proc_col == proc_col-1 ) /* the last column */
    {
        *left = pid-1;
        *right = -1;        /* tells MPI not to perform send/rcv */
    }
    else
    {
        *left = pid-1;
        *right = pid+1;
    }

    if ( proc_col == total_p_num) /* no rows in the grid */
    {
        *up = -1;
        *down = -1;
    }
    else if ( pid < proc_col ) /* the first row */
    {
        *up = pid+proc_col; /* tells MPI not to perform send/rcv */
        *down = -1;         /* tells MPI not to perform send/rcv */
    }
    else if ( pid >= total_p_num-proc_col ) /* the last row */
    {
        *up = -1;          /* tells MPI not to perform send/rcv */
        *down = pid-proc_col;
    }
    else
    {
        *up = pid+proc_col;
        *down = pid-proc_col;
    }
}

```

```

}

Update_Boundary_Condition ( double ** solution_array, int mrow,
int mcol, int k, int left, int right, int up, int down ) {
    MPI_Status status;
    int i;
    if ( k % 2 == 0 ) { /* even numbered processes */
        MPI_Send(&(solution_array[mrow][1]), mcol, MPI_DOUBLE, right, 0, MPI_COMM_WORLD);
        MPI_Recv(&(solution_array[0][1]), mcol, MPI_DOUBLE, left, 0, MPI_COMM_WORLD,
                &status);
        MPI_Send(&(solution_array[1][1]), mcol, MPI_DOUBLE, left, 1, MPI_COMM_WORLD);
        MPI_Recv(&(solution_array[mrow+1][1]), mcol, MPI_DOUBLE, right, 1, MPI_COMM_WORLD,
                &status);
    }
    else { /* odd numbered processes */
        MPI_Recv(&(solution_array[0][1]), mcol, MPI_DOUBLE, left, 0, MPI_COMM_WORLD,
                &status);
        MPI_Send(&(solution_array[mrow][1]), mcol, MPI_DOUBLE, right, 0, MPI_COMM_WORLD);
        MPI_Recv(&(solution_array[mrow+1][1]), mcol, MPI_DOUBLE, right, 1, MPI_COMM_WORLD,
                &status);
        MPI_Send(&(solution_array[1][1]), mcol, MPI_DOUBLE, left, 1, MPI_COMM_WORLD);
    }

    ...
    if ( k % 2 == 0 ) /* even numbered processes */
    {
        MPI_Send(sbuf_up, mrow, MPI_DOUBLE, up, 2, MPI_COMM_WORLD);
        MPI_Recv(rbuf_down, mrow, MPI_DOUBLE, down, 2, MPI_COMM_WORLD, &status);
        MPI_Send(sbuf_down, mrow, MPI_DOUBLE, down, 3, MPI_COMM_WORLD);
        MPI_Recv(rbuf_up, mrow, MPI_DOUBLE, up, 3, MPI_COMM_WORLD, &status);
    }
    else /* odd numbered processes */
    {
        MPI_Recv(rbuf_down, mrow, MPI_DOUBLE, down, 2, MPI_COMM_WORLD, &status);
        MPI_Send(sbuf_up, mrow, MPI_DOUBLE, up, 2, MPI_COMM_WORLD);
        MPI_Recv(rbuf_up, mrow, MPI_DOUBLE, up, 3, MPI_COMM_WORLD, &status);
        MPI_Send(sbuf_down, mrow, MPI_DOUBLE, down, 3, MPI_COMM_WORLD);
    }
    ...
}

```

Note that the communicator and status arguments of the MPI API are eliminated

in the ClusterGOP API and that the VisualGOP, which will be described in the next chapter, hides the message details from the programmer unless the programmer is required to modify them explicitly. This makes programming in ClusterGOP simpler and more flexible.

3.4.2 MPMD Programming Example

In this example, we consider the problem of developing a library to compute $C = A \times B$, where A , B and C are dense matrices of size $N \times N$ (e.g., see Eq. 3.4.2).

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik} * B_{kj} \quad (3.4.2)$$

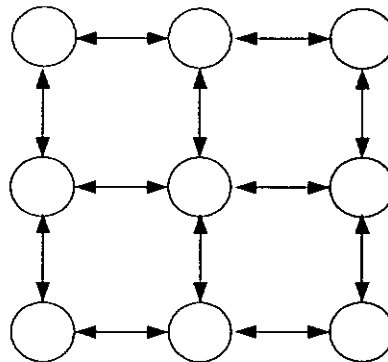


Figure 3.4: Parallel matrix multiplication structure

As shown in Figure 3.4, 3×3 mesh is defined. Besides $N \times N$ nodes in the mesh, there is an additional node, named master, which is connected to all the nodes of the mesh. There are two types of programs in this example: distributor and

calculator. There is only one instance of `distributor`, which is associated with the "master" node. Each mesh node (`node1` to `node9`) is associated with an instance of `calculator`. The `distributor` program first decomposes the matrices `A` and `B` into blocks, whose sizes are determined by the mesh's dimension. It then distributes the blocks to the nodes on the left most column (`node1`, `node4`, and `node7`) and the nodes on the bottom rows (`node1`, `node2`, `node3`), respectively. Each `calculator` receives a block of matrix `A` and matrix `B` from its left edge and bottom edge, and also propagates the block along its right edge and top edge. After data distribution, each `calculator` calculates the partial product and sends it back to the `distributor`. The `distributor` assembles all the partial products and displays the final result.

The programmer has support by `VisualGOP` and `ClusterGOP` in building this application:

- *Data distribution.* In writing the code for "distributor", the programmer must decompose the matrix `A` and `B` into blocks. Instead of writing the code manually, the programmer can select the data distribution option in `VisualGOP`. `VisualGOP` automatically generates the distributed object by using the API provided by GA toolkit.
- *MPMD representation.* In MPI, the SPMD version of the program needs to calculate the rank ID of the destination node. The MPMD program written in `ClusterGOP`, however, can be separated into tasks and each node is associates

with an independent program source. In ClusterGOP, the function `GetNode` returns the destination node by the edge, so that the program is not required to calculate the rank ID. The code structure is simplified and programmers can have a clear view of the application logic. The following code segments show the difference between the MPI SPMD and the ClusterGOP MPMD programs.

```

/* MPI SPMD program */
/* MPI needs to get the msg tag and size before receiving the msg */
MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
rcv_nid = status.MPI_SOURCE;
MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
rcv_nid = status.MPI_SOURCE;
tag = status.MPI_TAG;
MPI_Get_count(&status, MPI_INT, &num);
MPI_Recv(buf, num, MPI_INT, rcv_nid, tag, MPI_COMM_WORLD, &status);
if(tag<1000){ /* Matrix A */
    ... /* processing received data */
    if(nid % msize != 0)
        MPI_Send(buf, num, MPI_INT, nid+1, tag, MPI_COMM_WORLD);
}
else { /* Matrix B */
    ... /* processing received data */
    if((int)(nid-1)/msize != msize-1)
        MPI_Send(buf, num, MPI_INT, nid+msize, tag, MPI_COMM_WORLD);
}
}
/* ClusterGOP MPMD program on node1 */
for(i=0; i<2; i++){
    Urecv(ggraph, GOP_ANY_NODE, msg1);
    if(tag<1000){ /* Matrix A */
        ... /* processing received data */
        Usend(ggraph, GetNode("rightn1"), msg1, ASYN);
    }
    else { /* Matrix B */
        ... /* processing received data */
        Usend(ggraph, GetNode("top edge"), msg1, ASYN);
    }
}
/* ClusterGOP MPMD program on node9 */
for(i=0; i<2; i++){

```

```
Urecv(ggraph, GOP_ANY_NODE, msg1);
if(tag<1000){ /* Matrix A */
    ... /* processing received data */
}
else { /* Matrix B */
    ... /* processing received data */
}
}
```

ClusterGOP provides a GOP library, which is higher level than MPI library, for programmer to build parallel applications. The library contains Graph-oriented primitives (ClusterGOP API) for programmers to write programs for parallel programs and the implementations hide the programmer from the underlying MPI programming details. Programmers can thus concentrate on the logical design of an application, ignoring unnecessary low-level details. In the program design, the programmer does not require to remember or use the process ID of destination node. Instead of using process ID, ClusterGOP provides nodes with names which are more easy to identify the nodes in the application. Also, the destination node can be dynamically generated by querying the logical graph (i.e. parent nodes/child nodes), so that the programmer is not required to write a routine to calculate the destination node or assign a fixed process ID into the send/receive functions. This facilitates the design process of parallel applications.

3.5 Summary

ClusterGOP provides several high-level features to support message-passing programming in a parallel computing environment. A user-defined graph helps in better understanding and programming the parallel structure of the program code. The programmer is not required to understand the syntax and other low-level details of the underlying MPI message-passing system. The ClusterGOP APIs can help to hide the implementation details, and support for Node-to-Processor and LP-to-Node mapping can greatly help the programmer deploy the application and manage the system resources. The next chapter will continue our examples in working with VisualGOP, which provides a visual and interactive user interface for programmer to design their parallel application.

Chapter 4

VisualGOP

This chapter provides a detailed description of VisualGOP which provides support for developing graph-oriented parallel programs. We first discuss the framework design of VisualGOP, then we continue to describe our program examples in the previous chapter, to show how VisualGOP supports the visual programming design. We first describe how to construct the logical graph using tools in VisualGOP, and then examines program editing feature. After that, we describe the mapping procedure for programmers to do the LP-to-Node and the Node-to-Processor mapping. Finally, we discuss the remote compile and execution features in VisualGOP. In later sections, we talk about other salient features in VisualGOP, such as consistency check, graph scaling support, automatic mapping, and the interpretability.

4.1 The Architecture and Framework

This section introduces the overall architecture and the visual programming framework of VisualGOP. VisualGOP consists of the following major components:

Logical graph construction. This is a component provided for constructing a GOP program using graphical aids. The graphical aids are used to represent the logical program structure from a visual perspective. VisualGOP allows the manipulation of graph structures in both graphical and textual (XML-based) forms and can transform from one form to the other.

- *LP editing tool.* This tool is used to edit the program source of LPs in a GOP program.
- *Network resource management.* This component provides programmers with control over networking resource management. It facilitates the mapping of LPs to graph nodes and the mapping of graph nodes to processors. It also allows programmers to access information about the status of the network elements.
- *Compilation tool.* This tool is used for transforming the diagrammatic representations and GOP Primitives into the target machine codes for execution.

These components are organized to form the architecture of VisualGOP, as shown in Figure 4.1. They are divided into two levels: the visual level and the non-visual

level. Each part has its own data storage and communicates with the other through a shared, common data structure.

The visual level components present the design views, which are controlled by a visual graph design editor and a text program editor, and provide the mapping and deployment tools, which are accessed through a mapping panel and a processor panel.

Figure 4.2 shows a screen dump of the main visual interface of VisualGOP.

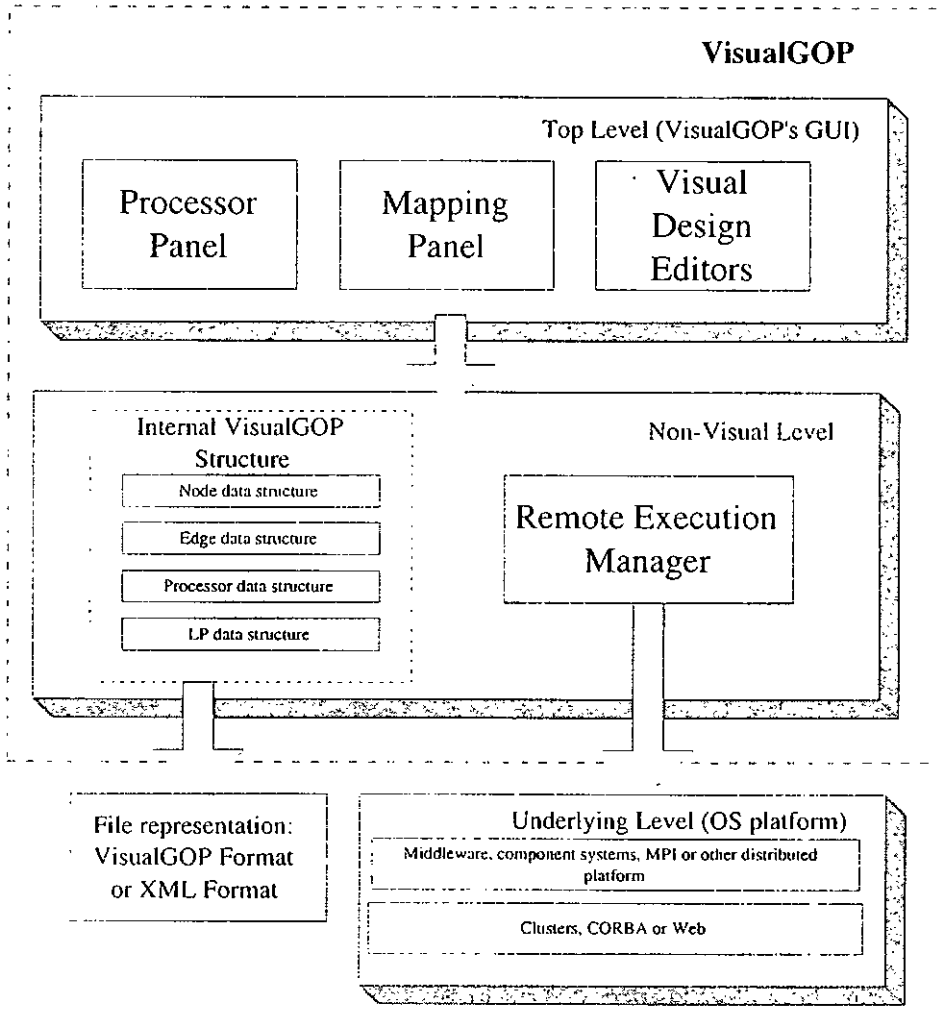


Figure 4.1: The VisualGOP architecture

The non-visual level contains components responsible for maintaining a representation for GOP program design and deployment information. This representation is kept in memory when program design takes place and is later stored in a file, in either a format internal to VisualGOP or in an XML format. This level also contains the distributed Remote Execution Manager, which makes use of the stored information to execute the constructed GOP program on a given platform.

Using VisualGOP

When programming in VisualGOP, the programmer starts program development with building a highly abstracted design and then transforms the design successively into an increasingly more detailed solution. More specifically, VisualGOP separates the program design and configuration (i.e., the definition of the logical graph) from the implementation of the program components (i.e., the coding of the LPs). This distinction features a form of design which helps in reconfiguring the program structure independent of LP coding.

The visual programming process under VisualGOP consists of the following iterative stages:

- *Specify the logical graph representing the configuration of a parallel program.*

The logical graph consists of a set of nodes representing LPs and a set of edges representing the interactions between the LPs in the program. The *graph design editor* is used by the programmer to visually create and represent the logical

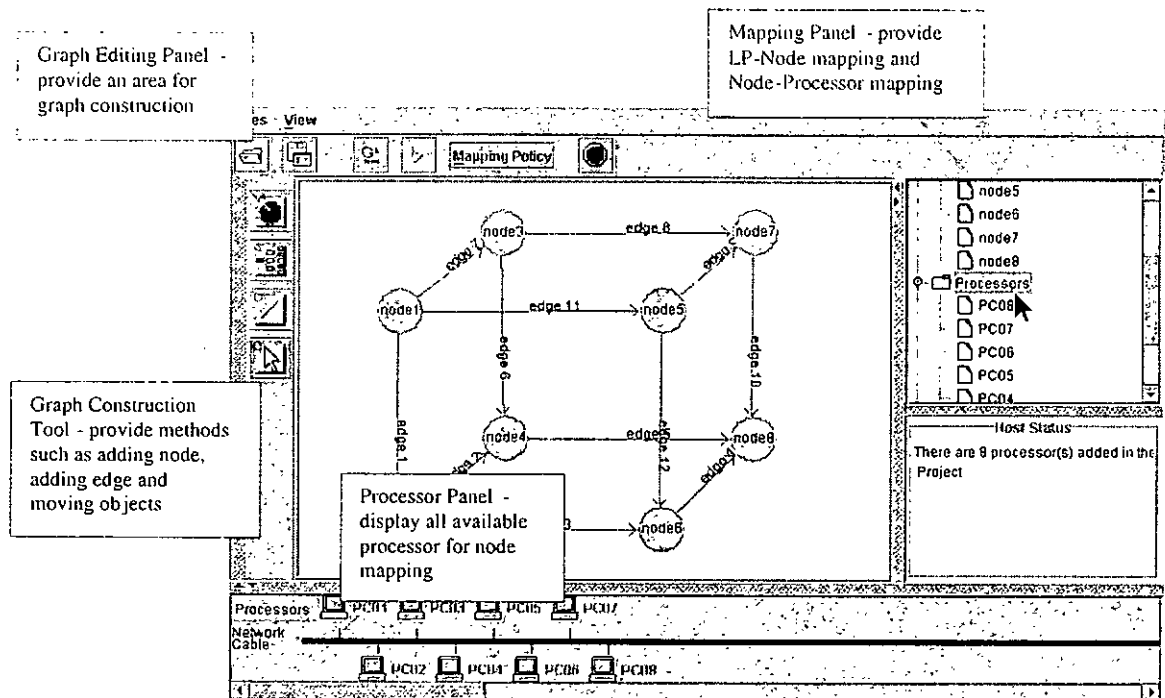


Figure 4.2: The main screen of VisualGOP

graph.

- *Create LPs of the parallel program and map them to the nodes in the logical graph.* There are two ways to do this. One way is to create the LPs first and then bind them to the graph nodes. Another way is to combine the two steps into one - click on a node of the graph to open the text editor by which the code of the LP mapped to the node can be entered.
- *Map the nodes of the graph to the processors of the underlying network.* The *Mapping panel* of VisualGOP displays the GOP program elements (nodes, processors, LPs) in a hierarchical tree structure. LPs, nodes and processors can be

added to and deleted from the panel. The programmer can use drag-and-drop to bind and unbind the LPs to graph nodes and the graph nodes to processors. A node's detailed binding information can be viewed by selecting and clicking the node. The *Processor panel* provides icons for displaying processors and their connections. When a processor is added, a new processor icon will appear on the panel. For Node-to-Processor mapping, the panel also provides the drag and drop function to bind and unbind graph nodes to one of the processors in the panel.

- *Compile the LPs and execute the application.* When the programmer needs to deploy the program to other platforms, VisualGOP will first distribute the graph information, LP source files, and the network information to the target machines. Source files are then compiled on the target machines. Finally, the constructed GOP programs are executed on the specified processors. Outputs will be displayed on the VisualGOP.

The following sections (Section 4.2 - 4.4) describe the basic features which are supported by VisualGOP. The demonstration uses the SPMD and MPMD examples in the previous chapter.

4.2 Program Construction

Graphical Programming

VisualGOP uses the graph abstraction method to represent parallel programs. It divides a parallel program into several LPs and defines their interactions. Each graph node can be allocated on a processor in a parallel system and then the LPs can be assigned to the nodes. As LPs are finally located on parallel processors, interactive behaviors describe the message-passing mechanism that is performed between parallel processors.

A graph is constructed in VisualGOP as a structure containing data types for describing a conceptual graph. First, an appropriate logical graph of the parallel application should be well designed. Based on the design, programmers can use the Graph Editing Panel to draw the graph, which will be translated to the textual form. The Graph Editing Panel displays the graphical construction view of VisualGOP, and this is where all of the editing of the program's logical graph structure takes place. Controls are provided for drawing the graph components within this panel. These controls include functions for adding nodes, subgraph nodes, and edges, as well as for moving nodes. We use our previous SPMD example to demonstrate the graph construction. Figure 4.3 shows how nodes can be added to the graph by simply clicking the add-node button in the Graph Construction Tool. Edges can be added between two nodes by joining the source and the destination nodes. Like the program

graph of FDM, we create four nodes and link the nodes with edges.

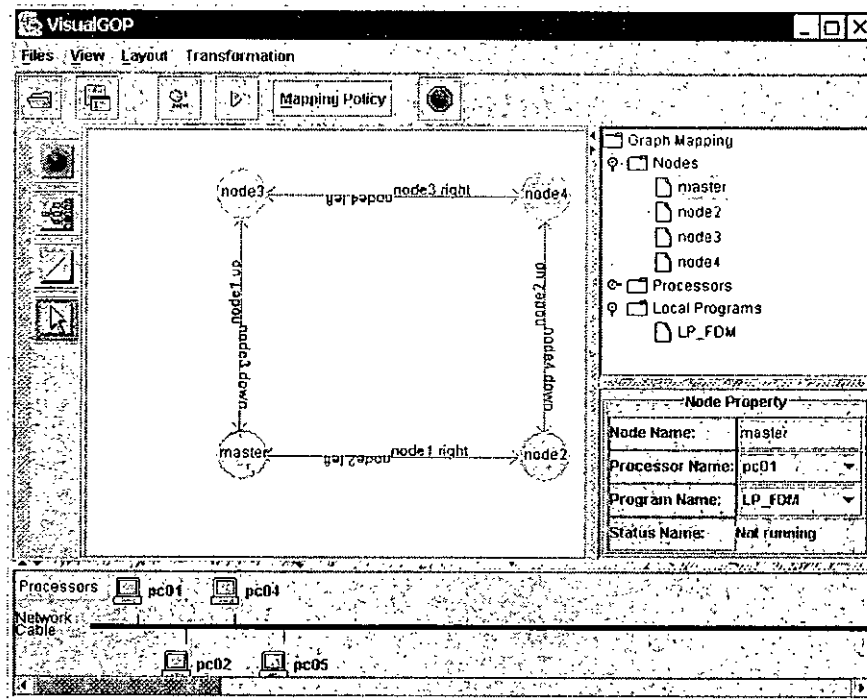


Figure 4.3: Logical graph for 4 processors in VisualGOP

A visual component (node or edge) can be manipulated directly on the screen when constructing a program. VisualGOP has one or more display areas, each depicting certain properties of the visual component. Properties of a visual class can be represented visually, such as a button or a pull-down menu. The associated panel, the Mapping Panel, presents the graph component properties in a simple tree style. The component is classified into one of three categories: node, processor and LP. Each component shows its properties in the program, and the mapping relationship that it belongs to. It is updated automatically whenever the program structure is

modified in the Graph Editing Panel. It is useful not only as an indicator of the overall program structure, but also as a navigational aid for browsing the different parts of the system. Navigation is assisted by highlighting the part of the tree diagram that corresponds to the component being displayed in the Graph Editing Panel. Due to the fixed screen size, only a limited number of visual components can be viewed and operated on the screen. It is desirable for programmers to hide the information that is not currently needed, and to show as many objects as they want to view and operate.

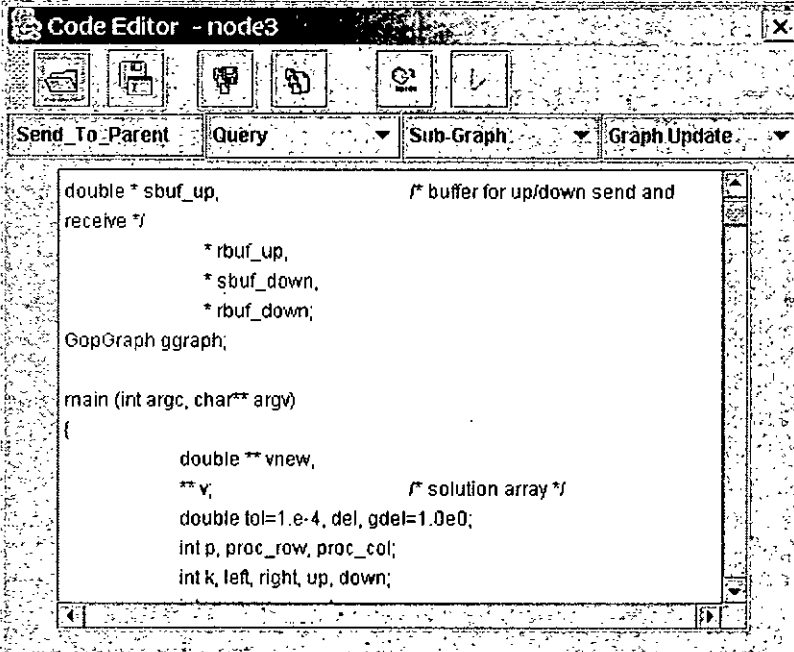
LP Code Editing

The Code Editing Panel (program editor) is one of the key components of the VisualGOP and it serves programmers in several tasks and different contexts. The editor is used to define data (program sources) attached to VisualGOP graph components. Each node of a program is displayed and edited within an appropriate edit dialog. Programmers interact with the editor through input devices that are uniformly interpreted by the editor according to the context of the action.

The program editor provides consistency check so as to prevent the creation of graph semantic errors. For example, the programmer is not allowed to make incorrect connections, such as between a node and a non-existent node. To assist the programmer with graph programming, the editor has features to ensure that newly created or modified node or edge names are correct within the logical graph structure. Further

details of consistency check are discussed in Section 4.5. VisualGOP also provides different sets of LPs for mapping a node to the program source. There are two methods to invoke the program editor:

1. Start editing a new program source.
2. Invoke the mapped LP by double click on the node.



```

Code Editor - node3
Send_To_Parent Query Sub-Graph Graph Update
double * sbuf_up, /* buffer for up/down send and
receive */
    * rbuf_up,
    * sbuf_down,
    * rbuf_down;
GopGraph ggraph;
main (int argc, char** argv)
{
    double ** vnew,
    ** v; /* solution array */
    double tol=1.e-4, del, gdel=1.0e0;
    int p, proc_row, proc_col;
    int k, left, right, up, down;

```

Figure 4.4: Editing the LP

To edit a new program, the programmer starts the program editor from the Mapping Panel (see Figure 4.4). In our SPMD example, the programmer use the C language to write an LPs, the LP_FDM, which calculates and submits the values and then collects the values from the neighbor nodes. On the other hand, the programmer

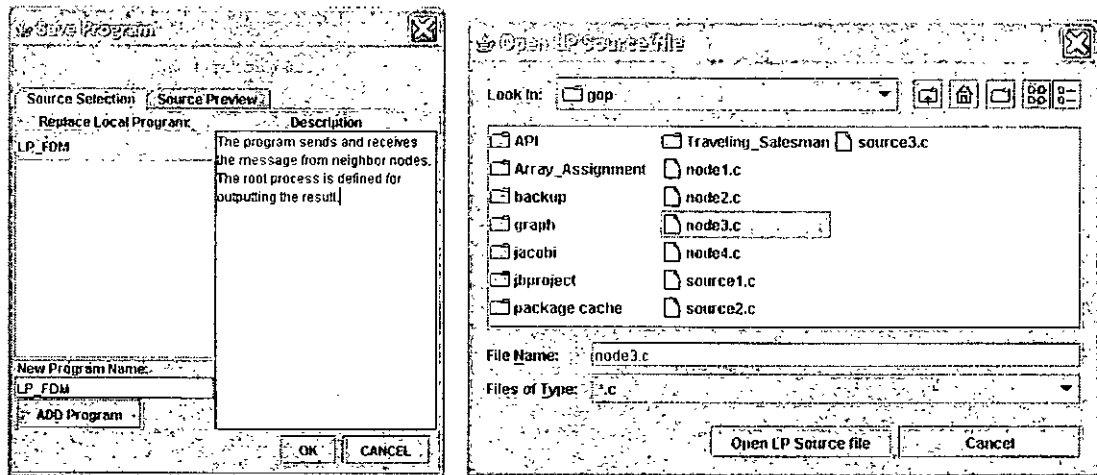


Figure 4.5: Loading and Creating LPs

can load the existing program source into the program editor (see Figure 4.5). After that, the new LP will bind to the node automatically.

Generation of GOP Primitives

The GOP model provides high-level abstractions for programming parallel applications, easing the expression of parallelism, configuration, communication and coordination by directly supporting logical graph operations (GOP primitives). In VisualGOP, programmers are provided with a variety of abstractions that are useful in coding parallel programs. By using the basic GOP primitives, the programmers do not need to know much detail about the low-level communication mechanism; they only need to know the usage of the selected GOP primitive. In the Code Editing Panel, GOP primitives can be automatically generated from visual icons. As described in Chapter 3, the primitives can be categorized into several groups: Communication,

Query, Sub-graph generation and Graph Update. Programmers can generate GOP primitives via two methods:

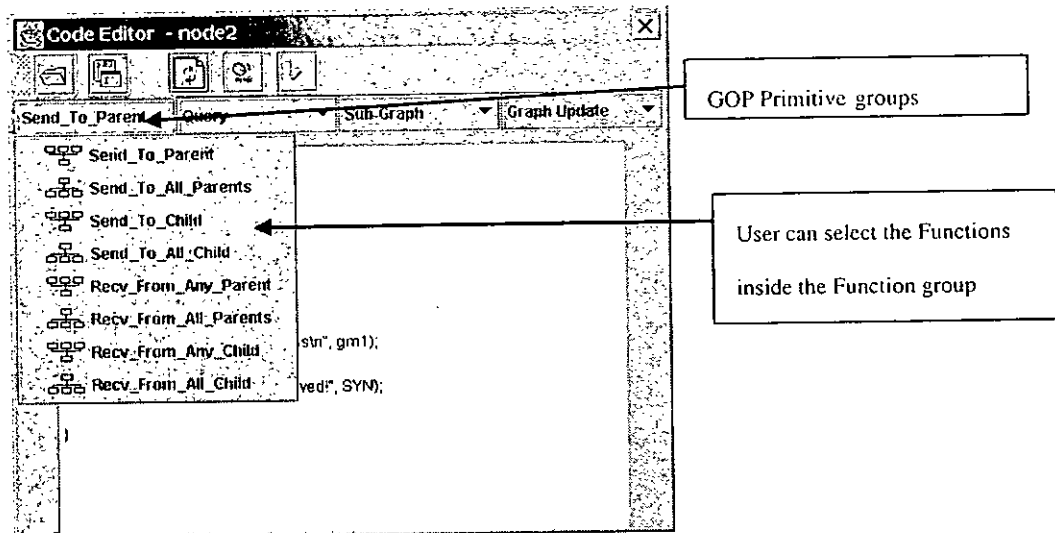


Figure 4.6: The GOP Primitive Menu

1. Adding the GOP Primitives from the program editor's menu directly (see Figure 4.6).
2. Dragging and dropping the node into the program editor to update the node parameter (see Figure 4.7).

After we have created the logical graph and LPs, we need to bind the LPs to the nodes. We also need to setup a processor list for binding the nodes to the processors, so that the LPs can be bound to processors and their processes can communicate through the GOP system. In the next section, we describe the steps to build up the mappings in VisualGOP.

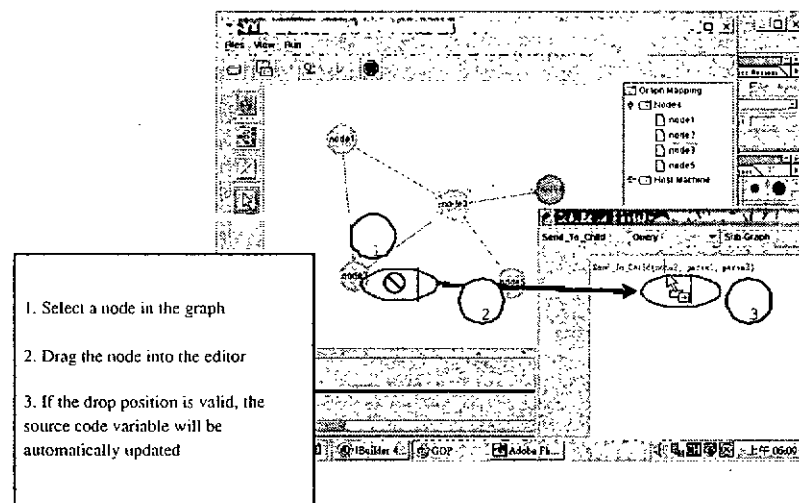


Figure 4.7: Manipulating graph nodes directly

4.3 Two-Step Mappings

An important task in implementing GOP is to manage the mapping of LPs to the nodes of a logical graph (LP-to-Node Mapping), and the mapping of the graph nodes to the underlying network processors (Node-to-Processor Mapping). LPs must be bound to the nodes of a logical graph for naming, configuration, and communication purposes. When the Node-to-Processor is mapped, the solution to the problem is straightforward if the programmer specifies the mapping. Otherwise, the GOP system should provide support for task allocation in order to make efficient use of system resources and/or to speed up the computations. Once mapped, the graph node has all this required information (by Node-to-Processor Mapping) such as IP address/hostname, compiler path, etc.

4.3.1 An Example of Mapping

In our MPMD example of parallel matrix, we have one master node and nine nodes connected with edges, as shown in Figure 4.8. Although the application is designed using MPMD style, there are still some programs must be shared with the same program source (e.g. node1, node2, node4 and node5). VisualGOP provides the mapping interface that programmers can use for managing the program source mapping, so that they need not worry about the complex task of resource management.

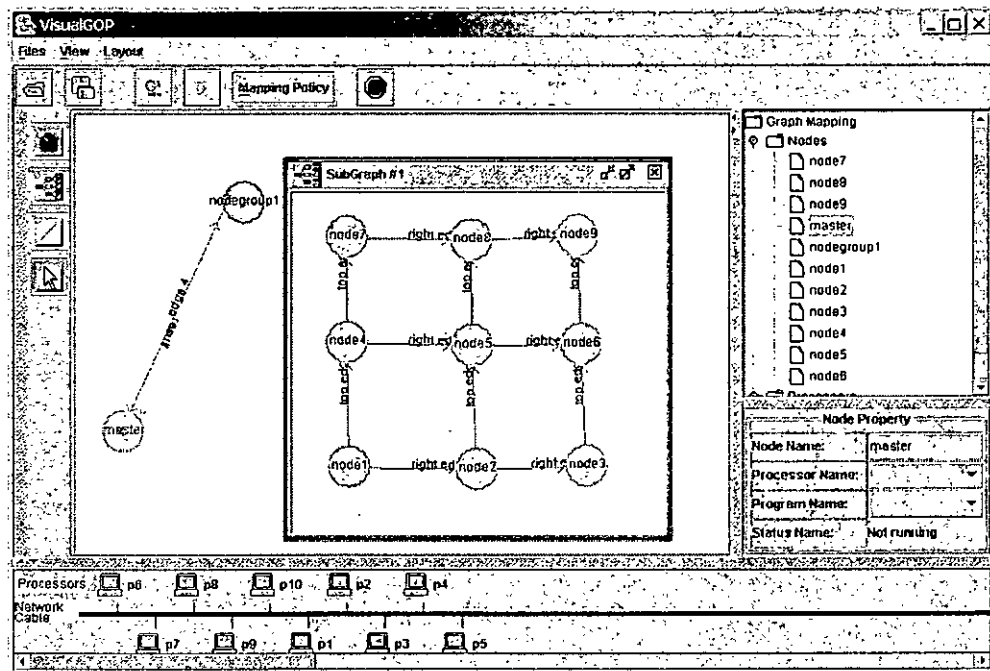


Figure 4.8: Diagram for the parallel matrix multiplication

When the programmer selects the status of a specific node, an LP can be chosen from the node property's pull-down menu. After the selection, the LP is mapped to that node (See Figure 4.9). The mapping can also be carried out by dragging and

dropping the node into one of the processors in the Processor Panel. Let us define a map of the MPMD example, named M1, which defines the relationships between the graph nodes and the LPs. In the map, there are several types of LPs: the `distributor`, which receives and distributes the data, and the `calculator`, which calculate and submit their own partial data to the `distributor` and receive data from the neighbor nodes and from the `distributor`. Our definition of M1 is (given in the C language, where LV-MAP is the corresponding map data type):

```
LV-MAP M1 =
    { {0, "distributor"}, {1, "calculatorA"}, {2, "calculatorA"}, {3, "calculatorB"},
      {4, "calculatorA"}, {5, "calculatorA"}, {6, "calculatorB"}, {7, "calculatorC"}, {8,
      "calculatorC"}, {9, "calculatorD"} }
```

In the same way, the programmer can specify a mapping M2 of the graph nodes onto the processors. It is assumed in VisualGOP that each node will be executed on a different processor. For example, an LP on processor A sends a message to another LP on processor B. The message passes from one platform to another, so the Node-to-Processor mapping must perform such a task.

The Processor Panel displays information about processor availability. If it is currently assigned, the relevant node is indicated. The whole application is run using parallel processors. Programmers can manually choose which processor runs each of the LPs or this can be assigned by VisualGOP automatically. With the aid of the

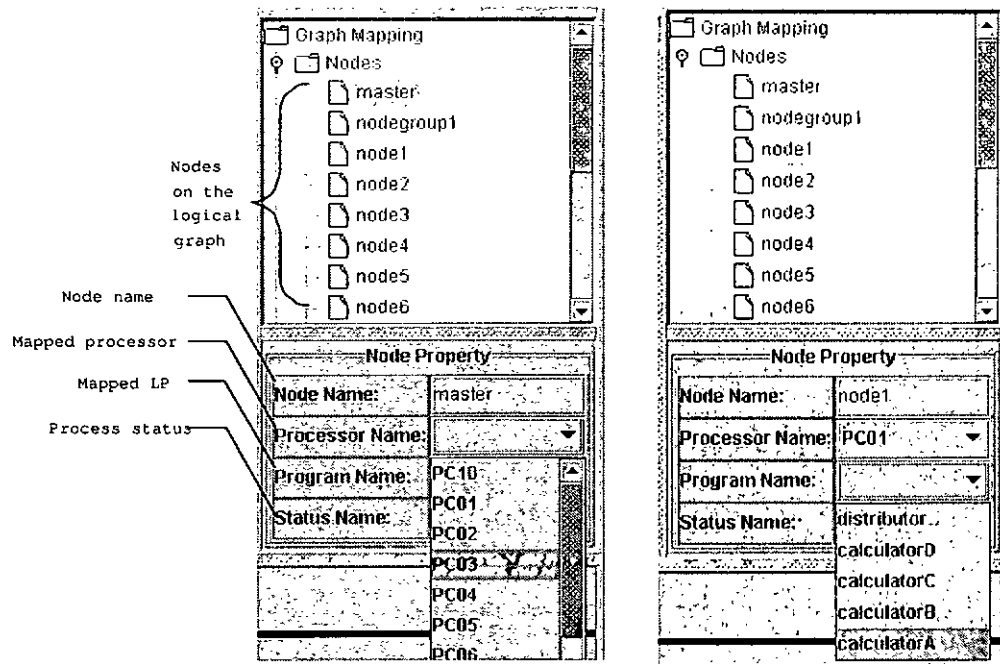


Figure 4.9: LP-to-Node and Node-to-Processor mapping

Processor Panel the node is mapped onto the target processor.

The mapping of the node onto the target processor is carried out in two steps. First, the programmer has to configure the target processor in the processor configuration dialog (see Figure 4.10). Inside the dialog, the programmer can specify the new identify name, IP address or hostname (currently only supported for one processor per machine). Second, the node is mapped onto the configured processor, either manually or automatically. In the manual mode, the programmer clicks on a node in the node property and specifies the name of the target processor. The automatic mode will be discussed in Section 4.7.

The final step of building an application is to compile the LPs and to execute it.

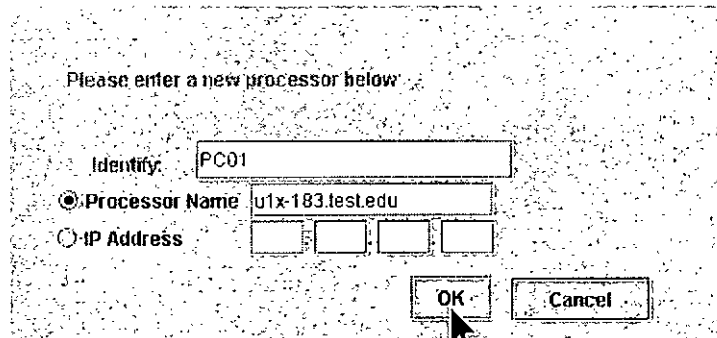


Figure 4.10: Processor configuration

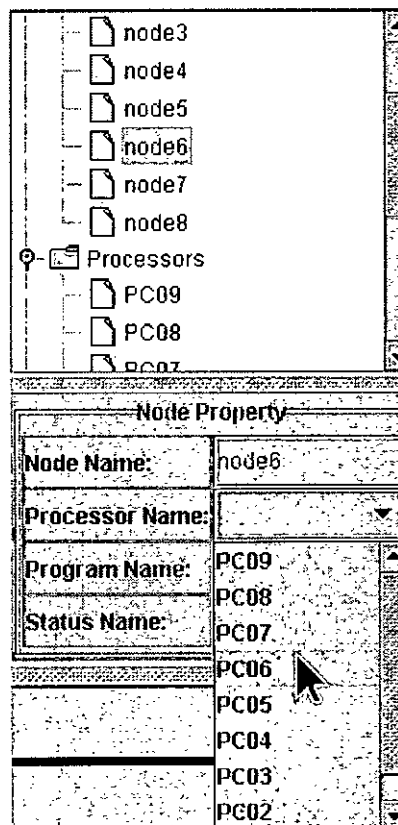


Figure 4.11: Node-to-Processor mapping

We will show how VisualGOP support the features in the next section.

4.4 Remote Compilation and Execution

VisualGOP supports automatic code generation, which generates the compiled machine code and starts execution remotely. The code generator recognizes the LP's implementation language type and assigns the corresponding remote compilation and execution procedures to the LP, which will be compiled and executed automatically. Using this technique, code generation of VisualGOP would be simple, high-level and a single programming environment could generate the parallel program code used for a wide variety of systems. Figure 4.12 shows the process of automatic code generation for C and Java languages. Remote controlled programs installed on the target machines are listening for remote requests. When VisualGOP submits the requests for remote compilation or execution, a connection is established between VisualGOP and the remote machine.

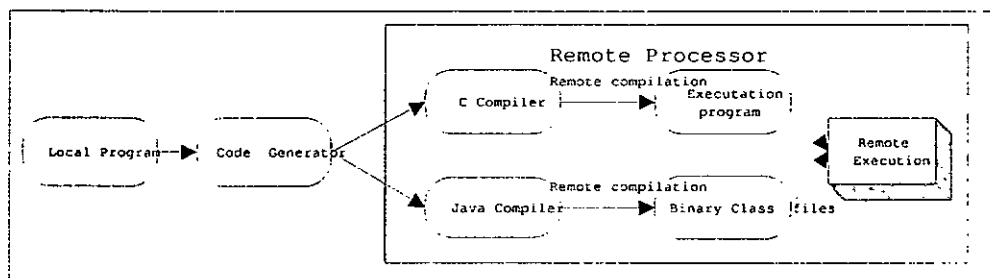


Figure 4.12: Automatic code generation framework

When the programmer selects the remote compile option within a node, a selection

dialog (see Figure 4.13) is provided for the programmer to select the processor to be used as the target for compiling the program source for execution. The machine's name is the same as that of the processor that the programmer specified in the Graph Panel. After a processor is selected, the program source is ready for compilation. The programmer is only required to input the command argument which is used for compiling or executing the LP on the target machine (see Figure 4.14). After that, the remote compilation and execution will be done automatically. The execution option is similar to setting up the compilation option. After the programmer selects the processor to run the application, the results can be viewed in the window of a VisualGOP dialog (see Figure 4.15).

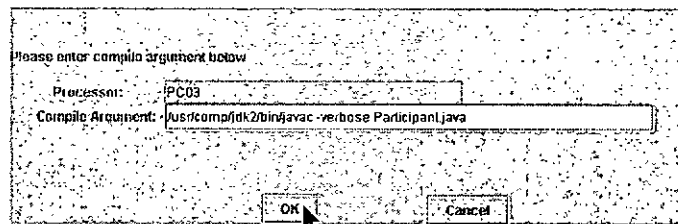


Figure 4.13: Dialog for choosing the host machine to compile

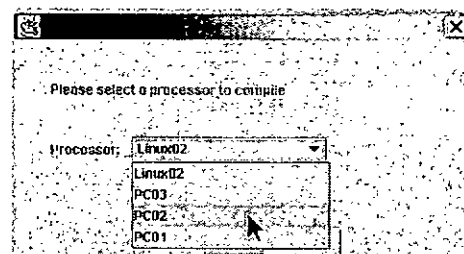


Figure 4.14: Dialog for entering the compile argument

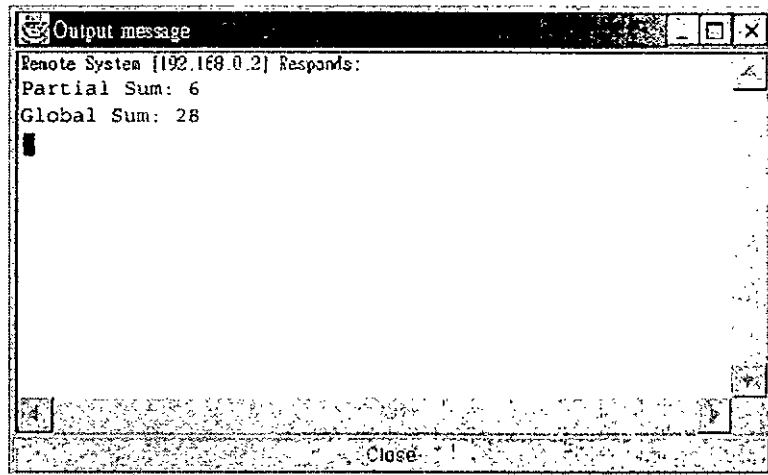


Figure 4.15: Dialog for returning the remote command result

The remaining sections in this chapter show all other features that are supported by VisualGOP. Programmers can use the features to enhance the program design and the programming efficiency.

4.5 Consistency Check

Program editors can automatically check the consistency of program code by using the graph structure specified by the programmer. When the programmer opens the Code Editing Panel, the automatic consistency checking module also starts up, as shown in the flow chart of Figure 4.16. While the programmer is editing the program source, the editor submits the current code statement to the program code parser. Then, the parser finds the supported GOP primitives and its parameters, stores them in a variable array called the Function Parameter Array. After that, the parser gets the current editing node's name from the Graph Structure (an internal VisualGOP

structure). Based on the $N \times N$ matrix truth table, the parser can find the parent nodes of any child node and store the result in the Valid Node Array. By comparing the Function Parameter Array and the Valid Node Array, the consistency checking system can determine whether the parameters in the GOP primitives contain the valid parent or child node name. If the parameter is found to be invalid, the corresponding part of the program code will be highlighted to indicate that the parameter is inconsistent with the graph structure. Then, the programmer is expected to notice the problem and correct the parameter value.

Figure 4.17 shows active screen of the Participant in the program editor, using in this example the GOP primitive `Usend`. `Usend` defines a unicast message-passing primitive, delivering a message from the current node to another node. The second parameter of `Usend` accepts a node variable or value. When the parameter is detected invalidated by the program editor, it will be highlighted in color. If the parameter is a variable (i.e., a reference value), it will be underlined, and the corresponding variable value will be found throughout the program source automatically.

When the programmer clicks on the invalid parameter value, a list of valid values will pop up, allowing the programmer to select the correct node name (see Figure 4.18). If the selected parameter is a static string, the program editor updates the string directly. Otherwise, program editor will try to find out the actual value of the selected variable, and update it with the valid node value.

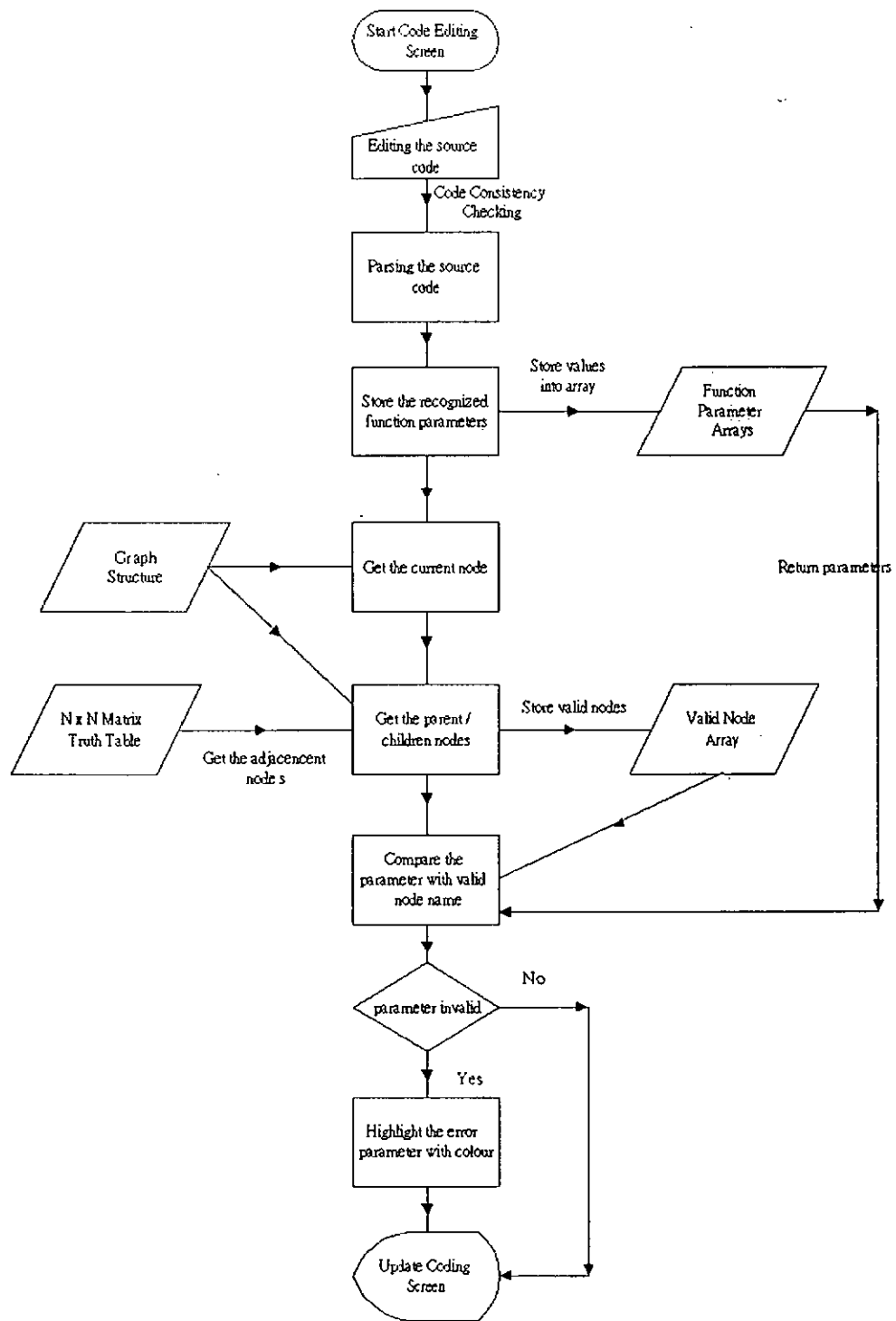


Figure 4.16: Flow chart of automatic consistency check

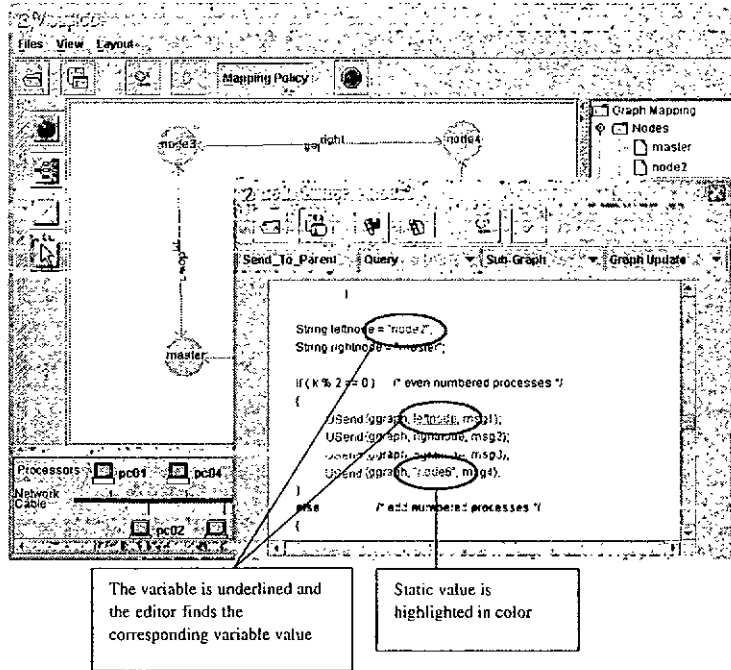


Figure 4.17: Participant program code

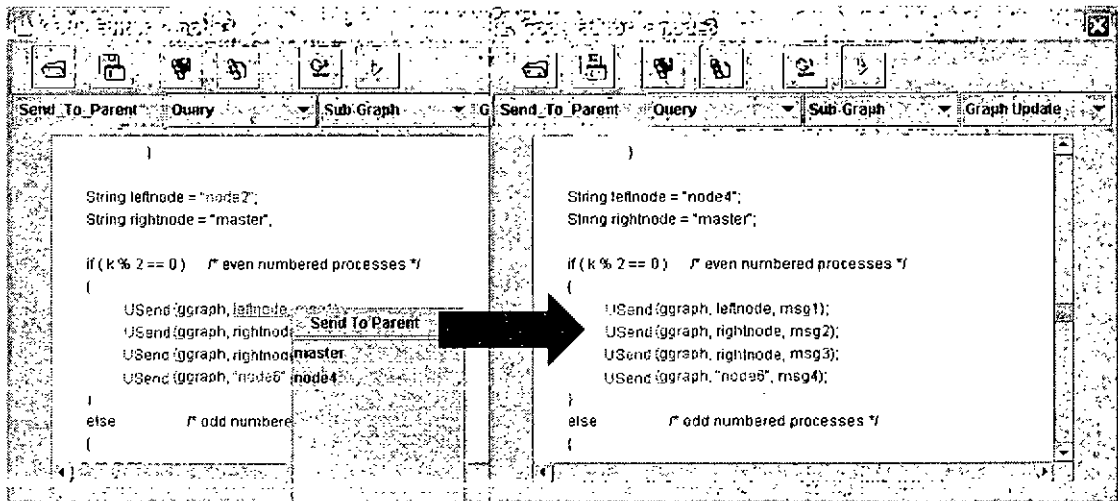


Figure 4.18: Validation in a participant program

To correct the invalid node value, the programmer can update the value by dragging and dropping the node into the GOP primitive. In Figure 4.19, the programmer has two options (node4 or master) for updating the parameter. By providing the programmer with automated, intelligent assistance throughout the software design process, VisualGOP creates a flexible programming environment and eliminates most of the mundane clerical tasks.

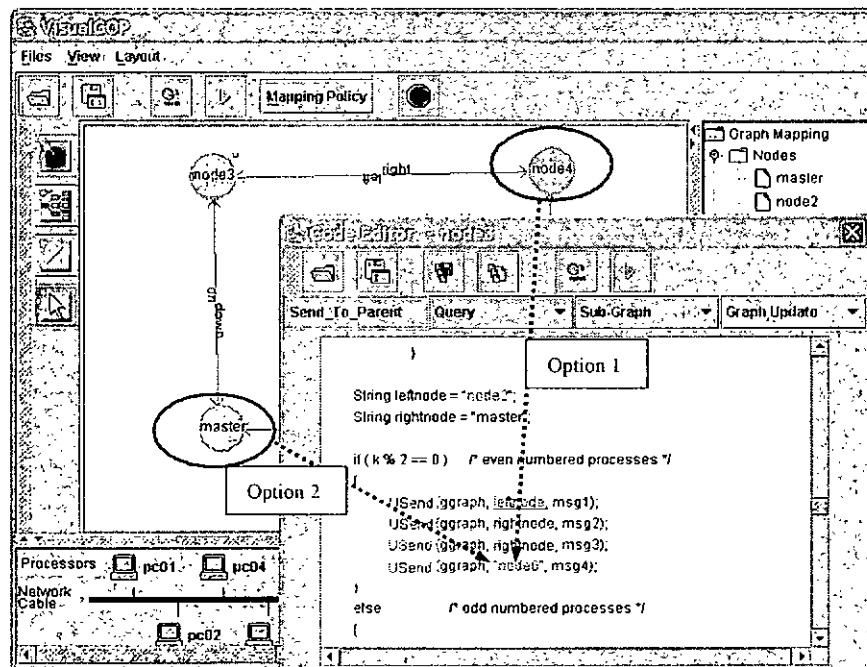


Figure 4.19: Updating GOP primitive parameter through the Graph Editing Panel

4.6 Graph Scaling

When the programmer needs to create a realistic graphical representation for parallel application, a task graph should be defined as an abstraction of program structure. In

addition, it should be highly scalable so as to be adaptable to parameters such as the size of the problem and number of processors. Our graph scaling approach is based on Task Interaction Graph (TIG) [30, 43] by which the graph scaling algorithms and the graph mapping strategy will be implemented. In graph scaling, the nodes in the graph can be decomposed or merged, and the edges are reconstructed based on the original graph structure to produce a new task graph to match the parameters. TIG provides a concise topology to describe process-level computation and communication. It has flexible structure for graph scaling. Graph scaling can be carried out in two modes, expansion and compression. If the number of parallel tasks is less than the required problem size or the number of processors, the graph is expanded to generate more tasks. On the other hand, if the number of parallel tasks is greater than the problem size, the graph is compressed so that it includes fewer nodes, although this is a rare situation in a task graph. Also, if the number of parallel tasks is greater than the available processors, the graph may be compressed. This approach has been implemented and described in detail in another paper [10]. For our example of FDM, we use a mesh structure (see Figure 4.20) for the basic pattern of the task graph, so that the application is scalable.

VisualGOP will ask for the graph pattern, which is required for the basic graph of the application. This example uses the mesh tree template (see Figure 4.21). When editing the LPs, we use a GOP primitive that can generate the node name from the

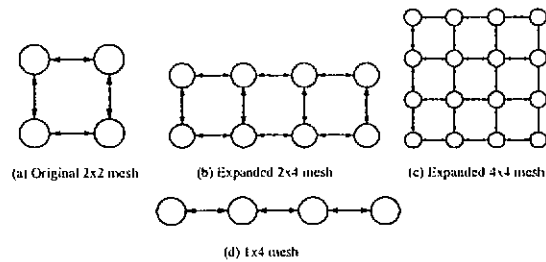


Figure 4.20: Graph expansion for mesh structure

neighbor edge, so that it is not necessary to enter the static node name into the program source. The LPs can use one copy of the source program and easy to design and maintain when the application grows large. After that, VisualGOP require the programmer to input the processor number, VisualGOP will choose graph expansion if the available processor number is larger than the graph node number, otherwise the graph compression is used. The programmer can also make a preview on the expanded or compressed graph, for accepting or rejecting the changes.

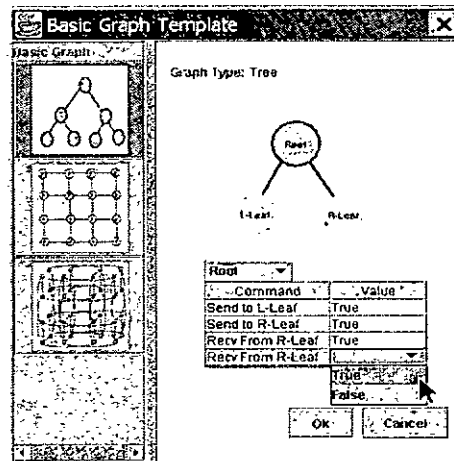


Figure 4.21: Graph Template for the Basic Graph Diagram

Another example of graph scaling uses the tree pattern for graph expansion. The

tree in Figure 4.22(a) can be expanded in two directions. It can be expanded in breadth as shown in Figure 4.22(b), in which all expanded nodes are attached to the root, and it can be expanded in depth, in which case the functional nodes spawn children beneath as shown in Figure 4.22(c).

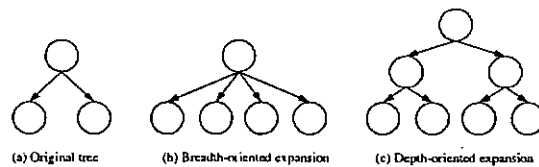


Figure 4.22: Graph expansion for tree structure

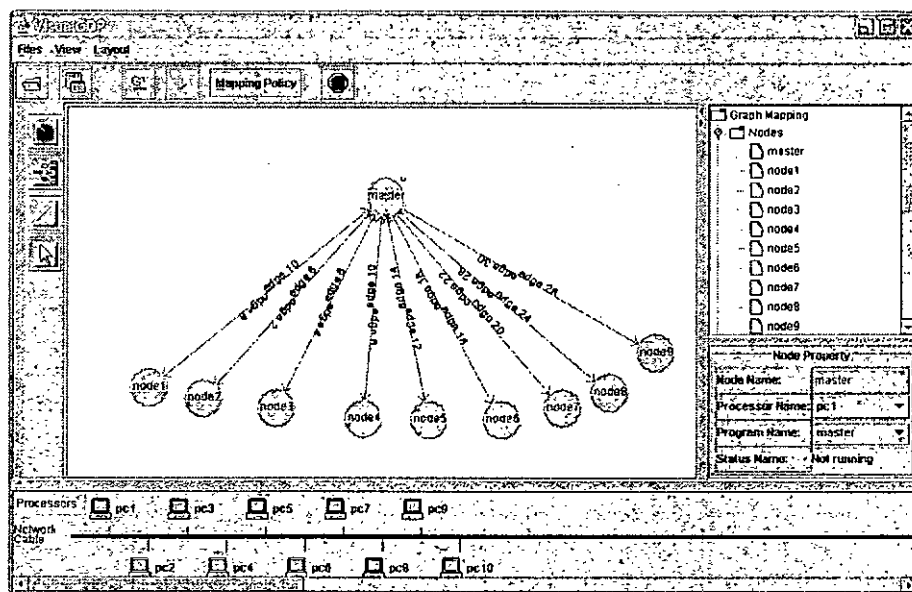


Figure 4.23: Array assignment program in VisualGOP

In this simple example, the master task initiates N number of worker tasks. It then distributes an equal portion of an array to each worker's task. Each worker's task receives its portion of the array, and performs a simple value assignment to each of its element. The value assigned to each element is simply that element's index

in the array+1. Each worker's task then sends its portion of the array back to the master task. VisualGOP allows the programmer to expand the graph using breadth-oriented expansion (see Figure 4.23). The array distribution is not affected since the programmer can program the code to calculate the array assignment according to the number of processors (or the number of processes running in the environment).

4.7 Automatic Mapping

After the program design, the programmer can either choose the LP-to-Node mapping manually or automatically. In SPMD model, all the nodes share the same copy of the program, so the mapping is simple. In the MPMD model, each node may work on different tasks. The programmer can choose a set of rules to automatically map the LP-to-Node. There are rules for classifying the LP into different groups, e.g., a range of node IDs, similar node names, and node types. Finally, a task graph will be mapped to processors. Each processor is responsible for executing a node in the task graph; i.e., there is a one-to-one correspondence between a processor and a node.

Programmers can also choose a mapping policy for the nodes-to-processors mapping. Using the mapping policy, the system binds nodes to the processors automatically. This simplifies the mapping process and helps programmers to assign the most useful/powerful processors to the more important LPs.

- *Map Processors Sequentially.* When we use this simple mapping algorithm, we

assume that process nodes are mapped to processors in their listed order. This means that the first node in the node list is matched with the first processor in the processor list, and all the others follow this simple matching order. Programmers can take advantage of this if they write the programs in the order of their relative importance, starting from the core program followed sequentially by the smaller components of the parallel application.

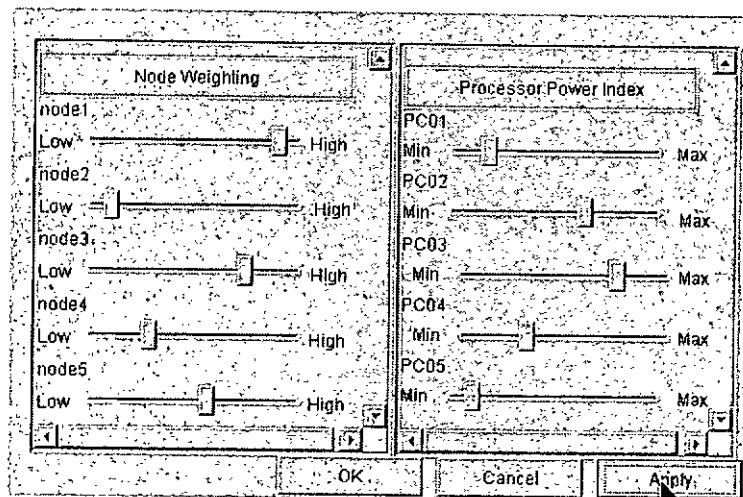


Figure 4.24: Mapping Index Table

- *Map Processors by Priority.* Unlike the mapping algorithm described above, this mapping algorithm calculates different priorities between the processors and the nodes. The processor power index and the node weighting have to be specified by the programmer. This index number, in both cases, indicates the relative priority; a larger value indicates a higher priority. This indication of relative priority allows the most powerful nodes and processors to be matched

(see Figure 4.24).

4.8 Interpretability

Representations of Graph and program configuration represented in the VisualGOP system require to be exchanged across different GOP platforms. To enable interoperability between the different tools performing various tasks on various platforms, a standard representation is required for the exchange of the GOP program design between different systems. Other benefits of the standard textual representation of logical graphs include that such structured or semi-structured graph and program representation may be transformed into any other data representation schemes automatically, and can also be easily understood by a human reader. ClusterGOP implements the interface so that the GOP program design can be accessed by Cluster nodes without any problems.

Figure 4.25 illustrates the layered architecture of GOP-XML. GOP-XML is composed of two layers, the Programming Layer and the Mediation Layer. The result LPs, graph representations and program configurations are deployed to the target machines. The GOP-XML Wrapper is a module for extracting data from each programming resource and converting into an XML file. Mediator in the mediation layer is a module for programming resource integration. The mediator controls the wrapper

and distributes the program sources to the target machines. The mediator is written using DOM [52]. The DOM specification defines the Document Object Model, a platform and language neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The Document Object Model provides a standard set of objects for representing HTML and XML documents, a standard model of how these objects can be combined, and a standard interface for accessing and manipulating them. Therefore, programmers are not required to know how the mediator works and can concentrate on designing the application.

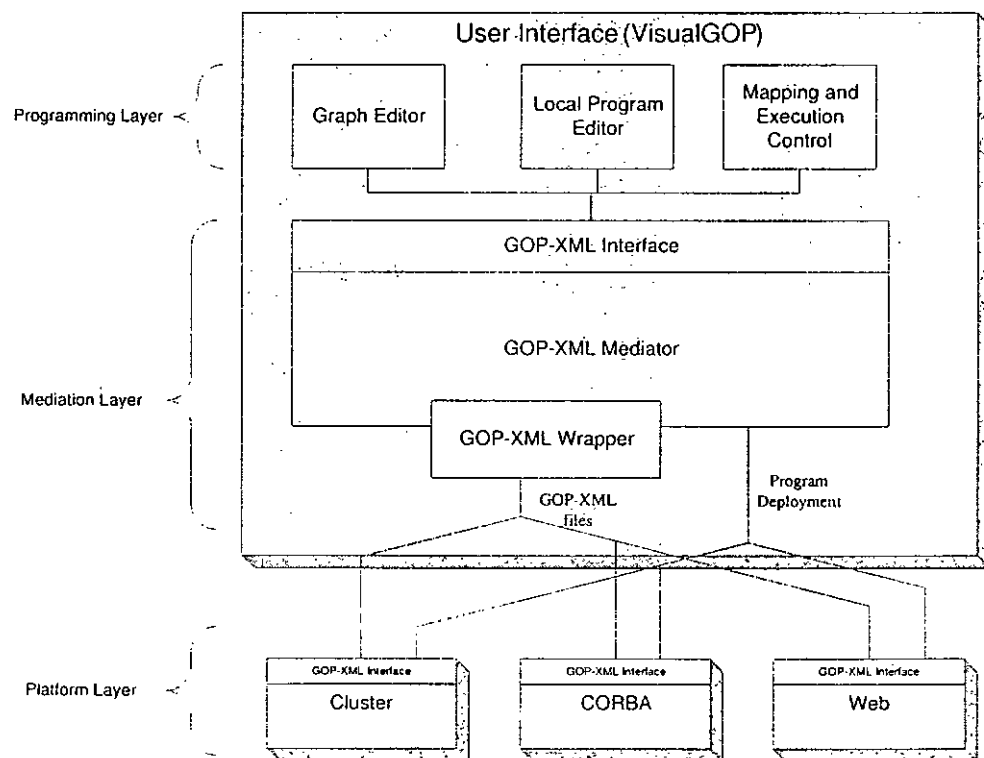


Figure 4.25: Layered Architecture of GOP-XML

4.8.1 Mediator

When the programmer needs to compile or execute the application, a GOP-XML document will be transferred to the corresponding processors. VisualGOP has been developed based on Java so it is portable to many platforms. For the GOP-XML, we use the Java API for XML Processing (JAXP) to makes it easy to process XML data in VisualGOP. JAXP leverages the parser standards SAX (Simple API for XML Parsing) and DOM (Document Object Model) so that the programmer is supported to parse the data as a stream of events or to build a tree-structured representation of it.

4.8.2 Wrapper

The main function of Wrapper is to translate the programming details into XML data format. Wrapper is composed of Query Processor, DOM Parser and Result Translator. The role of Query Processor is receiving the request from the DOM structure, resolving the processor and the LP information. The DOM Parser retrieves the items from the GOP DOM structure, which includes the graph representation and mapping relationship. Result Translator in Wrapper coordinates all the result as XML format. After generating the result with XML format, Wrapper validates results against the XML syntax.

```
<?xml version="1.0" encoding="utf-8"?> <Graph Title="Sample GOP  
Graph" Type="Tree">  
  <GraphTopology>
```

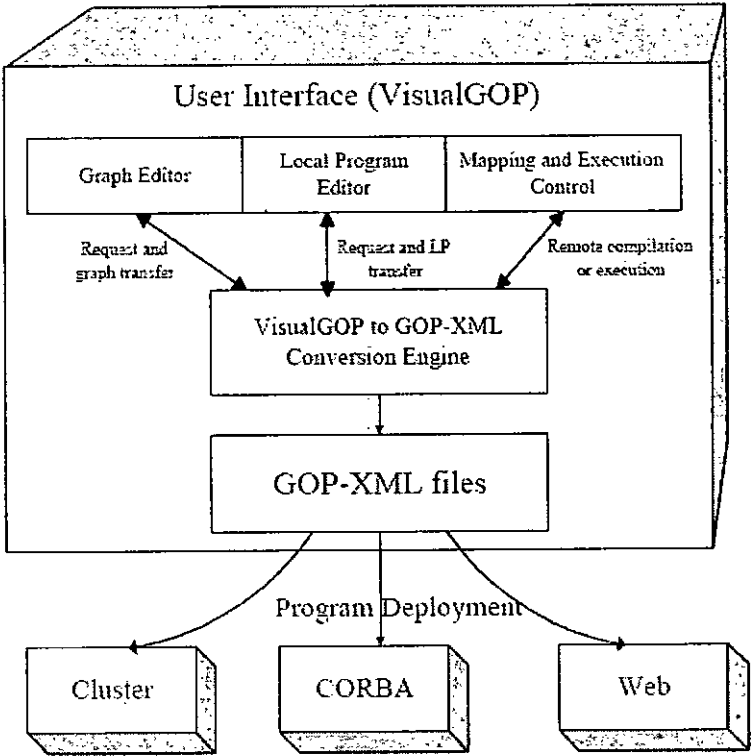


Figure 4.26: Program deployment process using GOP-XML

```

    <GraphLayout Number="0">
        <Node Name="master" IsGroup="false" Interface="Coordinator">
</Node>
        <Edge Name="edge1">
</Edge>
    </GraphLayout>
</GraphTopology>
<Processors>
    <Processor Name="pc01" Type="IP/Hostname">
</Processor>
</Processors>
<LocalPrograms>
    <LocalProgram Name="lp1" LangType="c">
</LocalProgram>
</LocalPrograms>
<API id="Simple Graph">
    <GROUP name="Communication and Synchronization"/>
    <ENTRY id="Usend"/>
</API>
<Additional>
</Additional> </Graph>

```

The XML example above shows the basic style of the graph description in the GOP-XML format. The first line describes the file as being in XML format. The second line indicates the title and the type of the graph. The third section (between the <GraphTopology> and <GraphTopology> tags) describes the logical graph in GOP. The fourth section (<Processors> tag) provides the details of the processor used. The fifth section (<LocalPrograms> tag) defines the LP. The sixth section (<API> tag) specifies the GOP primitives used in the application. The last section (<Additional> tag) adds the supplementary information about the graph, such as graph type, graph element descriptions. Attributes can also be defined as value pairs for each of the elements. The GOP-XML schema defines a set of rules to check

the allowable elements, attributes and structures in the XML. A parser has been developed to validate the syntax of any GOP-XML document.

When the programmer has to compile or execute the application, a GOPXML document is transferred to the corresponding processors. As shown in Figure 4.26, when VisualGOP is required to start the remote compilation or execution, GOP-XML Wrapper will collect the graph, the mapping and the network information and translate the data into GOP-XML format. The GOP-XML file and the LPs are then deployed to the processors of the target platform. It is important to note that GOP is independent of any particular language and platform. It can be implemented as library routines incorporated in familiar sequential languages and integrated with programming platforms such as clusters, CORBA, and the Web [11, 8].

In figure 4.27, the GOP-XML file is transferred to the target platform using the program deployment in VisualGOP. ClusterGOP runtime starts and the platform dependent parser reads the GOP-XML file. The GOP-XML is a flexible format for reading in all platforms. In different platform implementations, different XML libraries are supported for reading the GOP-XML file. ClusterGOP uses a library called Expat [15] to extract the data from the GOP-XML. Expat is a library for parsing XML documents. It is a stream oriented parser that requires setting handlers to deal with the structure that the parser discovers in the document. By using the Expat, ClusterGOP can implement functions for converting the GOP-XML document

into the program structure of the platform. In each platform or machine, the ClusterGOP runtime receives the logical graph structure and the resources configuration such as LP-to-Node and nodes-to-processors mapping. The ClusterGOP runtimes collect all the required runtime information. They initialize the parallel programs into processes, manage and control the activities of the processes.

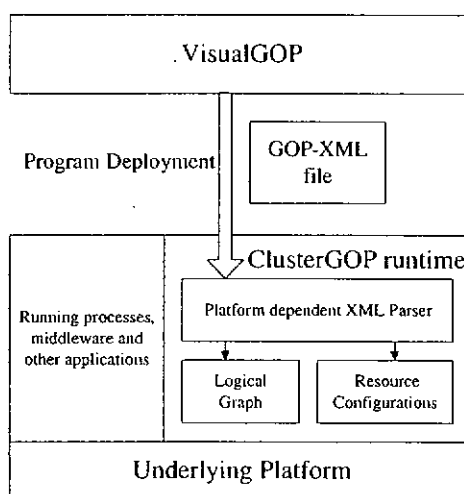


Figure 4.27: Platform independent structure in ClusterGOP

4.9 Summary

Building a parallel application is not an easy task. The complexity and large scale of the network makes the development process difficult. This chapter describes a novel graph-oriented approach to configuring and programming parallel software and a visual programming interface for the programmer to manage network nodes, processors and LPs.

VisualGOP supports a high-level program development, where the process structure is described using a GOP model. It provides integrated graphical tools for design, mapping, compiling and executing parallel programs. With the aid of the Graph Editing Panel the programmer designs the logical graph. The Coding Editing Panel allows the programmer to write LP using different programming languages, and provides a set of high-level programming primitives for writing parallel programs. It can also interact with the visual components, directly manipulates the textual source code visually. The Mapping Panel displays the information of visual components, and helps the programmer to specify the mapping of nodes to processors and the mapping of LPs to nodes visually. Additionally, VisualGOP has features to support the automatic Node-to-Processor mapping, the graph scaling, the XML-based graph representation and the MPMD programming.

The next chapter provides a detailed description of the ClusterGOP implementation.

Chapter 5

Implementation of ClusterGOP

In this chapter, we first describe the implementation of ClusterGOP, including the system architecture and the runtime library. Then we discuss the MPMD support in ClusterGOP.

5.1 System Architecture

As shown in Figure 5.1, the ClusterGOP system architecture is divided into three layers: the programming layer, the compilation layer and the execution layer.

In the programming layer the programmer develops a ClusterGOP program by using high-level abstraction in message-passing implementation. ClusterGOP exports a set of API that provides the implementation of the parallel applications with traditional programming languages, e.g., the C language. The ClusterGOP API contains a header file of global information, which is shared among the ClusterGOP library routines.

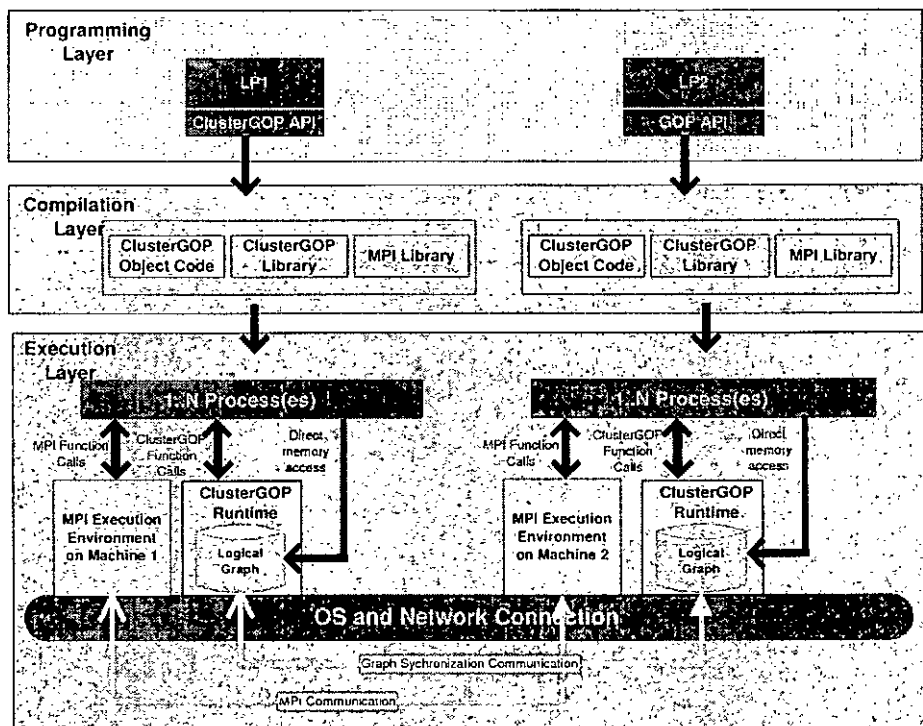


Figure 5.1: The ClusterGOP Program Communication Implementation

In the compilation layer, LPs will be transformed into an executable program on the target's execution environment. LPs are compiled together with the MPI and the ClusterGOP libraries.

The execution layer is realized through the services of two important runtimes, MPI runtime and ClusterGOP runtime. The MPI runtime is the environment used for communication through out the network. The ClusterGOP runtime is developed as a daemon process on top of the Operating System. It helps the ClusterGOP processes to dynamically resolve the node names into process IDs and to prepare for MPI communication. It also provides synchronization among all the nodes, so that each node can get the most updated logical graph for running ClusterGOP primitives. The ClusterGOP runtime system is implemented using the C language with the socket communication and synchronization scheme. In the ClusterGOP runtime system, a logical graph is used for supporting the GOP primitive operations. The ClusterGOP runtime lies on each node so that the nodes can exchange the graph information in a synchronized way. Nodes in the same machine use the shared memory to access the graph. Nodes on different machines require a memory coherence protocol in order to synchronize graph updates. We choose the Sequential Consistency model as the graph synchronization scheme [42].

5.2 ClusterGOP Runtime Library

The ClusterGOP Runtime Library consists of a set of basic communication operations based on the GOP model. This section first describes the basic structure in the library, and then describes the implementation of each library functions.

5.2.1 Basic Library Structure

When started, the LP program invokes the routines `init` and `finalize`

```
Init(argc, argv); /* start ClusterGOP */
```

In MPI, the programmer needs to add additional statements for program initialization, as shown below:

```
MPI_Init(&argc, &argv); /* starts MPI */
MPI_Comm_rank(MPI_COMM_WORLD, &k); /* get current process id */
MPI_Comm_size(MPI_COMM_WORLD, &p); /* get # procs from env */
```

To understand how ClusterGOP hides the details from the programmer, let us have a look inside the initialization code in GOP. The routine `Init` performs several operations, including obtaining the command line arguments, determining the number of processes in MPI environment, getting its own process ID, getting the processor name and reading initialize the graph representation. The implementation is shown below:

```
void Init(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
}
```

```

MPI_Get_processor_name(processor_name,&namelen);
/* init graph node ID mapping */
Graph_Init();
}

```

The routine `Graph_Init` is invoked to read the logical graph into the ClusterGOP system during runtime. Logical graph is stored in an XML format, ClusterGOP uses the Expat (XML parser library) to extract the XML structure and convert the data into program structure. In the following routine, the `XML_Parser` variable is initialized for preparing the XML parsing process, and then the logical graph is input as the source of parsing data. The routines `XML_SetElementHandler` and `XML_SetCharacterDataHandler` are the handlers of start/end tags and handler of text respectively. After that, the XML parsing begins by submitting the parsing data into the routine `XML_Parse` continuously.

```

int graph_xml_init() {
    FILE *fp;
    XML_Parser p = XML_ParserCreate(NULL);

    fp = fopen("graph.xml", "r");
    if ( fp == NULL ) {
        fprintf(stderr, "error in reading graph xml\n");
        exit(-1);
    }

    if (! p) {
        fprintf(stderr, "Couldn't allocate memory for parser\n");
        exit(-1);
    }

    XML_SetElementHandler(p, start, end);
    XML_SetCharacterDataHandler(p, char_hndl);

    for (;;) {

```

```

int done;
int len;

len = fread(Buff, 1, BUFFSIZE, fp);
if (ferror(stdin)) {
    fprintf(stderr, "Read error\n");
    return -1;
}
done = feof(fp);

if (! XML_Parse(p, Buff, len, done)) {
    fprintf(stderr, "Parse error at line %d:\n%s\n",
        XML_GetCurrentLineNumber(p),
        XML_ErrorString(XML_GetErrorCode(p)));
    return -1;
}

if (done) break;
}
return 0;
}

```

In the finalizing routine, just like MPI, ClusterGOP releases the resources used by the ClusterGOP library such as graph representation and temporary variables, and then calls the routine `MPI_Finalize` to end the MPI process.

```

void Finalize() {
    free_Graph(); /* release the memory of ClusterGOP graph */
    MPI_Finalize();
}

```

5.2.2 ClusterGOP Primitives

ClusterGOP primitives are built based on the MPI, so that calling them actually invokes the MPI operations. In order to get the MPI process ID from the ClusterGOP node, a conversion between ClusterGOP nodes to MPI process IDs is processed inside

the ClusterGOP communication primitives, as shown below in implementing routine `GetNodeID` as an example:

```
int GetNodeID(Graph gname, Node gn) {
    int i=0;
    if ( (gname == NULL) || (gn == NULL) ) return -1;
    nodes = get_graph_nodes(gname);
    /* Here shows a simple method for searching the node,
       */
    /* a more efficient algorithm will be replaced if necessary */
    for (i=0; i<node_count; i++) {
        if (strcmp(gn, nodes[i].name) == 0)
            return i;
    }
    return -1;
}
```

The following code segment is the implementation of point-to-point primitives: routine `Usend` and `Urecv`. The code shows how the parameters are translated from ClusterGOP to MPI.

```
int Usend(Graph gname, Node nodename, Msg msg, CommMode mode) {
    int nodeid; /* MPI process ID */
    int status; /* return status */

    /* resolve process ID from ClusterGOP runtime */
    nodeid = GetNodeID(gname, nodename);
    if (nodeid==-1) return -1;

    if (mode == SYN) { /* synchronous send */
        status = MPI_Ssend(msg.data, msg.length, msg.datatype, nodeid, msg.tag,
            MPI_COMM_WORLD);
        if (status == MPI_SUCCESS) return 0;
    } else if (mode == ASYN) { /* asynchronous send */
        status = MPI_Send(msg.data, msg.length, msg.datatype, nodeid, msg.tag,
            MPI_COMM_WORLD);
        if (status == MPI_SUCCESS) return 1;
    }
    return -1;
}
```

```

int Urecv(Graph gname, Node nodename, Msg msg) {
    int nodeid; /* MPI process ID */
    int status; /* return status */

    /* resolve process ID from ClusterGOP runtime */
    nodeid = GetNodeID(gname, nodename);
    if (nodeid==-1) return -1;

    status = MPI_Recv(msg, size, datatype, nodeid, tag, MPI_COMM_WORLD, &status);
    if (status == MPI_SUCCESS)
        return 0;
    else
        return -1;
}

```

The implementation of the collective primitives is similar to the point-to-point primitives, except they use NodeGroup to represent a list of nodes that are required for communication. Therefore, ClusterGOP invoke the function createGroup to resolve the NodeGroup into nodes. The detailed description of NodeGroup will be introduced in the next section. The following code segment presents the routine GOP_Gather of the collective primitives.

```

int GOP_Gather (Graph gname, NodeGroup ng, Msg msg, Node s)
    int status /* return status */
    /* create the MPI communicator from the NodeGroup */
    createGroup(gname, ng);

    status = MPI_Gather(msg.data, msg.length, msg.datatype,
                        msg.recv_data, msg.recv_length, msg.recv_datatype,
                        s, ng.comm);
    if (status == MPI_SUCCESS)
        return 0;
    else
        return -1;
}

```

5.3 Implementing the MPMD model in ClusterGOP

With the MPMD programming model under ClusterGOP, each LP is associated with separate source code. Data can be distributed and exchanged among the LPs. ClusterGOP also has a better node group management than MPI so that the processes can form groups easily and efficiently. In this section, we describe the methodology for MPMD programming support in the ClusterGOP environment. We focus on the new features added to VisualGOP to support high-level MPMD programming, including forming process groups, data distribution among the processes and deployment of processes for execution. With these new features, programmers can program group communication by using NodeGroup, manage distributed tasks and processors through visual interface, map resources to tasks, and compile/execute programs automatically. The underlying implementation using MPI is hidden from the programmer.

5.3.1 NodeGroup Implementation

ClusterGOP's NodeGroup is implemented using MPI's communicator. The basic functions of a communicator include managing processes, defining scope of process communication, and communication between communicators. When two processes do not belong to the same communicator, they cannot send or receive information

from each other. When the parallel application starts, a default communicator, namely `MPI_COMM_WORLD`, is created. By default, all processes belong to the `MPI_COMM_WORLD` and can communicate with each other. Programmers can create new communicators in addition to `MPI_COMM_WORLD`. However, MPI does not provide an easy way to create a new communicator. The example below shows how MPI creates groups and communicators inside the program.

```
MPI_Group MPI_GROUP_WORLD, first_row_group;
MPI_Comm first_row_comm;
int row_size;
int* process_ranks;

process_ranks = (int*) malloc (q*sizeof(int));
for (proc=0; proc<q; proc++)
    process_ranks[proc] = proc;
/* create the group from all processes */
MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
/* select only the process by including the rank no. in the process ranks */
MPI_Group_incl(MPI_GROUP_WORLD,q,process_ranks,
               &first_row_group);
/* create a new communicator according to the group */
MPI_Comm_Create(MPI_COMM_WORLD, first_row_group,
               &first_row_comm);
```

To introduce a new communicator into the application, MPI requires that an MPI group be created to store the neighbors in an array of rank ID. Therefore, the programmer needs to write the corresponding code, remembering every rank ID in the new communicator. This decreases the readability of the program and increases its complexity. In contrast, `NodeGroup` simplifies the procedure for building the communication group and provides better handling of group communication. The following code segment shows the creation of the `NodeGroup`.

```

NodeGroup InitNodeGroup(Graph gname, char *group_name, Nodenodes[]) {
    int i=0;
    NodeGroup group;
    strcpy(group.name, group_name);
    while (nodes[i] != NULL) {
        group.nodes[i] = nodes[i];
        group.nodeids[i] = GetNodeID(gname, nodes[i]);
        group.size++;
        i++;
    }
    return group;
}

int getNodeID(Node gn) { int rank_id=0; /* This variable stores
the rank ID */
    for (rank_id=0; rank_id<global_node_count; rank_id++)
        if (strcmp(gn, nodes[rank_id].name) == 0) return rank_id;
    return -1;
}

```

InitNodeGroup is the API for programmer to initialize the NodeGroup from a list of nodes. The function stores the name and rank ID of the each node into the NodeGroup data structure. For getting the rank ID, the function simply invokes GetNodeID to retrieve it.

The programmer can add, remove or clear the nodes in the NodeGroup by invoking the corresponding API functions. The implementation of function AddNode is shown below.

```

AddNode(NodeGroup *ng, Node s) {
    int i=ng->size;

    strcpy(ng->nodes[i], s);
    ng->nodeids[i] = getNodeID(s);
    ng->size++;
}

```

After a NodeGroup is created, the function createGroup generates the MPI_Group

object (group_world) by providing the NodeGroup information to MPI function MPI_Group_incl. Then, the MPI_Group object is passed to the function MPI_Comm_create to form a new MPI communicator.

```
int createGroup(Graph gname, NodeGroup ng) {
    MPI_Group group_world, new_group;
    MPI_Comm new_comm;

    MPI_Comm_group(MPI_COMM_WORLD, &group_world);
    MPI_Group_incl(group_world, ng.size, ng.nodeids, &new_group);
    MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
    return new_comm;
}
```

5.3.2 Support for Data Distribution

In the MPMD programming model, tasks have different programs to execute but usually need to exchange or share some data. MPI provides API functions for distributing data to different processes, but the programmer still has to write the code for the data distribution procedure. In ClusterGOP, tasks share data by keeping a portion of the global memory in each node that is involved in the communication. The node can update the memory without having to communicate with other nodes.

Using VisualGOP, data distribution can be performed by the programmer through the visual interface. The distributed shared memory can be created by selecting an option in the program editor of VisualGOP as shown in Figure 5.2. Currently, there are three options of the distributed memory styles: vertical, horizontal and square memory distribution. By default, the memory is distributed to the nodes in a

balanced way such that each node will almost share the same size of the distributed data object. VisualGOP also provides a visual interface to allow the programmer to manually specify the memory distribution on each node.

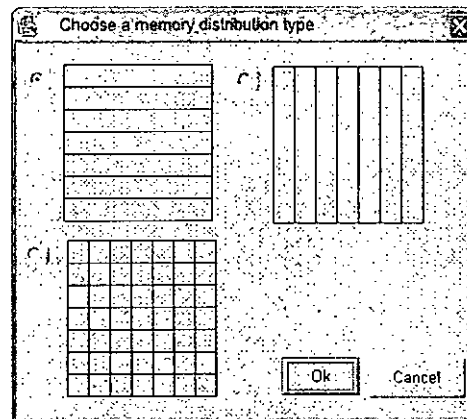


Figure 5.2: Choosing Memory Distribution Type

In many DSM systems, the distributed memory object is built-in function and most of the objects are distributed in the whole environment. However, due to its complex design nature, overheads occur frequently which reduce its efficiency. ClusterGOP implements the distributed shared memory in a different way that distributed objects are used only if programmer explicitly demands them.

In ClusterGOP, the programmer can use predefined data distribution algorithms to put the distributed objects into different parts of the program. VisualGOP also provides a visual way for the programmers to define the data distribution on tasks. ClusterGOP translates the data distribution specified by the programmer into MPI code. For example, when the programmer wants to distribute the object among

several tasks, he/she can insert a distributed data object, and then select which nodes (or NodeGroup) they want to share the object and define the sharing rule. All these can be done with the aid of VisualGOP.

ClusterGOP implements the distributed shared memory functions by using the GA toolkit, which is based on MPI for communications. Before compiling the application, all distributed objects are converted into Global Arrays (GA) codes. GA provides an efficient and portable shared-memory programming interface. The Global Arrays installs itself on top of the MPI library, allowing programmers to take advantage of the interfaces in the MPI and Global Arrays in the same program.

5.3.3 Automatic compilation and execution support

In MPMD programming, managing the mapping of nodes (tasks) to processors could be a complicated task. VisualGOP provides the programmer support to visually map LPs to nodes and nodes to processors. It also provides support for automatic compilation and execution of the applications. This facilitates the development process and simplifies the running of the large-scale MPMD application.

After the program is written, through VisualGOP, the programmer can send the program source codes to the target processors in the system for compilation. When deploying the application, VisualGOP provides information about the target platform such as the address of the machines, the logical graph representation and the

compilation arguments. The programmer creates a processor list for the parallel environment so that the Nodes-to-Processors mapping can be established. After the application has compiled, VisualGOP allows the programmer to start the application for execution. The programmer can provide some pre-defined input argument or data to the application. The final step is the execution of the root process, and VisualGOP monitors the status of the root process and receives the feedback from it.

A ClusterGOP demon process runs on each target processor to receive the incoming compilation and execution requests and interpret them as system commands.

5.4 ClusterGOP Daemon and Runtime Configuration

Before remote compilation and execution, VisualGOP connects to the remote machines and deploys the logical graph, LPs, compilation and execution information to them. Each machine installs and starts the ClusterGOP daemon process for receiving requests from VisualGOP. The daemon program is implemented by a Java network package, using socket to communicate with VisualGOP.

To start the daemon process, programmer runs the daemon program in the terminal environment using the following command:

```
java GOPServer
```

After the daemon process is started in the local machines, the programmer can

use VisualGOP to connect the local machines. VisualGOP selects the mapped LP and transfers the source code to the local machine by using java file stream. Also, the logical graph will be distributed into each local machine. The source code for transferring LP and logical graph is shown below:

```
String str = br.readLine();
// start to read the file and save to temp file
if ( (str!=null) && (Integer.parseInt(str) > 0) ) {
    int filelen = Integer.parseInt(str);

    str = br.readLine();
    FileOutputStream outtmp = new FileOutputStream(str);
    byte[] tmpbytes = new byte[Integer.parseInt(str)];
    for (int i=0; i<filelen; i++) {
        outtmp.write(br.read());
    }
    // save the file
    outtmp.close();
}
```

Upon receiving the compilation command from VisualGOP, the daemon process issues and executes the compilation command on the local machine automatically, for example:

```
make OBJS=local_program.c
```

There is an OS makefile to standardize the compilation format of LP in each local machine. The OS makefile contains the required libraries, compilation tools' path and configuration. An example of Unix makefile is shown below:

```
# Generated automatically from Makefile.in by configure.
ALL: default
##### User configurable options #####
```

```

SHELL      = /bin/sh
ARCH       = solaris
COMM       = ch_p4
MPIR_HOME  = /usr/comp/mpi
CC         = /usr/comp/mpi/lib/solaris/ch_p4/mpicc
CLINKER    = $(CC)
.....
.....

### End User configurable options ###

CFLAGS     = $(OPTFLAGS) -I../include
CFLAGSMPE  = $(CFLAGS) -I$(MPE_DIR) $(MPE_GRAPH)
CCFLAGS    = $(CFLAGS)
LIBS       = -L../lib -lexpat
.....
.....

### End User configurable options ###

CFLAGS     = $(OPTFLAGS) -I../include
LDFLAGS    =
LIBS       = -L../lib -lexpat
OBJS      =
.....
.....

default: $(EXECS)

all: default

run: ${OBJS} nodeconv.o gopmpi.c mergesort.c $(MPIR_HOME)/include/mpi.h
    $(CC) $(CFLAGS) -o $@ ${OBJS} gopmpi.c mergesort.c nodeconv.o -lm $(LIBS)

nodeconv.o: nodeconv.c nodeconv.h $(MPIR_HOME)/include/mpi.h
    $(CC) $(CFLAGS) $(LDFLAGS) -c -g nodeconv.c -lm $(LIBS)

```

The compilation and execution output will be stored in a file under the remote machine's local directory. The daemon process helps programmer by capturing the compilation messages and sending the messages back to VisualGOP for displaying

the result.

Before executing the ClusterGOP programs, the root process of the application requires a program list to start MPI processes. VisualGOP generates the required information to the remote machine, such as machine name and program path of all LPs in the application. Then the daemon process generates the MPI program list for the root process, as shown below:

```
FileOutputStream outtmp = new FileOutputStream("pgfile1");
PrintWriter pw = new PrintWriter(outtmp);
str = br.readLine();
int processor_num = 0;
// start to read the file and save to MPI program list
if ( (str!=null) && (Integer.parseInt(str) > 0))
    processor_num = Integer.parseInt(str);
int index = 0;
// read each node's program path
for (int i=0; i<processor_num; i++) {
    str = br.readLine();
    String dir = br.readLine();
    if (i > 0)
        index = 1;
    else
        index = 0;    // for processor is the host
    String line = str + " " + index + " " + dir + "/run " + System.getProperty("user.name");
    pw.println(line);
}
```

Below shows an example of the MPI program list:

```
U5x-17x 1 /home/program/node1/run
U5x-17x 0 /home/program/node1/run
U5x-17x 0 /home/program/node1/run
...
```

ClusterGOP runtime will start with the local processes. It is implemented based on MPICH (release version 1.2). MPICH v1.2 follows the MPI 1.2 standard and provides full function for running MPI programs. It is a freely available implementation

that runs on a wide variety of systems, such as Unix, Linux, Windows, etc.

When executing the program, the local machine of the root process uses the follow command:

```
mpirun -np [no_of_nodes] [mpi_program_list]
```

The ClusterGOP runtime and library are written in C language, using the standard C and MPI library. The header file is shown below:

```
#include <stdio.h>
#include "mpi.h"
#include "gopmpi.h"
```

ClusterGOP uses the following MPI library functions to form ClusterGOP library:

```
MPI_Init()
MPI_Comm_size()
MPI_Comm_rank()
MPI_Get_processor_name
MPI_Finalise()
MPI_Barrier()
MPI_Ssend()
MPI_Send()
MPI_Irecv()
MPI_Recv()
MPI_Waitall()
MPI_Bcast()
MPI_Gather()
MPI_Scatter()
MPI_Allgather()
MPI_Alltoall()
MPI_Reduce
MPI_Allreduce()
```

5.5 Summary

This chapter has described the design and implementation of the ClusterGOP system.

The system consists of run-times and libraries, and it has been implemented on MPI.

It provides a high-level programming abstraction (ClusterGOP library) for building parallel applications. Graph-oriented primitives for communications, synchronization and configuration are perceived at the programming-level and conceals the underlying programming activities from the programmer.

The next chapter shows the performance results of three examples. We use the results to compare the performance between ClusterGOP and MPI.

Chapter 6

Example Applications

In the previous chapters, we have introduced two examples which demonstrate the use of GOP in parallel programming applications. They are Finite Difference Method (FDM) and Parallel Matrix Multiplication. This chapter will present the performance of each example and compare them with the original MPI programs. Moreover, another example, Two-Dimensional Fast Fourier Transform (2-D FFT), will also be described. It is an example of the mixture of SPMD and MPMD programming style. The program performance is also mentioned in the example.

6.1 Finite Difference Method

The example of FDM has been described as an SPMD application in Chapter 3, Section 3.4.1. In this section, we present the evaluation results of FDM. In our preliminary experiment we have communications between processes in the parallel application. We first compare the performance of our proposed GOP application

library with those of the MPI library, and then identify the overhead involved in the GOP system.

The experiment uses a 20 processors SGI Origin 2000 machine [35], running on IRIX64 6.5. The release of the IRIX implements the MPI 1.2 standard and all the testing programs are written using C language. We use the example described in Section 3.4 for testing. For the program input, we choose large problem sizes 256×256 and 512×512 .

Execution times were measured in seconds using the routine `MPI_Wtime`. Measurements were made by inserting instructions to start and stop the timers in the program code. To make the result more accurate, the lowest bound value from 10 measurements is chosen. The major differences between the two implementations are that the MPI program must contain an extra routine to calculate the runtime processor ID for each node, while GOP must resolve the node names into process IDs before invoke the communications which is based on MPI.

Figure 6.1 shows the execution times for the core-code of the Finite Difference Method in MPI and GOP system, without considering other factors such as program initialization and finalization. The MPI program performs slightly better than GOP, as GOP must spend more time to manipulating the graph topology and preparing the message data for communications. However, the difference is small and the speedups achieved by the MPI and GOP programs are almost the same. The corresponding

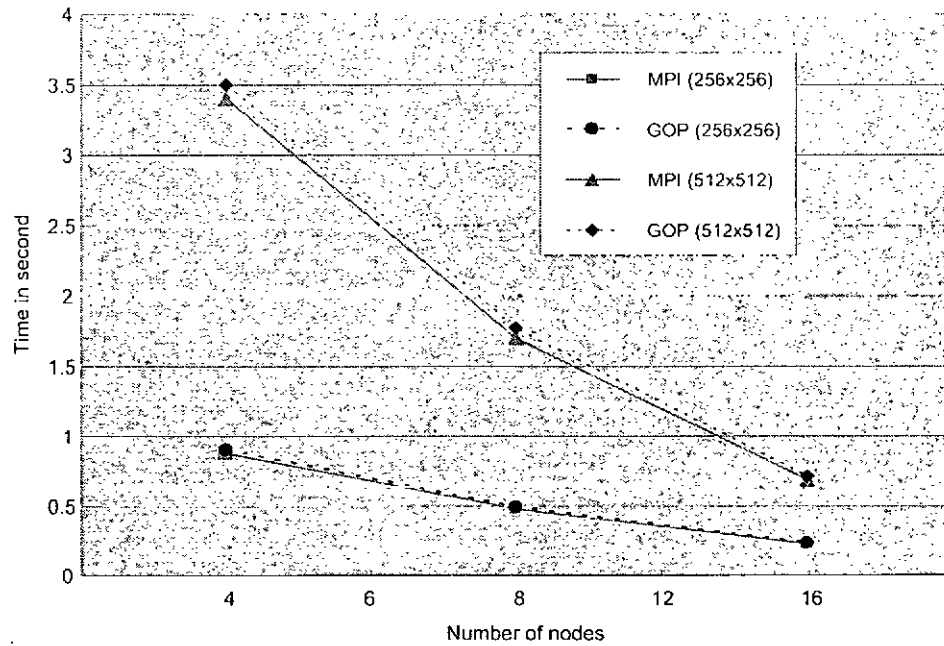


Figure 6.1: Time required by MPI and GOP programs

speedups are shown in Figure 6.2.

We also compared the performance of the MPI and GOP communication routines. Here, we assume MPI does not have any overheads and used it as the baseline for comparison. GOP will spend processing time on converting node names to process IDs and converting the message into arguments of MPI operations. Hence, we use a pair of routines `MPI_Send` and `MPI_Recv` for the MPI Node-to-Node communication setup. On the other hand, we evaluate a pair of sending and receiving operation (routines `Usend` and `Urecv`) in GOP and calculate the overhead.

During the GOP communication, all node names will be translated into process IDs of the mapped processors. The source node invokes routine `Usend` to convert the

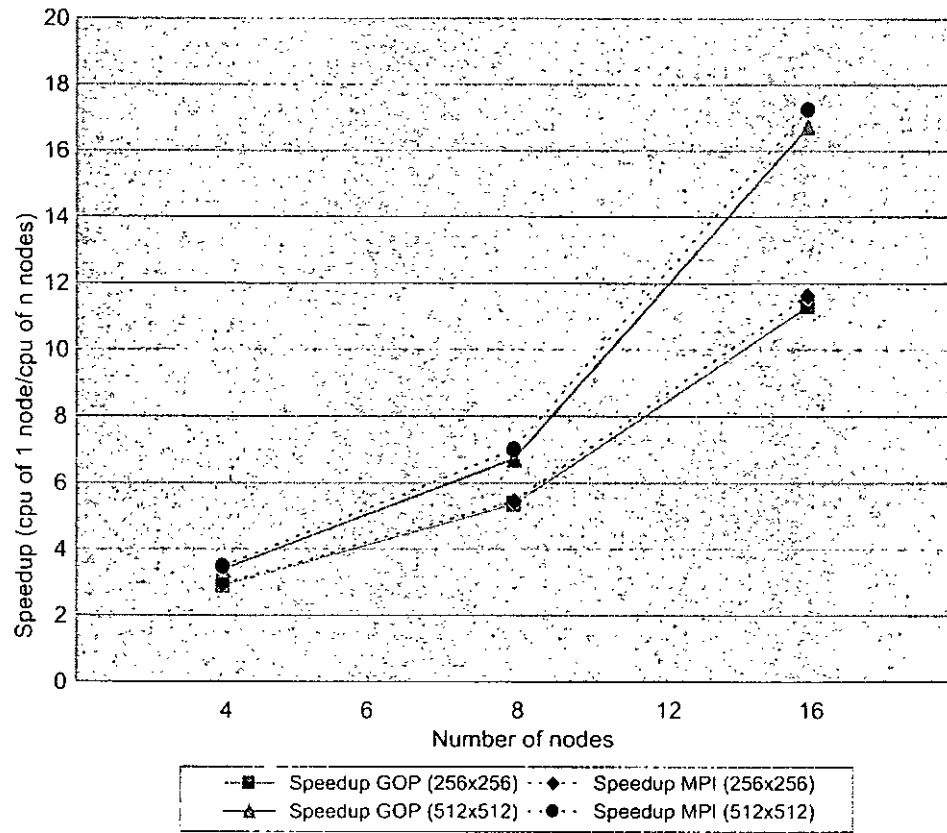


Figure 6.2: Speedups achieved by MPI and GOP programs

GOP message type into input arguments of MPI routine `MPI_Send`, and sends out the message to the destination node. On the other side, the destination node invokes routine `Urecv` and waits for the message from the source node. Once the message has arrived, it will be converted into a GOP message.

Figure 6.3 shows the overhead in handling the GOP message-passing operation. We can see the overhead imposed by this is small compared with using the same procedures under the MPI environment.

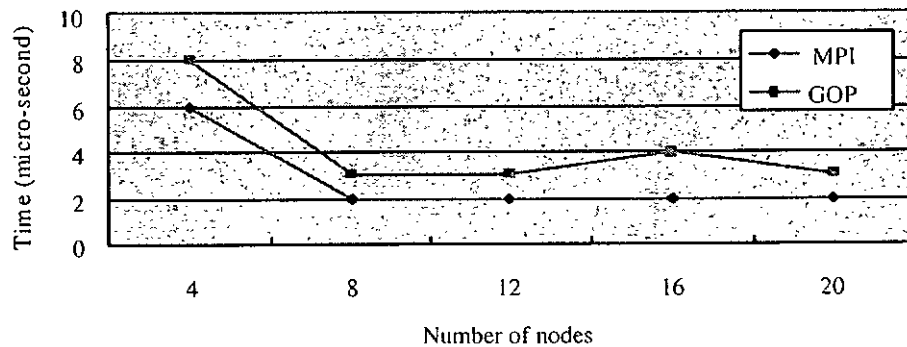


Figure 6.3: Message-passing overhead on GOP

Figure 6.4 shows the program initialization time. Both GOP and MPI require some setup before their library routines can be called. MPI includes an initialization routine `MPI_INIT`. GOP performs the same step as MPI along with the graph initialization. In the graph initialization, each running node reads the graph structure from the graph representation, then it converts the graph structure into its programming structure and loads the graph into the memory. We can see that the initialization

time for both MPI and GOP increases linearly with the number of the processes (nodes). The difference between them is very small.

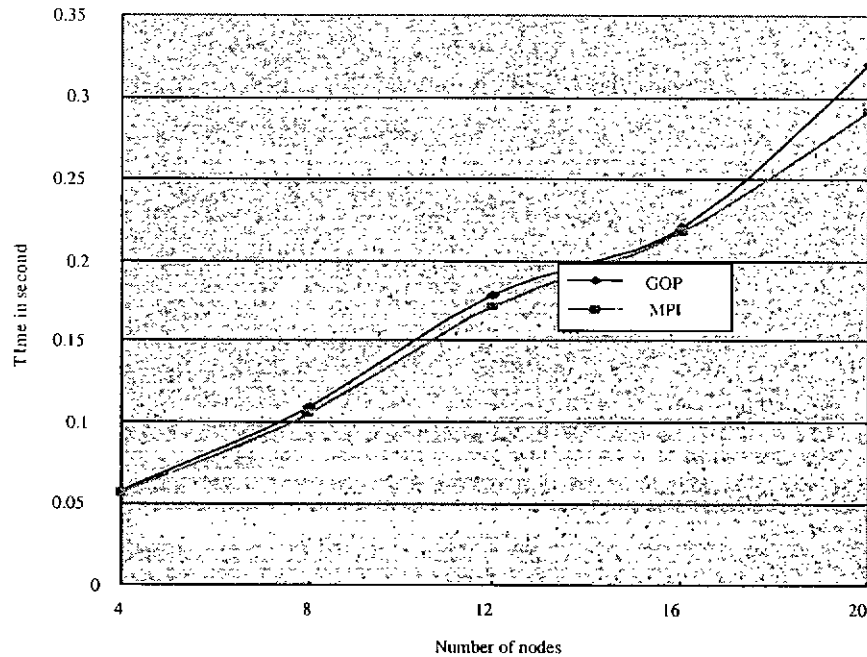


Figure 6.4: Graph initialization time

6.2 Parallel Matrix Multiplication

We have introduced the example of parallel matrix as an MPMD application in Chapter 3, Section 3.4.2. In this section, we compare the performance between the ClusterGOP MPMD and the MPI SPMD programs.

The experiments used a cluster of twenty-five Linux workstations, and each workstation is running on Pentium-4 2GHz. The workstations are setup with MPICH 1.2 and all the testing programs are written in C. Execution times were measured

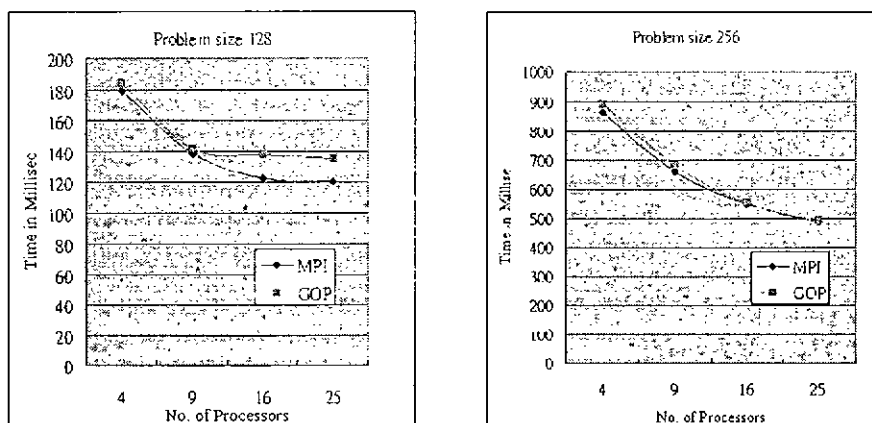


Figure 6.5: Execution time per input array for the parallel matrix multiplication application

in seconds using the function `MPI_Wtime`. Measurements were made by inserting instructions to start and stop the timers in the program code. The execution time of a parallel operation is the greatest amount of time required by all processes to complete the execution. We choose to use the minimum value from ten measurements.

Figure 6.5 shows the performance result in execution time. We can see that the MPI program runs slightly faster than the ClusterGOP program. This may be the result of conversion overheads (nodes to the rank ID) in the ClusterGOP library. However, there are no significant differences between MPI and ClusterGOP when the problem size and processor number are getting larger.

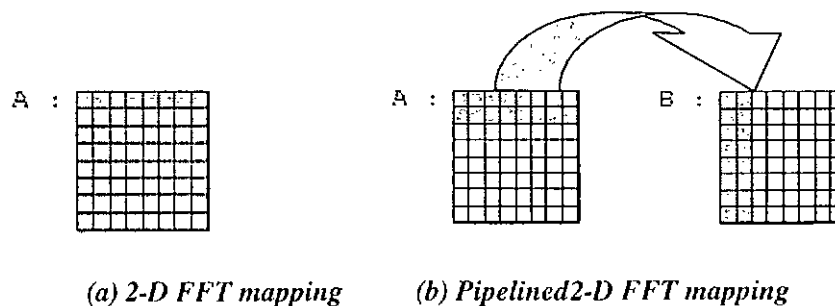


Figure 6.6: Two implementations of a 2-D FFT, the shading area indicates the elements of the array that are mapped to one processor

6.3 Two-Dimensional Fast Fourier Transform

In Figure 6.6a illustrates that the program calculating 2-D FFT first calls the subroutine `rowfft` (row FFT) to apply an one-dimensional (1-D) FFT to each row of the 2-D array A, and then transposes the array and calls `rowfft` again to apply a 1-D FFT to each column of A. The 1-D FFTs performed within `rowfft` are independent of each other and can proceed in parallel. The image data structure needs to be distributed to processors. This distribution allows the calls to the `rowfft` routine to proceed without communication. However, the transposition (or FFT in column) involves all-to-all communication.

An alternative pipelined algorithm is often more efficient (see in Figure 6.6b). The algorithm partitions the FFT computation among the processors such that $\frac{P}{2}$ processors perform the read and the first set of 1-D FFTs, while the other $\frac{P}{2}$ processors perform the second set of 1-D FFTs and the write. At each step, intermediate results are communicated from the first to the second set of processors. These intermediate

results must be transposed on the way; as each processor set has size $\frac{P}{2}$, this requires $\frac{P^2}{4}$ messages. In contrast, the normal 2-D FFT algorithm's all-to-all communication involves $P(P-1)$ messages, communicated by P processors: roughly twice as many per processor. In accordance with the pipelined algorithm, the application is separated into row FFT and column FFT programs. The row FFT can pass the value to column FFT and then continues work for its next data stream. As a result, the network utilization is improved and the application's performance is increased.

Figure 6.7 shows the logical graph of the application represented in VisualGOP. The input node sends data to the row-fft node, which contains four nodes in the subgraph. The col-fft node also contains four nodes. The corresponding programs are mapped to the nodes, and the nodes are mapped to the processors for execution. VisualGOP and ClusterGOP provide several advantages:

- *MPMD representation.* An FFT program is divided into several parts as mentioned above. Each node is associated with a separated program and mapped to a processor to run. There are totally eight processors involved in the computation. Four processors are used for computing the row FFT and others are used for computing the column FFT.
- *NodeGroup formation.* The nodes are classified into two NodeGroups, namely the row FFT group and column FFT group. The two node groups communicate with each other. The row FFT group collects the result and then sends it to

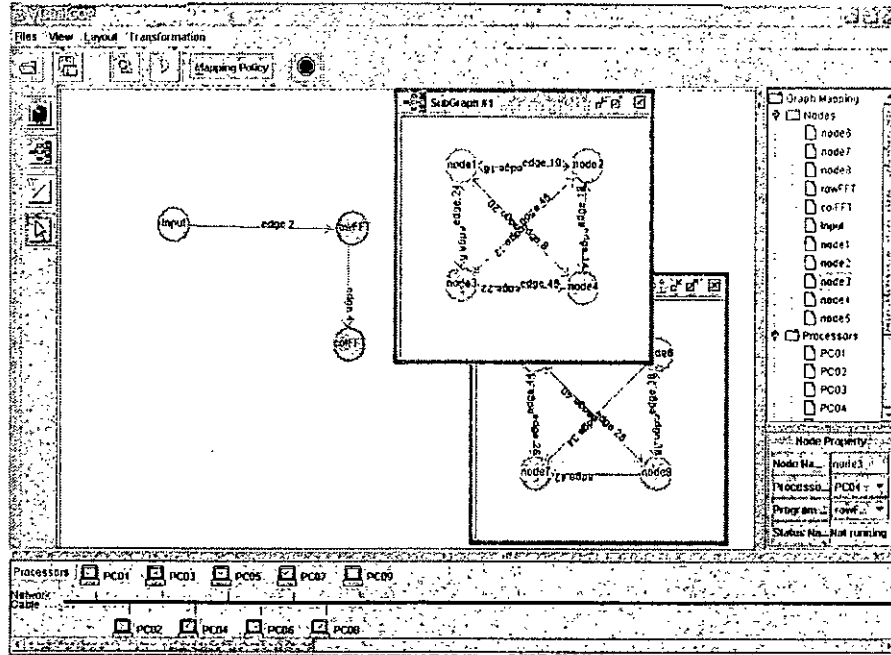


Figure 6.7: Diagram for the 2-D FFT program in VisualGOP

the column FFT group.

- *Data distribution.* To simplify the process communication, a distributed memory object is used in each NodeGroup (row FFT group and column FFT group). It provides a better way to share data between the processes within the same NodeGroup.

In our experiment, the ClusterGOP code is executed as a pipeline of two kinds of tasks in a MPMD model, with an equal number of processors assigned to each task. The MPI code is executed as a single SPMD program. The experiments used a 24-processor SGI Origin 2000 machine, running on IRIX64 6.5, which implements the MPI 1.2 standard. The programs are written using the C language and measurements

were made by inserting instructions to start and stop the timers in the program code. Execution times were measured in seconds using the function `MPI_Wtime`. The execution time of a parallel operation is the greatest amount of time required by all processes to complete execution and we choose to use the smallest value from 10 measurements.

Figure 6.8 presents the results of the experiments, which are performed for various program sizes to render pipeline startup and shutdown costs insignificant. Again, the MPI code is faster than the ClusterGOP code when the problem size is small. This is because ClusterGOP has some overheads in processing graph access and communication. However, this effect is eliminated when the problem size increases. As expected, the ClusterGOP code is faster than the MPI code when the number of processors is getting larger.

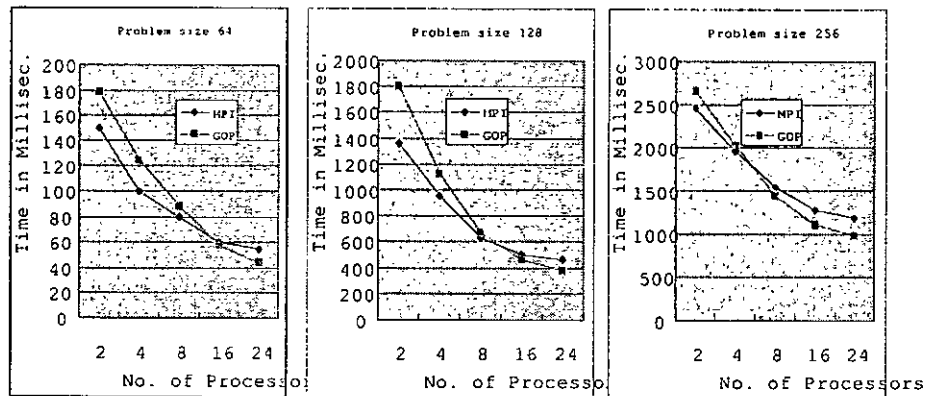


Figure 6.8: Execution time per input array for the 2-D FFT application

6.4 Summary of Results

This chapter has given examples which demonstrate how the GOP concepts described in earlier chapters that allow high-level support for writing parallel programs. The performance result shows that the GOP implementation performs well compared with the low-level programming model (MPI).

Chapter 7

Conclusions and Future Work

In this thesis, we have described a graph-oriented approach to providing high-level abstraction in message-passing parallel programming. The GOP model has the desirable features of expressiveness and simple semantics. It provides high-level abstractions for programming parallel programs and by directly supporting logical graph operations eases the expression of parallelism, configuration, communication and coordination. Furthermore, sequential programming constructs blend smoothly and easily with parallel programming constructs in GOP. We have implemented ClusterGOP based on the GOP model, to support high programming development in clusters. We also provide a visual programming environment, VisualGOP, to provide a visual and interactive way for the programmer to develop and deploy parallel applications. We have described the implementation of the GOP environment and reported the results of the evaluation on how GOP performs to compare with the MPI. The results showed that GOP is as efficient as MPI in parallel programming.

For our future work, we can enhance the current implementation with more programming primitives, such as update and subgraph generation. We can also define commonly used graph types as built-in patterns for popular programming schemes.

Bibliography

- [1] A.L. Beguelin. HeNCE: Graphical development tools for network-based concurrent computing. In *1992 Scalable High Performance Computing Conference*, pages 129–136, Williamsburg, Virginia, 1992.
- [2] A.L. Beguelin and G. Nutt. Visual parallel programming and determinacy: A language specification, an analysis technique, and a programming tool. *Journal of Parallel and Distributed Computing*, 22(2):235–250, August 1994.
- [3] T. Bray, J. Paoli, and C.M. Sperberg-McQueen. *Extensible Markup Language (XML) 1.0*. World Wide Web Consortium, 1998.
- [4] James C. Browne, Jack Dongarra, Syed I. Hyder, Keith Moore, and Peter Newton. Visual programming and parallel computing. Technical Report UT-CS-94-229, University of Texas, 1994.
- [5] James C. Browne, Syed I. Hyder, Jack Dongarra, Keith Moore, and Peter Newton. Visual programming and debugging for parallel computing. *IEEE Parallel and Distributed Technology*, pages 3(1), 75–83, 1995.

- [6] J. Cao, L. Fernando, and K. Zhang. Programming distributed systems based on graphs. In *Intensional Programming*. World Scientific, 1994.
- [7] J. Cao, Y. Liu, L. Xie, B. Mao, and K. Zhang. Portable runtime support for graph-oriented parallel and distributed programming. In *ISPAN'2000 - International Symposium on Architectures, Algorithms, and Networks*, December 2000.
- [8] J. Cao, X. Ma, A. T.S. Chan, and J. Lu. Webgop: A framework for architecting and programming dynamic distributed web applications. In *International Conference on Parallel Processing (ICPP'02)*, Vancouver, British Columbia, Canada, 2002.
- [9] J. Cao, Z.H. Ren, A. T.S. Chan, L. Fang, L. Xie, and D. X. Chen. A formalism for graph-oriented distributed programming. In *Visual Programming - From Theory to Practice*, pages 77–109. Kluwer Academic, 2003.
- [10] F. Chan, J. Cao, and Y. Sun. Graph scaling: A technique for automating program construction and deployment in ClusterGOP. In *The Fifth International Workshop on Advanced Parallel Processing Technologies (APPT03)*, pages 254–264, September 2003.
- [11] F. Chan, J. Cao, and Y. Sun. High-level abstractions for message-passing parallel programming. *Parallel Computing*, 29(11-12):1589–1621, 2003.

- [12] K. Chandy and C. Kesselman. Compositional c++: Compositional parallel programming. In *Proceedings of 6th International Workshop in Languages and Compilers for Parallel Computing*, pages 124–144, 1993.
- [13] C. Chang, G. Czajkowski, and T. Von Eicken. MRPC: A high performance RPC system for MPMD parallel computing. *Software - Practice and Experience*, 29(1):43–66, 1999.
- [14] C. Chang, G. Czajkowski, T. Von Eicken, and C. Kesselman. Evaluating the performance limitations of MPMD communication. In *Proceedings of ACM/IEEE Supercomputing*, November 1997.
- [15] Clark Cooper, Fred Drake, and Paul Prescod. Expat XML parser. <http://expat.sourceforge.net/>, 2000.
- [16] J.Y. Cotronis. Message-passing program development by ensemble. In *PVM/MPI 97*, pages 242–249, 1997.
- [17] J.Y. Cotronis. Developing message-passing applications on MPICH under ensemble. In *PVM/MPI 98*, pages 145–152, 1998.
- [18] J.Y. Cotronis. Modular mpi components and the composition of grid applications. In *Proceedings of the 10th. Euromicro Workshop on Parallel, Distributed and Network-Based Processing 2002*, pages 154–161, 2002.

- [19] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumeta, T. Von Eicken, and K. Yelick. Parallel programming in split-c. In *Proceedings of ACM/IEEE Supercomputing*, pages 262–273, 1993.
- [20] M. Evangelist, V.Y. Shen, I. Forman, and M.L. Graf. Using raddle to design distributed systems. In *10th International Conference on Software Engineering*, pages 102–111, 1988.
- [21] Colin J. Fidge, Peter Kearney, and Mark Utting. A formal method for building concurrent real-time software. *IEEE Software*, 14(2):99–106, 1997.
- [22] I. Foster and K. Chandy. Fortran m: A language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 26(1):24–35, 1995.
- [23] I. Foster, C. Kesselman, and S. Tuecke. The nexus approach for integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, 1996.
- [24] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V.S. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, USA, 1994.
- [25] A. Grimshaw. An introduction to parallel object-oriented programming with mentat. Technical Report 91-07, University of Virginia, July 1991.

- [26] P.E. Hadjidoukas, E.D. Polychronopoulos, and T.S. Papatheodorou. Integrating mpi and nanothreads programming model. In *Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-Based Processing 2002*, pages 309–316, 2002.
- [27] Ivan Herman and M. Scott Marshall. GraphXML - an XML-based graph description format. In *Graph Drawing 2000*, pages 52–62, 2001.
- [28] R. Holt, A. Winter, and A. Schurr. GXL: Toward a standard exchange format. In *7th Working Conference on Reverse Engineering*, Brisbane, Australia, November 2000. IEEE CS Press.
- [29] C. Hu, H. Lu, A. Cox, and W. Zwaenepoel. OpenMP for networks of SMPs. *Journal of Parallel and Distributed Computing*, 60(12):1512–1530, 2000.
- [30] C. Hui and S. Chanson. Allocating task interaction graphs to processors in heterogeneous networks. *IEEE Transactions on Parallel and Distributed Systems*, 8(9):908–925, 1997.
- [31] I.E. Jelly and I. Gorton. Software engineering for parallel systems. *Information and Software Technology*, 36(7):381V397, July 1994.
- [32] K. Johnson, M. Kaashoek, and D. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, volume 29, pages 213–226, 1995.

- [33] P. Kacsuk, G. Dzsá, and T. Fadgyas. A graphical programming environment for message passing programs. In *2nd Int'l Workshop on Software Engineering for Parallel and Distributed Systems (PDSE'97)*, pages 210–219, Boston, 1997.
- [34] K. Kazemi and C. S. McDonald. Process topologies for the parallel virtual machine. *Australian Computer Science Communications*, pages 20(1):43–56, 1998.
- [35] J. Laudon and D. Lenoski. The SGI origin: A ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97), Boulder, Colorado, USA*, volume 25 of *ACM SIGARCH Computer Architecture News*, pages 241–251. ACM Press, 1997.
- [36] Frank Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1991.
- [37] J. Magee, N. Dulay, and J. Kramer. A constructive development environment for parallel and distributed programs. In *International Workshop on Configurable Distributed Systems*, Pittsburg, USA, March 1994.
- [38] P. Newton. *A Graphical Retargetable Parallel Programming Environment and its Efficient Implementation*. PhD thesis, University of Texas at Austin, Dept. of Comp. Sci., 1993.
- [39] P. Newton and J. Dongarra. Overview of VPE: A visual environment for message-passing. In *Proceedings of the 4th Heterogeneous Computing Workshop*, 1995.

- [40] K. Ng, J. Kramer, J. Magee, and N. Dulay. A visual approach to distributed programming. *Tools and Environments for Parallel and Distributed Systems*, pages 7–31, February 1996.
- [41] J. Nieplocha, RJ Harrison, and RJ Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2):197–220, 1996.
- [42] C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th annual international symposium on Computer architecture*, pages 234–243. ACM Press, 1987.
- [43] M. A. Senar, A. Ripoll, A. Cortes, and E. Luque. Clustering and reassignment-based mapping strategy for message-passing. In *12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*, pages 415–421, Florida, United States, 1998.
- [44] V.Y. Shen, C. Richter, M.L. Graf, and J.A. Brumfield. VERDI: A visual environment for designing distributed systems. *Journal of Parallel and Distributed Computing*, 9(2):128–137, 1990.
- [45] M. Snir and et al. *MPI: the complete reference*. MIT Press, Cambridge, MA, USA, 1996.

- [46] N. Stankovic and K. Zhang. Visual programming for message-passing systems. *International Journal of Software Engineering and Knowledge Engineering*, 9(4):397–423, 1999.
- [47] N. Stankovic and K. Zhang. A distributed parallel programming framework. *IEEE Transactions on Software Engineering*, 28(5):478–493, May 2002.
- [48] H. Topcuglu, S. Hariri, W. Furmanski, J. Valente, D. Kim I. Ra, Y. Kim, X. Bing, and B. Ye. The software architecture of a virtual distributed computing environment. In *High-Performance Distributed Computing Conference*, pages 40–49, 1997.
- [49] S.J. Turner, W. Cai, and H-K. Tan. Visual programming for parallel processing. In P. Eades and K. Zhang, editors, *Software Visualization*, pages 119–140. World Scientific, 1996.
- [50] G. Wirtz. Graph-based software construction for parallel message-passing programs. *Information and Software Technology*, 37(7):405–412, 1994.
- [51] R. Wolski, C. Anglano, J. Schopf, and F. Berman. Developing heterogeneous application using zoom and HeNCE. In *Heterogeneous Workshop, IPPS*, 1995.
- [52] L. Wood and et al. Document object model (DOM) level 1 specification version 1.0. World Wide Web Consortium, October 1998.