

**A HIERARCHICAL SOFTWARE DEVELOPMENT ENVIRONMENT FOR
PERFORMANCE ORIENTED PARALLEL PROGRAMMING**

by

David A. Feldcamp

B.S.E.E. University of Texas at Austin

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
COMPUTER SCIENCE

We accept this thesis as conforming
to the required standard

The University of British Columbia

December 1992

© David A. Feldcamp, 1992

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

(Signature)

Department of Computer Science
The University of British Columbia
Vancouver, Canada

Date April 13, 1993

Abstract

The future of high performance computing lies in massively parallel computers. In order to create software to utilize this ever more powerful and complex hardware, software designers must reconcile the desire to provide simplifying abstractions with performance requirements. This thesis examines one approach to addressing this problem.

It has been observed that most parallel programs are written so that they possess the structure of one of a relatively small number of paradigms which describe the essence of a virtual machine. It is well known that better performance will be achieved when there is a close match between the structure of a virtual machine and the algorithms of a program built using the machine. We have proposed the skeleton-template-module (STM) as a parameterized, partially implemented virtual machine, corresponding to a particular paradigm. An STM is implemented by an expert system programmer and serves to hide the internal complexity of the virtual machine from application programmers who are only presented with an interface to the missing components and parameters.

To be truly useful a construct such as an STM must be supported by and fully integrated into a viable programming environment. Such a system must support both the construction of STMs and their use to create applications as part of a general programming framework. However, the lessons of existing sequential and parallel programming systems must not be forgotten. "Ease of use" must remain a high priority. Reuse must be facilitated and encouraged wherever possible. Scalability in its several forms must exist. Finally, the system must also recognize that programming is but one task in the development of parallel programs and so must support the existence and cooperative functioning of a variety of tools. Of these tools, the most important from the developer's point of view is the user interface which provides both support for constructing programs and interacting with the rest of the environment.

We have designed and built such an interface and environment (Parsec – the Parallel System for Efficient Computation) along with prototype mapping and loading tools as a follow-on to our original TIPS environment. The system utilizes the process-port-channel model of parallel program description and supports these primitives with a graphical interface. This interface provides arbitrary hierarchical organization within a program by the use of STMs and logical subgraphs within STMs. In addition, objects such as module parameters, process types, process groups, and parameterized graphs are provided to both support STMs as well as meet the objectives of "ease of use," reuse, and scalability.

Table of Contents

Abstract	ii
1 Introduction	
1.1 Motivation	1
1.2 Reconciling Development Complexity and Performance	
1.2.1 Program Design.....	3
1.2.2 Environment Support.....	7
1.3 Objectives	8
1.4 Outline of Thesis	10
2 Related Work	
2.1 Paradigms, Skeletons, and Templates	11
2.2 Software Development Environments	
2.2.1 Similarities.....	14
2.2.2 Tools and Integration.....	16
2.3 Parallel Software Development Tools	
2.3.1 CODE	20
2.3.2 HeNCE.....	21
2.3.3 Paralex	22
2.3.4 Poker	23
2.3.5 ParaGraph	23
2.3.6 Frameworks	24
2.3.7 PIE.....	25
3 Primitives for a Parallel Environment	
3.1 Computation	
3.1.1 Choosing a Primitive	28
3.1.2 Implementation.....	29
3.2 Communication	
3.2.1 Complexity	31
3.2.2 Ports.....	33
3.2.3 Channels	34
3.3 Summary	35

4	Supporting the Process-Port-Channel Primitives	
4.1	Projects	37
4.2	Modules.....	37
4.3	Parameters	40
4.4	Process Types	42
4.5	Process Groups	44
4.6	Parameterized Process Graphs.....	46
4.7	Target Environments	48
4.8	Files.....	48
4.9	Summary	49
5	Interface Issues	
5.1	Base Program Representation	50
5.2	Controlling Display Complexity.....	52
5.3	Interface Levels.....	55
5.4	Auxillary Views.....	56
5.5	Summary	56
6	Implementation and Example	
6.1	Defining Parameters	58
6.2	Defining a Parameterized Graph.....	60
6.3	Defining Process Types.....	63
6.4	Adding Processes.....	65
6.5	Establishing Port Bindings and Other Descriptive Information	65
6.6	Defining Process Creation Information.....	67
6.7	Creating a Reusable Module.....	68
6.8	Instantiating a Reusable Module	69

7 Conclusions

7.1 Synopsis	73
7.2 Current Status	74
7.3 Future Work	
7.3.1 Communication Support and Typing	74
7.3.2 Data Decomposition	75
7.3.3 Derived Objects	76
7.3.4 Advisors	76
References	78

1 Introduction

1.1 Motivation

The future of high performance computing lies in massively parallel computers. The forces of technological evolution clearly point in this direction [Ke92]. Computers with hundreds of processors are being designed and built and systems with thousands of processors will soon follow. However, just as in the past, this rapid and steady march of technology in the hardware domain is being followed at a considerable distance by progress in the software domain which is neither so steady or rapid.

The essential problem facing software designers is how to reconcile two critical, but conflicting, objectives. The first is that software developed for these powerful machines must make reasonably efficient use of their power to justify their existence. There is, after all, little point in using a 100 processor computer to obtain a ten percent performance improvement over a 10 processor system. The second objective requires that it must be reasonably easy to create programs for these machines in order for them to be broadly useful. The obvious conflict is a classic computer science dilemma: abstraction versus performance.

In the sequential domain there is a relatively smooth correlation between increases in abstraction and decreases in performance. As a consequence it has been possible for a wide spectrum of choices to be available to suit almost any requirement. Furthermore, performance improvements in sequential computers have consistently led to significant, albeit not proportional, increases in program performance. In contrast, in the parallel domain the correlation between abstraction and performance is far from smooth. It is not unusual for relatively minute adjustments in a parallel program to produce dramatic speedups.

Furthermore, parallel domain abstractions seeking to provide a level of simplification comparable to that in the sequential domain must hide a much greater underlying diversity. One only has to observe the enormous variety of parallel architectures as compared to that of serial architectures to see the scope of the problem. And, of course, the less correlation between abstractions and reality, the greater the potential for performance degradation.

It is also, of course, possible to take the view that future improvements in parallel hardware performance and design will make such complexities irrelevant. Based on this assumption, the primary problem to solve is that of ease of use. There already exist a number of high-level textual and graphical and textual languages for programming parallel machines (see Chapter 2). Many of these have been implemented, do produce significant simplifications, and in some cases, produce significant speedups. In essence, these systems seek to present a simple, universal virtual machine to the user. The assumption underlying these systems is that a generic code generator can with a few user supplied function create a complete and efficient parallel application.

However, what if the virtual machine implemented by the system is not a good fit for the machine the application is to run on? What if the application algorithms are not a good fit for the virtual machine? Certainly the existing systems create complete applications, but their performance over a wide spectrum of applications is at best highly suspect. Even under the best of circumstances it is fairly optimistic to hope that a generic code generator or compiler will be able to convert an arbitrary application in a generic high level representation into an efficient program on a particular parallel architecture. Creating compilers to perform this task is one of the most difficult in parallel programming [Ke92]. Thus, its not surprising that the best existing parallel compilers are highly specialized to optimize very specific forms of parallelism for particular architectures.

Furthermore, there is no clear concept of precisely what amount of performance is expendable for gains in ease of use. What also must be remembered is that the performance of any software system is rarely uniform. While the worst case performance of a “universal” software system may indeed be unacceptable, it may also perform very well for a more restricted set of applications or platforms. Thus, talking about the quality of performance provided by any given system is not simple.

What is clear is that efficiency is desirable. Software that performs well today is very likely to perform even better on tomorrow's hardware. History has shown that expected

improvements in hardware performance, when they do materialize, are frequently needed just to keep up with the demands of applications. It is unlikely, particularly in domain of high-performance computing, that in the foreseeable future such high performance hardware will be available that future improvement will genuinely be “surplus,” available to be sacrificed for ease of use. Thus, it is important to develop software systems that strive to maximize efficiency while still achieving the goal of limiting development complexity to a reasonable level.

1.2 Reconciling Development Complexity and Performance

1.2.1 Program Design

A significant hint as to how parallel programming can be simplified without sacrificing performance can be seen in one trend in current parallel programming. It has been observed that most parallel programs are written so that they possess the structure of one of a relatively small number of paradigms [AhCaGe91, Al91]. A parallel programming paradigm is a concise strategy for parallelizing a program [Fl79]. Examples include processor farm, divide and conquer, multi-pipelines, and geometric decomposition.

Another useful observation is that these paradigms describe the essence of a virtual machine. Of course, seeking a universal virtual machine for all parallel programming is attractive. Many such machines have been proposed, some still primarily on paper such as the PRAMs and Valiant's model [Va90], others such as Linda [Ca89] already implemented and in use. However, whether any of them will be able to provide generally acceptable performance for all programs is as yet an open issue. Notwithstanding, it is well known that better performance will be achieved when there is a close match between the structure of a virtual machine and the algorithms of a program built using the machine. Even if an acceptable universal machine is found, there will always be a significant community which will not find the relatively modest performance costs associated with it acceptable. These facts in conjunction with the previous observation that programmers of parallel machines currently use a *set* of paradigms suggest that providing a choice of virtual machines from which the best fit can be chosen is both natural and efficient and likely to remain desirable well into the future.

The concept of a skeleton program has long existed in programming contexts where there is a significant amount of overhead involved in creating an application (i.e., code, data

structures, and functionality required for a program to operate in a particular environment). A skeleton program implements the essential structural components required of any non-trivial program in the context. For example, there exist skeleton programs for Macintosh and XWindow applications. Such programs are frequently provided by vendors as programming examples. Another term for this type of program is application shell. The important attribute of such programs is that someone other than the application programmer has provided most of the system dependent infrastructure code in such a form that it needs merely to be fleshed out by the application programmer. The diversity of parallel programming is such, though, that no single skeleton can hope to simultaneously provide a complete framework while at the same time still be general enough to be viable for the entire domain, or even a majority of it [Co89].

A closely related concept is that of programming templates. This is typically seen as a partially coded program implementing a basic framework with annotated gaps left where application specific code can be added to create a complete program (see §2.1 for examples). Frequently some sort of support mechanism is provided for moving between the gaps. While this is not drastically different from the idea of program skeletons, it introduces the idea of presenting an interface for completely specifying a partially implemented program.

A notable contribution to the unification and extension of these concepts is the work of Cole on algorithmic skeletons [Co89]. Cole identifies the following two important characteristics of an algorithmic skeleton:

- it provides only an underlying structure that can be hidden from the user
- it is incomplete.

Much of the incompleteness of a skeleton can be expressed by parameterization. This parameterization need not just be by simple variables such as the number of processors, but also by other factors, such as granularity and topology, which are machine dependent and can have an enormous impact on performance [FeSrWa92].

Consider now an implementation of the virtual machines borrowing from all of these concepts. In deference to its relationship to these concepts we call this object a skeleton-template-module or, more tersely, an STM. Such a construct could be a parameterized, partially implemented program hiding the skeleton structure from the user and only presenting an interface to the missing components and parameters. Such a concept provides a route to satisfying the need to limit development complexity while at the same time not shutting the door on performance concerns. In many respects, this approach to simplifying application

specification provides the best of both worlds – a high degree of freedom and flexibility for optimization to the expert developer of an STM and a quite simple instantiation interface for the application developer. Hiding underlying structure from the user by presenting a simple interface results in programs that are easier to understand and maintain, as well as less prone to error. In particular, the programmer can focus on the computational task at hand rather than the control and coordination of the parallelism. An STM encapsulates the control and communication primitives of the application into a single abstraction.

While communication encapsulation can be made complete in all cases, encapsulation of control is less clear. It has been noted that there are actually two types of control structure in virtual machines: skeletal control and procedural control (e.g., [Epcc92a]). Skeletal control is the case where all control decisions are made by the virtual machine and interaction between it and code that completes and utilizes the virtual machine only involves data. Examples of this case include processor farm, divide and conquer, and geometric decomposition. Procedural control exists in cases where the virtual machine is more flexible and the application can choose an operation to be applied to a set of data and/or the order of operations. A good example of this case is the vector virtual machine. There is not a clear delineation between these cases as is evident if one considers a processor farm where part of the data specifies which of a set of operations is to be performed on the data. We simply maintain that as much control as possible should be maintained inside a virtual machine and, in particular, all control involving parallelism. For example, while an application developer should specify operations and their order for a vector virtual machine it should not be his responsibility to perform ordering optimizations within those constraints or to worry about which instructions are compatible with pipelining.

With respect to performance, the obvious advantage to our approach is that, because of the restricted nature of a paradigm, it is easier to make efficient use of a specific parallel machine. And, because it is derived from a simple strategy, it is easier to provide application developers with a machine-independent, easy to use interface. Since the underlying structure is hidden from the user it can be implemented in an optimized, machine dependent way without impacting the portability of the applications built on top of it. Furthermore, it has been noted even in the sequential domain that programming paradigms are easier to implement and debug than normal, general purpose programs and, in turn, applications built out of these implementations also have a simpler structure and are easier to debug [Za89].

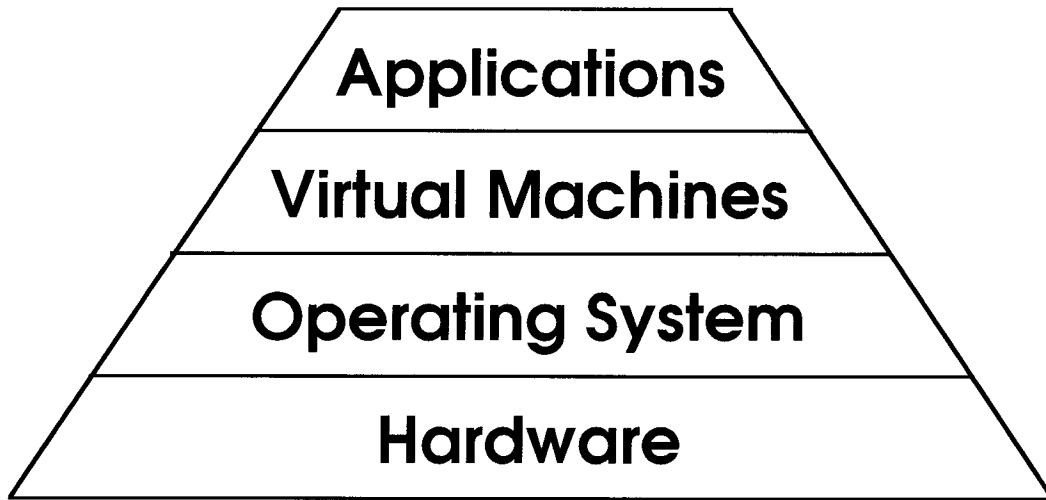


Figure 1.1 Layers of abstraction for parallel program development

Our approach implies that there are four primary layers of abstraction in parallel program development as depicted in Figure 1.1. The top two layers follow the above discussion. The bottom two layers exist in almost any system although the scope, complexity, and portability of the operating system can vary greatly. The nature of programming taking place at our virtual machine and applications layers corresponds respectively to that in the Y and Z layers in Snyder's three layer decomposition of parallel programming [Sn91]. Snyder's X layer corresponds to serial programming.

Ideally the virtual machine layer would be operating system independent to facilitate portability and simplify implementation. This could be achieved by some form of lowest common denominator operating system or interface to operating systems. While some systems (see Chapter 2) have attempted this type of generality, it fails to take into account the significant hardware differences which frequently, but not always, underlie operating system differences. Ignoring these differences means sacrificing performance. Thus, we take the position that the developer of a virtual machine on a particular hardware architecture must have access to a fully featured operating system for accessing the functionality of the architecture in order to insure adequate performance.

1.2.2 Environment Support

Another means of simplifying the construction of efficient parallel programs is to provide support in the form of an integrated environment for program development and execution. As discussed in [ChG091] in the context of our earlier TIPS environment, such an environment incorporating intelligent tools can simplify or take over many tedious and error prone tasks while at the same time facilitating better performance and portability. There are a large number of functionalities which such an environment should support and integrate; for example,

- structural specification
- editing source code
- program compilation
- answering queries about available resources
- mapping (contraction, placement, assignment)
- module loading and execution
- runtime support
- multi-module scheduling
- monitoring
- visualization
- debugging.

This type of environment should not be confused with “programming systems” which provide only a language or those that only provide some form of user interface for program specification (a single component of a full environment).

The potential to incorporate considerable amounts of intelligence into the system exists in several of the functionalities, such as mapping and scheduling. While such intelligence may be considerable, it should never be seen as a replacement for the developer and should always produce results that can be examined and modified by the developer. Given the current levels of algorithm sophistication, in the best cases such tools may remove the need for developer involvement, and in the remainder at least provide the developer a reasonably good starting point.

One of the problems of intelligent tools is that they frequently require large amounts of data to make good decisions. For example, a mapper will need information about resource availability and can make use of statistics about those resources and performance information

from a monitor to avoid creating bottlenecks. Generating and dealing with this data can, of course, be a substantial burden on a developer. However, by integrating all functionalities in a single environment it is possible that this data can be produced, managed, and consumed without developer involvement.

The value of such integration should not be underestimated. As an example, consider the problem noted in the last section of the need for virtual machine developers to directly access the operating system in order to make efficient use of the hardware. This problem commonly arises when there exist several different communications primitives which each have different costs and restrictions. Making the best use of these is essential to obtaining good performance but is a burden to the developer and significantly inhibits flexibility and portability. However, making such optimizations is exactly the type of task a mapper can perform. With access to hardware descriptions and the program specification, a mapper can determine optimized specifics (e.g, physical resources, routing, multiplexing) for any communication. These specifics can be passed to the program when it is loaded and utilized by the runtime system. Of course, given the complexity of the optimization problems in parallel programming, no tool, even when utilizing a spectrum of algorithms, can guarantee to do a perfect job or even an acceptably good job in all cases. Thus, such tools must always provide mechanisms to allow a developer to adjust or even ignore their decisions. Nevertheless, with the bulk of these tasks handled by the environment behind the scenes, a developer is free to focus his attention on programming at a higher, more portable level of primitives provided by the runtime system – *without sacrificing significant performance*.

In addition to the above general functionalities, an environment should reflect, support, and take advantage of programming models such as the one discussed in the previous section. For instance, it could reflect that model by possessing a hierarchy of programming levels and could support it by incorporating mechanisms for declaring, updating, and utilizing STM parameters.

1.3 Objectives

As noted in Chapter 2, the essence of the STM concept, both in parts and as a whole, has been proposed before. There do of course exist comprehensive parallel programming environments and even vague proposals for such environments utilizing programming

paradigms. However, there does not yet exist a system which turns these concepts into a reality. Such a system must support both the construction of STMs and their use to create applications. In order to be relatively flexible such programming should be part of a general programming framework. Such an approach insures that the inevitable variations and exceptions can be accommodated cleanly within the system rather than as tacked on, case by case extensions. Furthermore, building on top of a more general, lower level framework creates the potential for greater consistency and homogeneity within the system.

The system must also recognize that programming is but one task in the development of parallel programs. The system must be an environment which supports the existence and cooperative functioning of a variety of tools which aid the developer in creating, debugging, tuning, and executing a parallel program. Of these tools, the most important from the developer's point of view is the user interface which provides both support for constructing programs and interacting with the rest of the environment.

While the design of this interface must reflect all of these concerns, it also has the responsibility for making the objective of "ease of use" a reality. While there is no simple formula for performing this task, there are a number of issues which must be addressed:

- interaction with the system should be as easy and intuitive as possible
- reuse must be facilitated in order to minimize the amount of actual programming which must be done
- scalability must exist in several forms
 - it must be possible to move with little or no effort from small development versions to massively parallel applications of different sizes to suit the resources of different machines
 - it must be possible to specify complex programs without becoming overwhelmed by that complexity
 - it must be possible to compose existing programs into larger programs with minimal effort

The objective of this thesis is to motivate and describe the design of an environment and user interface which addresses all of these concerns. In doing so, we recognize the need to both share as much as possible in our context with conventional software development environments as well as borrow useful concepts from other parallel environments. Where these sources of inspiration do not suffice, novel approaches or modifications to existing

approaches will be proposed. The initial implementation of this design is discussed where relevant.

1.4 Outline of Thesis

The remainder of this document is structured as follows. Chapter 2 outlines several related topics, briefly surveys a number of related projects, and motivates a number of the design decisions in the following chapters. Chapter 3 discusses the basic parallel programming model which serves as the foundation of our system, and particularly the interface. Chapter 4 describes and motivates a number of objects which have been added to the system to make the basic model more usable. Chapter 5 discusses the basic considerations in the design of the user interface display to support user manipulation of the objects presented in the previous two sections. Chapter 6 presents an example of how the interface may be used to construct an STM while Chapter 7 presents conclusions and a brief view of directions for future work.

2 Related Work

2.1 Paradigms, Skeletons, and Templates

There are two different ways of looking at parallel programming systems. In one the system should provide useful, small set of hopefully simplifying primitives which the author of a program incorporates into a program. The responsibility for structuring and implementing most, if not all, parallel control and communication is still the user's responsibility. Linda [Ca89], Strand [FoOv91], general purpose MIMD 'operating systems' (e.g., Trollius [BuPf88]), and operating system 'interfaces' (e.g., PVM [Su90], CHIMP [Wi91b]) are examples of this approach. The other view, exemplified by our project, is that the system should provide everything possible, such as program organization, control, communication, and performance tuning facilities. The user's only task is to provide those few routines unique to his application such as computation and data generation. Of course, it is impossible for such a system to possess the generality of the first type, but we see the benefits of restricting generality to the level of a single paradigm far outweighing the costs.

Numerous parallel programming systems are available [LeER92] and many of these do attempt to utilize parallel paradigms; however, none of these systems directly integrate the paradigms and their virtual machine implementations into the environment. For example, in some languages the characteristics of the paradigm are often not used by either the compiler or other tools in the environment. There are also compiler optimization tools that do source to source translation of programs for parallel machines, but the underlying abstract model of the machine utilized for the optimization is more general and less accurate than the models we consider [EiB191]. However, an approach similar to ours can be seen in Gelernter's proposal

for integrating the Linda Program Builder into a support environment [AhCaGe91]. Another similar approach can be seen in PIE's [GrSe85, LeSeVr89] implementation machines [RaSeVr90] which are discussed in §2.3.7.

Our approach was originally motivated by the algorithmic skeletons proposed by Cole [Co89]. It differs from and extends Cole in that support from and integration into an environment is included in the definition of our STM. Furthermore, our focus is on performance tuning instead of asymptotic behavior. Like Cole we see the potential for writing highly portable applications using virtual machines. However, we believe that it is not possible to implement virtual machines independently from the underlying hardware and environment and that the quality of the virtual machine ultimately relies on the potential for it to be tuned to the characteristics of the parallel system on which it is to execute. Maximizing this potential while minimizing the user complexity requires that a virtual machine implementation be designed to take advantage of whatever performance information is available with as little assistance from the application developer as possible. Such information may be obtained from a virtual machine specific source such as a parameterized model of its execution or a general mechanism such as a performance monitor. The role of the support environment is to provide the general mechanisms, enable virtual machine specific extensions, and provide means by which they may be integrated with other sources of relevant information such as a mapper.

We do not claim that those systems which only provide primitives are not capable of presenting a virtual machine to a user. For example, there is a project to build on top of CHIMP libraries implementing different virtual machines (e.g., [Epcc92a]). Another example is the Linda Program Builder which provides a specified paradigm in the form of the full Linda coding of the skeleton with marked areas which the user must fill in with application specific code. This code must to some extent employ the Linda language (extensions). In contrast, our user components are kept distinct from system components and are defined so that the user should need to use no communication or other, explicitly parallel, primitives. This contributes to portability since the user must only recompile and relink his components to migrate to a different platform which supports the same skeleton, even though it exists on different hardware and uses a different language, communication technique, and/or algorithm. Providing source code templates to speed and assist user module coding does not compromise this separation.

In addition to the relatively small number of libraries implementing virtual machines, there are a large number of general purpose libraries available for parallel programming. Some, such as those available with Logical C [LC89], provide little more than wrappers for access to the underlying hardware and are, thus, not at all portable. Others are part of a targeted for a particular type of parallel system. These may be part of a relatively complete operating system as in the case of Trollius or part of a system to provide specific functionality as in the case of PICL [HeGi91]. As in the case of Trollius, the more complete systems may provide full access to the hardware in addition to higher level, more flexible, and more performance costly features (e.g., non-local communication). Still other libraries such as PVM and CHIMP have been developed with the specific purpose of providing a uniform multi-platform programming environment on top of platform or architecture specific operating systems. However, it also true that these systems carry a performance cost which varies and do not support all types of architectures, and do not always perform optimal communications (e.g., use a non-local communication primitive for a local communication when faster primitives for local communication are available). Some of the more widely used libraries for parallel programming, such as PICL and Trollius, are supported by monitoring and visualization tools. ParaGraph [HeEt91], for example, allows users to visualize performance information about communication intensive PICL function calls and is also supported as an option in Trollius.

The problem with these general purpose libraries from an application programming point of view is that they are at too low a level, as are, necessarily, the information and visualizations gained by monitoring them. Even with this degree of support there is large amount trial and error involved in tuning a program since there is no direct, obvious correlation between the actual causes of problems and the effects which the user sees. One of the more sophisticated environments at this level is the work of Fox [FoKe91]. This system utilizes experimentally measured “typical” times for all of its primitive operation and uses these to *predict* how long a given block of code will take to execute. Tools at this level help users locally optimize their program, but they do not address the problem of global optimization. In contrast, our use of an virtual machines with analytic models clearly indicates to a user which properties of a program *most* affect performance, perhaps even without running it once. Even when analytic models do not exist for virtual machines, the fact that a virtual machine

implements a restricted, parameterized model of computation serves to simplify and focus potential performance issues.

2.2 Software Development Environments

2.2.1 Similarities

Any environment which supports the software development process can be seen as a software engineering environment. Software engineering environments in the sequential domain have evolved to the point where there is a considerable degree of standardization, at least with respect to functionality and general design principles (see for example [EnWe91]). Since environments for both the sequential and parallel domains share the general purpose of facilitating the building of large, complicated programs with as much generality as possible, it is potentially useful to examine compatibilities between these “standard” features of sequential environments and the requirements for parallel environments.

However, one must be careful when making such an examination. While sequential and parallel environments can be viewed as existing in two different domains, the sequential case can also be viewed as below the parallel case in a hierarchical organization since most, if not all, existing sequential environment functionality is applicable to the design and implementation of the sequential components out of which a parallel/distributed program is constructed. Notwithstanding this connection, the interesting question is to what extent the results of work with sequential environments can be applied to parallel environments. Some of the compatibilities include:

- a small set of primitive building blocks
- a rich set of functionalities supporting the use of the primitive building blocks
- a focus on modularity and reusability
- a basic structural representation – graphs
- a basic implementation structure consisting a graphical interface well integrated with a set of intelligent tools which perform specific functions.

Conventional software engineering environments utilize functions (or some similar unit of code encapsulation) as the basic building blocks out of which programs are built and function invocations as the basic method for connecting these building blocks into larger structures. Most environment operations deal with these objects or aggregates of them.

Keeping their number so small makes the environment simpler and homogeneous. A corresponding set of primitives for a parallel environment are discussed in Chapter 3. Furthermore, just as much of the utility of conventional environments derives from their rich set of features for defining and manipulating functions, a set of features to support parallel primitives is needed to make a parallel environment useful. Such a set of features is discussed in Chapter 4.

A common high priority on modularity and reusability is not at all surprising since both share the objective of developing large, complex programs. However, it should also be noted that the units of modularity will be different, just as are the basic building blocks and the features built on top of them. Also, reuse is capable of taking on a different meaning in the parallel context since much of the size and complexity derives from replication rather than inclusion. Modularity and reusability are guiding motivations in the discussion of Chapter 3 and particularly §4.2 and §4.4. An additional requirement for providing reusable modules, flexibility, is discussed in §4.3 and §4.5-6.

It should also be noted that other high priorities in conventional environments such as supporting large work groups and documentation are still issues in the parallel context. However, their importance is considerably less in the parallel context because the currently envisioned typical user (an individual in a research environment with a task specific application) does not have as great a need for these features as the typical users of sequential environments (groups of programmers in a commercial environment creating applications with long lifespans). The typical user of a parallel environment instead places a very high priority on performance and scalability which are also issues in sequential environments, but not top priorities there.

Conventional software engineering environments rely heavily on graph representations of structural information and graphical displays of this information. Given the importance of structure in parallel programs and the correlation of logical structure to physical structure, such a choice makes even more sense in a parallel context. Graphs are of course widely used in many different contexts as a convenient and understandable representation of structures and their internal dependencies. However, the use of a graph representation does not necessitate the use of a graphical display and interface. For instance, both the Prep-p mapping tool [BrSt89] and the Oregami mapping system [LoRa90] use a textual language to specify a graph representation of a parallel program. It may be argued that such an approach is more powerful

and flexible than a graphical interface approach and is in many cases more terse. Even granting the validity of these arguments, a graphical interface has two surpassing advantages. First, it provides a much easier to understand representation, and, second, it has the potential to provide an intuitive set of manipulation operations instead of the implementation specific syntax and semantics of a particular language. Some of the issues involved in specifying a graphical representation and interface for a parallel environment are discussed in Chapter 5.

2.2.2 Tools and Integration

Conventional software engineering environments must support a number of distinct but related activities. This is also true of environments supporting the development and execution of parallel programs although the number activities is potentially larger, particularly for distributed memory systems. A minimal set of these activities will include:

- structural specification
- editing source code
- program compilation
- answering queries about available resources
- mapping (contraction, placement, assignment)
- program loading and execution.

In addition, support for other activities such as

- multi-program scheduling
- monitoring
- visualization
- debugging

is highly desirable. While some of these activities parallel those in conventional environments, others are not relevant in the sequential domain (e.g., mapping) and others are handled by operating systems (e.g., loading, scheduling). Another difference is that while these activities are typically implemented in a single, general purpose form in the conventional case, in the parallel case many will be implemented as a set of specialized variants (e.g., a set of mapping algorithms which each perform well over a restricted domain). Furthermore, the level at which the activities are carried out will be different in many cases (e.g., debugging a single process vs. debugging a group of communicating processes).

What is common is the recognition of the fact that a significant number of distinct activities must be supported and that any attempt to build them into a single tool would be unwieldy at best. However, even in conventional environments, there are no clear guidelines as to where application boundaries should be. In a parallel environment, this problem is even more complicated because of the existence of additional activities and the fact that the implementation of an activity may consist of a number of domain specific variants.

The early implementations of our project relied heavily on sharing information via files and, later, on combining multiple activities into one tool. However, there are several drawbacks with this approach:

- a base interface for a system as complex as this is itself a large and complex program
- a file interface for a system as complex as this is cumbersome and time consuming to maintain (e.g., establishing file formats, maintaining parsers, managing file locations and names)
- including conceptually independent tools such as the mapper in another program, regardless of design modularity, hinders their development
- a large amount of time and effort is required to create, maintain, and modify complex language data structures and the operations that utilize them.

These observations motivated us to re-examine our design options. Once again, conventional software engineering environments pointed a way out. Better and more sophisticated integration allows greater decomposition in addition to other side benefits. While this is hardly surprising, the concept that the overhead and restrictions implicit in such a decision might be acceptable and worthwhile was not intuitive.

Tool integration is itself a very rich and complicated topic. However, the discussion of it can be simplified by observing that it can be decomposed into a few orthogonal dimensions as illustrated in Figure 2.1 [Wa89]. Of these, clearly the most important with respect to our immediate problems was data integration.

Data Integration

After evaluating the data integration alternatives we chose to utilize a relational database. The prevailing opinion in the software engineering environment community is that these are not powerful enough to handle the real demands of software engineering environments [WiWo89]. Nevertheless, they are still used by many commercial software engineering tools. They are certainly acceptable, at least as an interim solution, because of our relative simplicity (compared to research software engineering environments for sequential systems) and attractive because they are well understood and reliable. Furthermore, we found that the relational model mapped well onto our data representation needs.

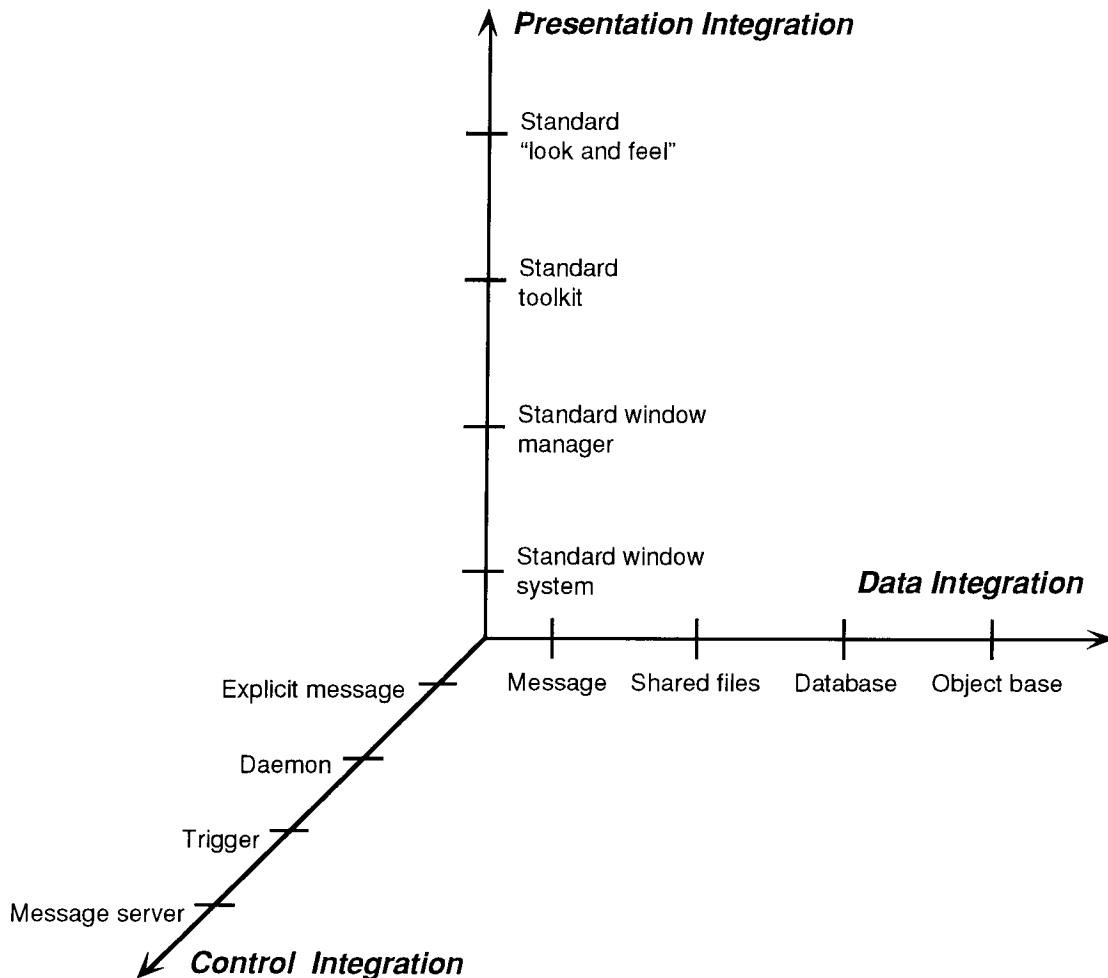


Figure 2.1 Three dimensions of tool integration [Wa89]

Before settling on relational databases a number of options were examined. We considered implementing a simple, custom database system in order to minimize the impact on overall project complexity and performance. However, we ultimately rejected this approach because of the effort required to implement and support such a system and because it would only have distanced us from, but not removed, the maintenance of a file interface. Other existing simple Unix databases such as ndbm and xrdp were considered, but they do not provide any real capability for expressing the structural richness of our data and our diverse query needs. We were unable to consider state of the art object databases (see [Lo89] for examples) because of they are not widely available. Furthermore, they are still a relatively new, and thus unstable, technology.

We initially used the Postgres relational database system [Postgres92], a follow-on to Ingres being developed at U.C. Berkeley, because it is state of the art, public domain, and promises “extended relational” features such as rules and an object oriented approach which should make it better able to address the needs of a software engineering environment than a conventional relational database. However, an initial implementation constructed on this platform encountered several problems. By far the most significant of these was very poor performance, even on relatively fast hardware. Performance problems with relational databases in general have been noted by other parallel interface system developers [SeRu85]. However, the severity of our problems with Postgres can be attributed to its being primarily a research project as well as inefficient programming necessary to avoid bugs and incomplete features. Since the performance was unacceptable for general use a second implementation using an Oracle V6 database was created. Even without performance tuning this implementation has much better performance and is fully acceptable for general use.

Presentation Integration

In contrast to our difficulties with data integration, presentation integration has been a much clearer issue. From the beginning we chose to develop the XWindows based OpenWindows™ environment which implements the OpenLook™ “look and feel” guidelines [OL89]. Choosing such a standard presentation environment is desirable both from a user’s point of view as well as from a design and development point of view. Our decision was

simple because of the ready availability of OpenWindows on our Sun workstations in addition to the availability of a well evolved interface prototyping tool (OpenWindows DevGuide™).

Control Integration

Control integration between our tools is currently very simple because of the tools involved: the interface, mapper, and loader. The mapper and loader programs are currently directly invoked from the interface via Unix process control primitives. However, more complex control interactions may be required for the scheduler which will require repeated interactions with both the mapper and a performance prediction tool to arrive at reasonably optimal scheduling decisions. A more complex, event driven mechanism will certainly be required to support user interaction with debugging and visualization tools.

2.3 Parallel Software Development Tools

There currently exist a number of substantial, relatively general purpose software development environments for parallel systems. Notwithstanding their desire to serve as general purpose tools, most of these systems have a particular focus to their development and tend to treat remaining issues as secondary. A number of the more interesting and relevant systems are discussed briefly in the following sections.

2.3.1 CODE

The primary objective of CODE (Computation Oriented Display Environment) [Br89] is to present a unified, well-defined, comprehensive model for parallel programming. CODE attempts to do this by presenting control flow (dependency) graphs as a language. Instead of utilizing the process as the basic building block for parallel computation CODE uses a “sequential unit of computation” [Br85].

Such a computation unit consists of a firing rule and an action that transforms a set of input dependencies into a set of output dependencies. A computation unit is triggered when data is available for all of its input dependencies. A major limitation of such computation units is that they, at least in theory, do not permit the existence of persistent state which is needed for a significant number of programs (e.g., search, geometric decomposition). Of course, it is possible in practice, with knowledge of the implementation, to find ways to store persistent

implementation at the price of losing portability and violating the model. The developers of CODE see the separation of dependency and firing rule specification from functionality as one way of facilitating reuse.

In addition, CODE provides objects such as merge and distribution filters. These are objects, independent of computation units, which CODE computation units may either send data to or receive data from. A merge filter provides a number of algorithms for merging many input channels into a single channel while a distribution filter provides several algorithms for distributing a single input channel among many output channels. Such filters also provide a means of constructing logic ‘OR’ dependencies.

CODE is targeted to shared memory and tightly coupled MIMD computers and, thus, assumes a homogeneous environment. The main component of CODE is in essence a program specification tool which primarily consists of a graphical interface for constructing dependency graphs, sequential units of computation, their firing rules, and filters. The other main components of CODE are source code generators to convert graphical specifications into compilable conventional languages. The CODE interface does support hierarchical graphs and computation unit or subgraph reuse via its ROPE tool [LeLi88]. However, CODE subgraphs are purely a user convenience for display and manipulation; the system itself deals solely with the flat, complete graph. CODE does not provide any support for special graph constructs such as for loops which are at best awkward when implemented manually and in many cases cause the system to fail. Some support for conditionals in the graph is available via special filters. CODE does provide very limited support for scalability in the form of ‘replication nodes.’ Such a node is essentially a statically resizable vector of completely identical replicas to which a single parent can distribute different pieces of data. All output from replicas must be sent to the same destination.

2.3.2 HeNCE

HeNCE (Heterogeneous Network Computing Environment) [BeDo91a, BeDo91b] is similar in a number of ways to CODE. Both systems require the explicit specification of parallelism within a program and do not facilitate scalability. Furthermore, both utilize a graphical interface to specify a directed control flow (dependency) graph in which a node corresponds to a computation which is triggered when all the input dependencies are available and which, at least formally, may maintain no state information. Unlike CODE, however, the

computation corresponding to a node in the graph may in HeNCE be a parallel computation itself as long as it executes on a single parallel machine. Although the issue is not discussed in their work, such parallel nodes could be used to implement particular paradigms and so correspond to unparameterized STMs.

Unlike CODE, HeNCE supports graph primitives for describing loops, pipelines, and conditionals and so avoids some of CODE's more obvious limitations. Pipelines are a generalization of the CODE's replication nodes and permit any delimited piece of graph to be replicated a specified number of times although they retain the constraint that all replicas must have the same input source and output destination. HeNCE also supports a 'fan' primitive which corresponds to a CODE replication node. HeNCE implements these graph primitives as graph writing operations (e.g., a pair of loop primitives cause the enclosed graph to be replicated the specified number of times when the full control graph is internally generated). However, in spite of its support for conditionals, HeNCE does not provide functionality equivalent to CODE's filters. HeNCE programs are intended to be run on a heterogeneous system. To facilitate this, HeNCE supports the existence of multiply defined, architecture specific node implementations with a cost matrix to help a user select the best for his application and available resources.

HeNCE programs run on a network of heterogeneous systems and are written using PVM [Su90]. In a HeNCE program a tightly coupled parallel machine such as a hypercube or network of transputers is treated as a single machine and, as a consequence, seems primarily applicable to coarse grained applications. Although HeNCE and PVM could conceivably be implemented to work inside tightly coupled systems, no effort has been made to so, perhaps in recognition of the significant differences in resource management, overheads, and performance expectations.

2.3.3 Paralex

Paralex [BaAl91] is a relatively new system which is similar to both CODE and HeNCE. Like both of these systems it utilizes a graphical interface to represent programs as control flow graphs (the developers prefer to view them as coarse grain dataflow graphs), assumes coarse granularity, and provides minimal support for specifying computation units (e.g., assumes only one source file, no libraries). However, unlike the other systems, it provides no support for scalability and neither supports or even permits cycles in its graphs.

Paralex does support filter nodes very similar to those in CODE although those in Paralex are restricted to dividing output data. Like HeNCE, Paralex does not support subgraphs and is solely targeted to distributed systems rather than more tightly coupled architectures.

Paralex is distinct from the other systems described in this section in its support for fault tolerance through replication, dynamic load balancing, and the sophistication of its mapping system. In addition, Paralex supports debugging with monitoring and visualization tools as well as a simple debugger which allows a user to stop execution when one or more nodes is reached, examine or modify node input data, and continue execution. By providing a monitor, visualization tools, a debugger, and a substantial mapper Paralex provides a much more complete system for parallel programming.

2.3.4 Poker

One of the most notable early parallel programming environments with a graphical interface is Poker [Sn84]. Poker was targeted towards a particular class of homogeneous, configurable, tightly coupled MIMD architectures. It provided a relatively simple graphical interface for drawing a program's communication structure. The communications graph is completely static with no provision for supporting the design of scalable software. Additional pieces of information such as the names of source code files are attached as annotations to the communication graph. Poker was one of the earlier parallel programming systems to highlight process and process communication ports as primitives for describing a parallel program. A large amount of the effort invested in Poker was involved in developing tools to map parallel programs onto real hardware architectures or simulations of them. Ultimately these tools for such tasks as describing a process graph, contracting it, placing processes onto processors, and assigning physical resources to communication channels evolved into the Prep-p mapping system [BeSt89].

2.3.5 ParaGraph

The ParaGraph graphical interface [BaCuLo90] is primarily dedicated to supporting the scalable specification of parallel programs. The designers see this task requiring the ability to specify graph families rather than graph instances, to compose graph, to annotate these graphs with labels or information (as in Poker), and to display such graphs in a usable manner. ParaGraph supports the specification of graph families by utilizing aggregate rewrite (AR)

graph grammars which are well suited to interconnection graphs which the authors characterize as being sparse, recursively constructed, nearly symmetrical, and of low radius [BaCu87].

The ParaGraph interface essentially allows a user to specify a base graph and a set of AR productions which both utilize and rewrite graph node attributes in addition to the graph. Construction of graphs which are not easily produced with AR mechanism can be aided by ‘scripts’ which allow the user to control graph generation directly (i.e., specify allowable derivation sequences). Support for composing graphs is not described in detail but relies to some extent on matching attributes for different graphs. Mechanisms for displaying the large graphs this system is capable of producing have not yet been presented. Like Poker, ParaGraph uses processes, ports, and channels as its underlying primitives for describing a parallel program. ParaGraph is currently used in conjunction with a parallel computer simulator and is not designed to support heterogeneous systems.

2.3.6 Frameworks

Frameworks [SiScGr91] is currently the only system which supports any sort of paradigm based approach to parallel programming with a graphical interface. Frameworks is designed and implemented to run on a homogeneous network of workstations in a multi-user context. This system supports two built in variations on a processor farm virtual machine: a ‘manager template’ which only supports static worker allocation and a ‘contractor template’ which supports dynamic worker allocation. The primary purpose of the graphical interface is to create multiple instances of these templates, attaching task specific application code to the templates, and connect the template instances to create a large application. These activities all correspond to our application level of parallel programming (see Figure 1.1).

The graphs created by connecting templates are required to be acyclic and template instances must not require persistent state. Frameworks does provide a typing mechanism for connections between template instances and does perform type checking on these connections. The developers of Frameworks have observed significant performance improvements for coarse grained applications. The developers also claim that the system could be easily adapted to a heterogeneous environment and to more tightly coupled parallel systems. While the implementation would be straightforward, it is not clear how the significant differences in resource management, overheads, and performance expectations would be addressed.

2.3.7 PIE

The PIE system [LeSeVr89] is primarily oriented towards debugging correctness and performance by providing performance monitoring and graphical visualization for both shared and distributed memory programs. PIE also provides graphical displays such as program ‘roadmaps’ and invocation trees to aid user understanding of parallel programs. As with our system, PIE is interested in efficiently mapping parallel application onto specific architectures.

However, with respect to our work, what is particularly notable about PIE is that its early design [GrSe85] called for a system built to support a level of abstraction which they called an ‘implementation machine’ (IM) between algorithm and ‘realization machine’ (architecture) levels. They saw these IMs as corresponding to parameterized parallel programming paradigms. They also noted the potential for simplifying development and, more importantly from their point of view, simplifying debugging of correctness and performance. More recently, in [RaSeVr90] they described the implementation of a dataflow virtual machine for shared memory architectures using precompilers, code templates, and runtime support libraries. This paper also presents worst case performance estimates for the virtual machine and empirically validates their correctness.

The motivation for IMs corresponds very closely that presented for our STMs. The two approaches are similar in a number of ways and their levels of parallel programming abstraction correspond to ours if the OS and hardware levels are seen as both residing in their ‘realization machine’ level. However, like the Edinburgh virtual machine libraries and unlike our system the PIE system provides no environment support for building, instantiating, and maintaining virtual machines. Support to help users create virtual machine applications exists, but is designed and implemented on a case by case basis. There is no attempt to fit these activities into a coherent larger framework. Overall, there is no discussion of support for scalability other than the early mention of the need for IMs to be parameterized. IM level parameters have not yet been described in detail or implemented. Furthermore, while PIE can and does collect performance information on virtual machines, no techniques have been presented for using this information to automatically tune performance.

3 Primitives for a Parallel Environment

All programs constructed with Parsec are ultimately described in terms of a simple, standard, and relatively general model for describing a parallel program [BaCuLo90, Wi91a, Sn84]. This model consists of three primitive types of objects and their relationships. These objects are processes, ports, and channels. Processes possess communication ports which can be bound to one end of a communication channel. Specifying these bindings defines the basic structure of a parallel program. A graphical representation of the process-port-channel primitives and their relationships is presented in Figure 3.1.

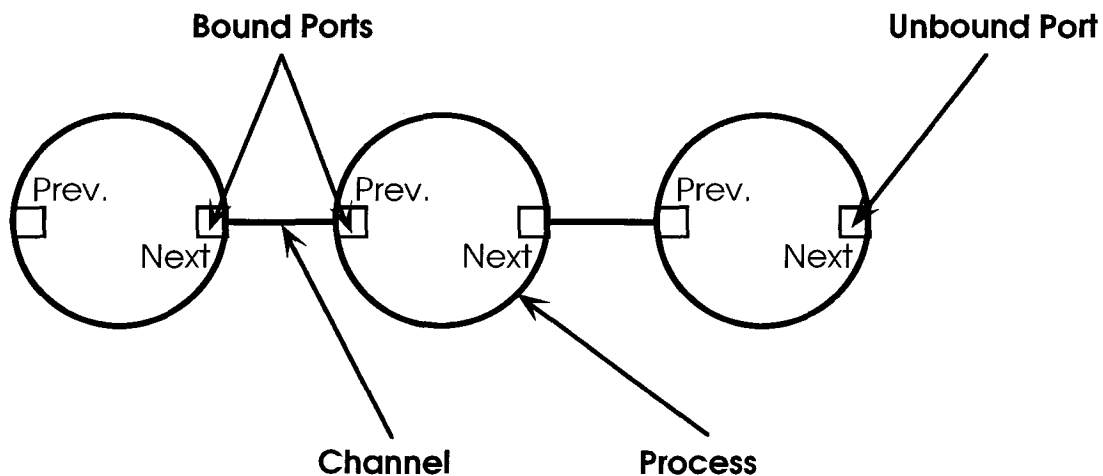


Figure 3.1 Parsec primitives

3.1 Computation

3.1.1 Choosing a Primitive

The process is probably the most widely used type of object in all types of multi-task programming and is the most obvious parallel to the role of the function in sequential environments. Even systems which do not present processes as objects visible to users (e.g., parallel Prolog implementations), almost always employ them internally. Not only is the process a natural unit of decomposition for parallel programming, but hiding it from the user also would mean that environment operations on processes such as mapping and scheduling would also be completely hidden from the user. Furthermore, debugging would become a more difficult problem since many components of a program should either be hidden from the user or mapped from the implementation representation to a user representation.

The main alternative to the process as the basic building block for parallel computation is the “sequential unit of computation” [Br85, Br89]. This object has the useful property of only accepting input communications at its start and producing output communications when it completes. While each such an object can be viewed as a member of a restricted class of processes, CODE makes no claims to the user as to how they will be implemented. Browne’s approach is conceptually attractive since it completely separates computation and communication and essentially preserves the sequential concept of invocation. However, this approach has a number of problems.

While the pleasing qualities of Browne’s model are evident in programs in which data flows from the top to the bottom of a program, its problems become apparent in any case where loops including both computation and communication exist. In the simplest case where there are a single computation and communication and the computation maintains no state information (e.g., a simple server task), the loop can simply be viewed as repeated invocations. However, the model fails in the cases where state must be maintained since no mechanism exists in the model for specifying and preserving this information. One of the proposed solutions to this problem is for a such a unit of computation to send itself a message containing its state, which in turn introduces the problem of initialization. Of course, with knowledge of the underlying implementation, one may use an appropriate means of preserving data.

The real issue is that no guarantees exist at any substantial degree of detail as to how the unit will actually be implemented. While a user may not be interested in whether a particular piece of code is running on a transputer, workstation, or supercomputer, he is likely to be interested in whether it is going to be a distinct process or not, whether that process will exist continually or be repeatedly restarted, etc. While a user may not be interested in or need to know what low level primitives are being used to implement a communication, in many cases it is important to know whether that communication is buffered or not. It is one thing to take away complexity, it is another to simplify by imposing restrictions. It can be argued that such decisions should be left up to a system which will choose optimally, but currently no such systems exist. To perform most types of tuning one must have access to a model that more closely corresponds to the actual running implementation. It is important to remember, however, that not everyone need work at this level – if some expert programmers develop efficient restricted virtual machines which provide a similar, or even simpler, interface than the generic systems both the objectives of ease of use and performance can be satisfied for the authors of applications built on top of these restricted virtual machines.

There are also problems of a more practical nature. If the system automatically translates every unit of computation into a process, then for many reasonable granularities where the units will be equivalent to single functions and in cases trivial functions performance significant performance overheads will be incurred. If the environment groups units of computation into process, then, as noted earlier, there is the question of how a user will debug such a program or interact with tools such as schedulers that must operate on processes. Furthermore, many performance optimizations rely on closely coordinating computation and communications.

3.1.2 Implementation

The process object is, not surprisingly, the most complex of the three primitive objects and a significant number of pieces of information involved in specifying one. Parsec divides this information into two groups – structural information and construction information. Structural information includes such things as process type, ports, port bindings, group membership, weight, and constraints. Construction information includes source files, libraries, compile configuration, and runtime parameter specification. Aside from providing a logical division in a large set of information, this separation makes the high level structure

independent of the actual process instances to the point where the same structure could be reused by changing the construction information for the processes.

3.2 Communication

Once a primitive building block is established, there still remains the problem of how those blocks will be connected to form a larger structure. As in the conventional software engineering case this connection is a form of communication. The use of processes as the building blocks means that the communication is simply a passing of data between existing objects rather than an invocation of one object by another. This certainly has implications for program semantics, but not for program structuring.

The need for reuse does, however, have a significant impact. Within parallel programs reuse takes the form of replication. Unlike distributed operating systems where communications abstractions may hide replicas from the user, replicas must play the same role as other processes in a parallel program. Just as for non-replicated processes, replicas must be bound into a program structure with explicit connections to other processes.

Changing the structure of a parallel program means changing these connections between its processes. However, if a process is to remain replicatable, the internal process specification must remain independent of the process's external connections. It is for this reason that ports are necessary. A port is essentially a named variable specifying a logical destination and whose value specifies a physical destination and other implementation attributes (see §3.2.2). Regardless of the complexity of this information or future extensions, the user need only deal with the name he has assigned to the port.

If ports are not used, then the only way that the process can specify a communication destination is by explicitly naming the destination. Since communications destinations in a program must have unique names, this implies that one would have to change the code for a process each time the objects it communicates with change or, even more frequently, every time the process is replicated. Thus, for example, it would not be possible in a mesh connected system to have a number of identical processes which each communicate with their neighbors via named objects north, south, east, and west. This is, of course, highly undesirable and demands the establishment of an interface internal communication specification from external specification.

Channels are not strictly necessary to establish simple communication relationships in this scheme since ports provide the necessary separation between logical and physical communication. However, when stating that logical port 'north' on process A is physically bound to a destination of port 'south' on process B one implicitly states that there is a connection (or dependency) between the two ports and processes. In graph representation, this relationship is invariably represented as an edge. A channel is the primitive which corresponds to this relationship.

While there is the similar concept of logical dependency in sequential environments, the correspondence of the logical to the physical is usually negligible. However, in the parallel context, the correspondence of this relationship to the physical will vary depending on the context. In distributed memory or point-to-point communication systems, it corresponds directly to a physical route between two processors. As such, a number of important properties, clearly not the property of either end of the connection, may be attached to it (e.g., weight, routing information, protocol). On the other hand, in shared memory or purely broadcast communication systems, the channel may only have its logical value although there may frequently be software options for the communication to be specified (e.g., buffering, protocol, timeouts). Since the logical relationship will exist regardless of how it is treated, raising its status costs so little that it may be conveniently ignored where it is of little use. However, omitting channels would cost generality in those cases where they do have physical parallel.

3.2.1 Complexity

Yet another advantage of ports is that they provide a means of hiding external communications complexity from a process. For example, it is possible to have several processes connected by several channels to a single input port which the process will simply see as a single data stream. Furthermore, the number of connections to the port can be changed with no impact on the process. Conversely, a process can send messages to an output port without regard to how many channels are connected to it and will not be affected if that number changes. The responsibility for dealing with multiple messages coming or going through a single port belongs to the system communication functions which implement the communication primitives (see §3.2.2).

Allowing multiple bindings to ports and channels provides a simple and abstract way of providing a useful and relatively general set of communication primitives: one-to-one, one-to-many (multicast/broadcast/scatter), and many-to-one (gather). Of course, this port functionality can be mostly duplicated by adding significant complexity to the user code for a process (e.g., an iteration over a set of connections to either send a message for a multicast or poll for data). While the process-port-channel model may not be the best fit for all types of systems it can be mapped in a logical, easy to understand manner to such diverse architectures as single processor multi-tasking, parallel shared memory, and MIMD in an efficient manner since it does not require any sophisticated features be present but can take advantage of them where they are present and useful.

Another approach to providing one-to-many and many-to-one communications can be seen in CODE's merge and distribution filters (see §2.3.1). This approach was motivated by CODE's design goal of isolating communication control from computation. It has the advantages of achieving this objective plus eliminating the need for the user to provide any communication control code. However, this approach does not map clearly onto a process oriented model and has the considerable disadvantages of preventing the developer from closely integrating computation with communications since it both compartmentalizes the functionality provides it the form of system utilities which are not intended to be user accessible. This integration is frequently necessary in order to provide optimal performance [SrChWa91].

Fundamental to the choice of the process-port-channel model is the design decision that a program developer should only be provided with a very simple and generic set of communication primitives. This is in contrast to other environments such as Trollius [BuPf88] which provide numerous primitives, each with their own capabilities, limitations, advantages, and disadvantages. While this rich set of choices provides considerable potential for hand optimization of programs, it makes programming more complicated and can make programs very platform and structure dependent (i.e., a program may assume not only that it will be run on a transputer network, but also a transputer network with a very specific interconnection pattern). In contrast, using a small set of generic primitives supports our aims of programming simplicity, at least modest portability, and providing support for automatic communication optimization techniques. Performance is not sacrificed for simplicity and portability since

automatic communication optimization is provided. Note also that this approach does not rule out providing users with the option of specifying constraints on particular communications.

3.2.2 Ports

Ports are the external interface to and from a process and may be used for input, output, or both. Port names must only be unique to a process. They may be bound to one end of one or more channels or left unused. Dynamic binding of ports at runtime is supported by the process-port-channel model and has been implemented in other systems (e.g., [Wi91a]). However, dynamic binding is not currently supported in our implementation since there is no way to optimize communication over such connections with static analysis tools. As a consequence all communications paths are specified as explicit port to port connections in the structural specification.

Processes communicate by either synchronously or asynchronously sending data to or receiving data from a specific port by invoking one of these four functions with the port as an argument:

```
port_send(Port *port, void *data, int size)
port_async_send(Port *port, void *data, int size)
port_recv(Port *port, void *data, int *size)
port_async_recv(Port *port, void *data, int *size)
```

The code for a process contains variables of type `Port` whose names correspond to those of the ports bound to channels in the structural specification. The information to fill these structures is automatically generated by environment tools (primarily the mapper) and automatically passed to the process by the loader as extra ‘command-line’ parameters. These parameters are bound to the port structures of a process by a call to the function

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

when the process is initialized. The loader orders the parameters by searching the source code for a process for this function and matching the function arguments with the names of ports in the specification.

There are currently four parameters: an identifier for the destination node, an identifier for the physical communication link locally associated with the port, a communication event identifier, and a set of flags specifying the implementation type of the connection (multiplexed,

non-multiplexed, neighbor-to-neighbor, multicast, etc.). This set of parameters is specific to our initial target environment of a network of transputers running Trollius. A corresponding, but different set will be needed for different environments. Again, note that these variations are hidden from the user.

This approach is not compatible with many current types of ‘pragmatic’ programming seen in parallel programs. For example, in many programs a process obtains its logical identifier (usually an integer) from the operating system at runtime (e.g., by `getnodeid()` in Trollius) and then establishes communications with a set of processes whose identifiers it derives from its own (e.g., the next and previous). Not only is this a problem because it is an example of dynamic channel binding, but it also includes assumptions about the execution platform and the overall mapping of the program. Such assumptions are not supported by our system since it is not aware of them and, thus, may invalidate them. Furthermore, even if the system were aware of them they would inhibit portability and optimization potential.

3.2.3 Channels

Channels are “pipes” down which data can flow between processes. As such they define the communication pathways within a program. Channels can be either unidirectional or bidirectional and can have one or more ports bound to each end. The implementation of channels is system dependent. It could be message passing over links between distributed memory units, some form of shared memory IPC within a shared memory unit, or sockets on a workstation. As noted in §3.2.1, even within a particular architecture, the operating system layer may provide several sets of communications primitives to choose from based on advice from tools that form part of the environment.

Channels possess properties such as a weight reflecting the traffic density on the channel and constraints. These properties are used by the mapping system to determine the best way to implement the channel as part of an optimal communication strategy. If duplicate connections between processes are not permitted, it is not necessary to name channels like processes and ports since a channel is uniquely identified by the pair of processes it is connected to. If duplicate connections are permitted as in our system, the port to which a channel is connected within each process is required for unique identification. It is attractive to avoid giving channels names since many users will want to ignore them and automatically generated names (e.g., Channel X) would convey no useful information to a user when

binding ports to channels. An intuitive approach for binding ports to channels is to let the user specify a port and select a channel implicitly by specifying the neighbor process with which communication is desired. Such a list of neighbor processes is simple to construct given a list of channels connected to a particular node. Neighbor/port pairs are required if duplicate channels between processes are allowed.

3.3 Summary

This chapter has discussed the process-port-channel primitives utilized by this project as well as some of their features in our implementation. The choice of these primitives was motivated by their flexibility and close correlation to existing implementation environments which both influence performance. The numerous advantages of including ports in the set were also discussed. In addition, the process-port-channel primitives were compared with the unit of computation/data dependency primitives utilized by several other systems. The next chapter will discuss objects built on top of these primitives to facilitate their easy and effective use.

4 Supporting the Process-Port-Channel Primitives

4.1 Projects

A project is the highest level structure in Parsec. It corresponds to the user concept of an application or program and is the root module (§4.2) of a (possibly trivial) tree of module instances. It also has the special properties of existing in a special name space and of owning any miscellaneous global configuration options set by a user.

4.2 Modules

A module is a named group of processes with an external interface. A module may either be an instance of an exported (public) module or an unique instance. In either case a module may include (instances of) other modules. All objects belong to a module or one of its instances although this may be the special “project module” with an empty interface at the top level of a project hierarchy. Thus, modules are the primary mechanism in Parsec for hierarchical organization as well as process group level reuse and information hiding.

While objects with such purposes are common in conventional programming contexts, in the parallel environment context reuse of subgraphs is only supported in CODE via its associated ROPE facility [LeLi88]. However, the subgraph reuse mechanism in CODE/ROPE only makes copies of the data in the database, thus making the propagation of fixes or improvements to the unit difficult. In the context of distributed operating systems there is the related concept of a process cluster, although the members are usually replicas created to

enhance throughput or fault tolerance with perhaps a common manager. In many cases such clusters either present a single public communication interface (e.g., through a manager process) or appear to do so because of a communication system based on one-to-many and many-to-one interactions that are hidden from the user.

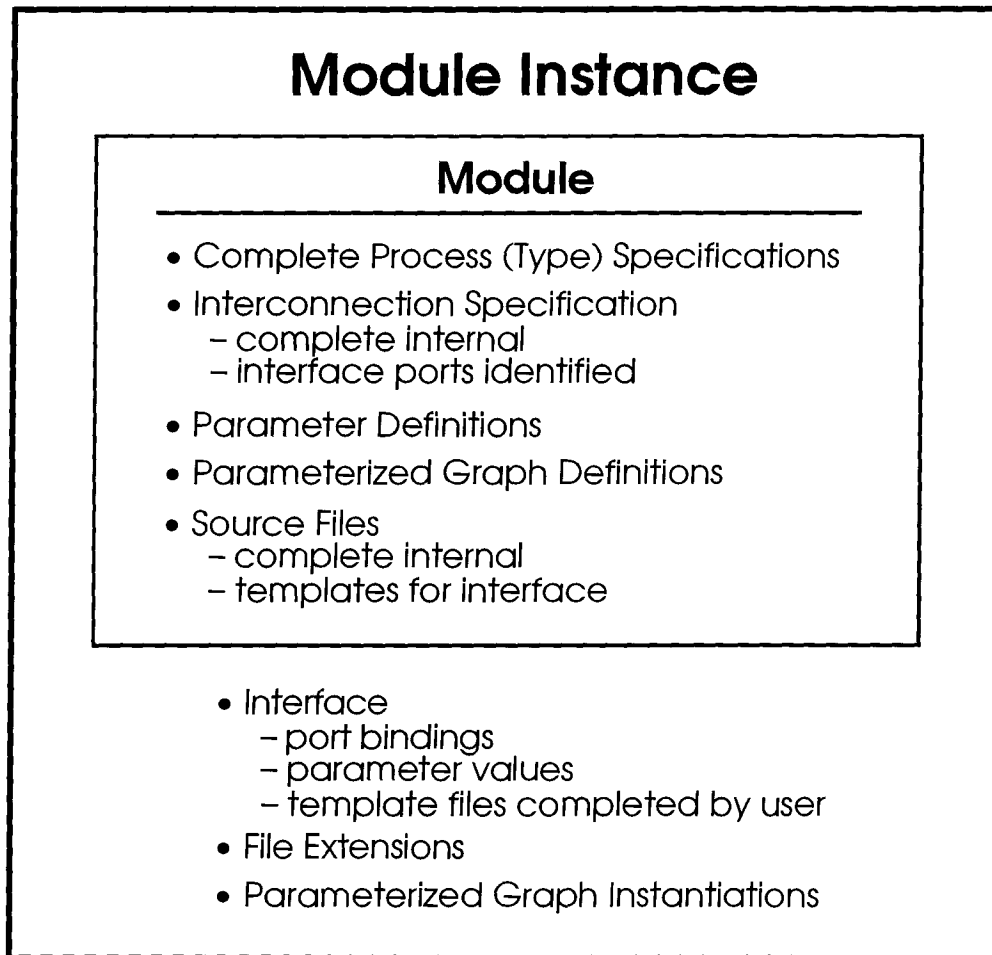


Figure 4.1 Relationship of modules to module instances

Currently, the module interface consists of parameters, ports, and source code files selected by the module developer. This interface, while addressing communications, is more generally targeted to facilitating as much (or as little) module flexibility as the module developer desires to provide. Enhancing the flexibility of a module increases its reusability, just as a

function may be generalized by adding parameterized options. We are not aware of other systems that provide an external structural interface beyond communication binding for a heterogeneous group of processes.

As noted previously, modules may contain other modules. Incorporating a module into a larger program involves binding the module's external ports to channels belonging to the module's parent. Ports are selected to be part of a module's interface by binding them to the placeholder channel `External` instead of a real channel when the module is created. At the interface external ports must be referred to by the combination of the port name and the name of the process owning the port since port names are only unique to a process and any process in a module may have any number of external ports. Otherwise, the binding of external ports to channels is the same as ordinary port to channel binding.

Modules either implement complete programs or STMs. Interface source code files (those created in the file category `External`) are the mechanism by which an instance of an STM is filled in and which provides STMs with their functional flexibility. These files are the collection of all files belonging to processes in the module created in the file category `External`. Many files are used by a number of processes because of the process type and file reuse mechanisms. Interface files can contain any amount of code the module developer wishes to provide for the user of the module, but each user gets a copy of the original which can be fully modified. Future modifications to the interface source code files cannot be directly propagated to existing instances since such changes are below the level of detail the system is aware of. Although it is not currently supported, it would be straightforward to warn the users of existing instances that interface modifications have been made and present the updated version which can be selectively copied to their existing version.

Parameters are the most general purpose component of the interface and provide STMs with their structural and performance flexibility. Most or all of the parameters for a module should be part of the interface since they are the most important mechanism for quickly changing a module without accessing its underlying representation. A number of uses for parameters within a module have been identified, while others have potential (§4.3). Parameters also provide a means by which information from outside the development environment can be used within the system (e.g., data from an analysis tool for a particular STM written by its developer).

4.3 Parameters

Module parameters are a key component of our environment for facilitating structural and performance flexibility and, hence, scalability. While parameters are central to STMs, their use is not restricted to that context. Parameters are not new; however, their nature in our environment is somewhat different from than in conventional contexts.

Usually parameters are used to either pass data to a module or to modify its runtime behavior. The former is not relevant to our system since data passing is handled via the separate communication pathways. The latter is a subset of a more general type of usage directed at modifying the structure and behavior of a module. However, many useful types of modification do not take place at runtime in a statically configured system (e.g., selecting the dimensions of a process graph structure). Furthermore, the computations determining the appropriate values for configuration parameters are not naturally located in the parent module, but rather in external analysis tools. Thus, parameters may, in our context, be thought of as settings rather than passed values.

Parameters may be either internal to a module or part of its interface (§4.2). This differentiation only extends as far as visibility – both variants may be used in exactly the same ways inside a module. Internally, parameters fulfill the role of program constants in conventional programming languages. Removed from the specific context of compiled languages, this role is essentially that of a parameter which is only available to the developer; hence, our treatment of them as variants of interface parameters.

Module parameters have a name, data type (e.g., integer, string), data string, kind, and description. The parameter ‘kind’ is a specification of what the source of the actual parameter value is and when it is obtained. This feature recognizes the fact that module configuration information may come from a number of sources and need to be obtained at different times in the development-execution cycle. Currently, three types of parameter kinds are available:

- static (user specified)
- external program to provide a value to the environment
- external program to provide a value at runtime.

The term ‘external program’ is used here to refer to a utility program not developed as part of the environment; i.e., written by the developer of a module, or possibly, an application. Each of these parameter ‘kinds’ has its own capabilities and limitations. Static parameters are the

default and are the value obtained by converting the data string to the parameter type. The value may be specified either by the user directly or by another tool in the environment under user direction, such as a general purpose performance analyzer. For the other two parameter kinds the data string is a host system command to invoke an external program which provides the parameter value on a standard output stream.

The last parameter kind is used as a runtime command line argument for processes and its value is never known by the environment. However, it is useful for machine generated, highly dynamic values which only influence the execution of the program. Many performance tuning parameters fall into this category since their values are process ‘constants’ for a particular execution, have little meaning to a user and can change with each execution of a process. This mechanism allows these changes to take place without any intervention on the part of the user or the environment.

The second type of parameter is like the last except that the environment executes the command and stores the value it returns. The user may force it to do this to see the value and may provide an alternative override value which will cause the program to be ignored. This kind is particularly useful for performance parameters which have effects on the structure of the module which is maintained and updated by the environment (e.g., the number of worker processes in a module, the number of children each process has in a tree structured computation).

Module parameters are currently targeted for use in

- process command line arguments
- parameterized structures
- weights
- compiler flags.

Other potential uses include

- selecting the type of a process network structure (e.g., a ring vs. a tree)
- selecting process types (e.g., to select a particular manager or worker algorithm implementation)
- setting pragma-type flags for processes (e.g., indicating priority, that a process should not share a processor with other processes).

Another possible logical extension for parameters is to allow them to be bound to other parameters or, even allowing any parameter to be replaced by a function of one or more

parameters. All of these extensions are implementable within the current framework and have potential uses; however, we require more experience with the current system before determining if they justify their implementation costs.

Notwithstanding the current limitations of parameters, no other existing parallel software engineering environment provides parameters integrated into the environment, much less with our degree of flexibility. ParaGraph does provide a facility for writing parameterized control scripts for graph structure definition, but the extent to which these parameters are visible to or managed by the environment is unclear [BaCuLo90]. Certainly, their role is restricted to the scripts and their means of definition restricted. Of course, most, if not all, systems permit parameters of varying types to be brought in through the back door of the underlying application implementation. Aside from the extra effort for programmers, this approach sacrifices the advantages of having parameters that are standardized, visible in the environment, influence environment constructs, and abstracted from the process implementation level.

4.4 Process Types

Unlike reuse in conventional programming contexts, process reuse in parallel/distributed environments may take two forms. The first, consisting simply of using the same process in several programs, parallels the conventional concept. However, in parallel/distributed programming, it is almost never the case that each process within a program is significantly unique. For example, master-slave models, domain decomposition models, and fault tolerant models all call for process reuse in the form of replication. In some restricted cases such replication can be completely hidden by the system as in Paralex where replicas of computation units are generated and managed by the system to create fault tolerance. Real problems arise, however, when the user wishes to create one or more replicas of a process so that the replicas are explicit components of an application, each with its own role and connections to other processes.

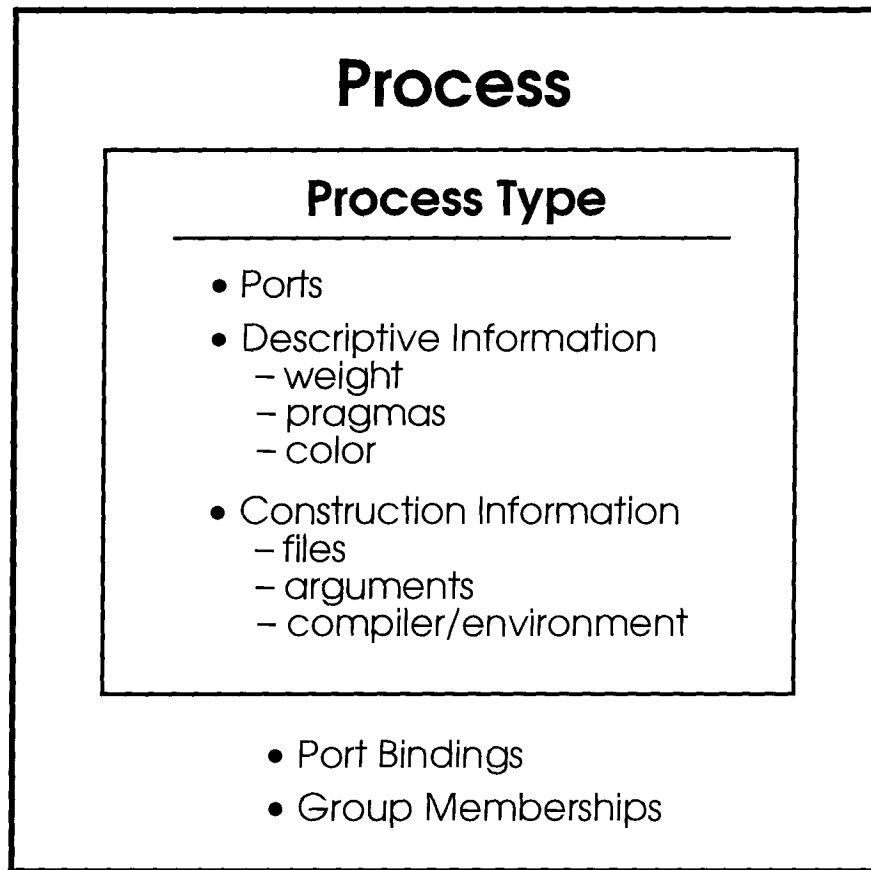


Figure 4.2 Relationship of process types to processes

As noted in §4.2 both forms of reuse are complicated by the use of static communication binding. The port primitive resolves this problem for our system and ParaGraph. However, many other systems require that the user manually replicate the processes by specifying each instance for precisely this reason. More sophisticated systems allow the user to copy a process specification and then modify it, but the copy becomes a unique object with no persistent connection to the original making modification of some common property tedious and error prone. Furthermore, there is no provision for handling the need to have slight variations of basic type (e.g., for a worker at the end of a chain as opposed to the other workers in the chain).

Our system addresses these issues by introducing the process type object. Processes are specified as an instance of a type or as a custom type (i.e., a unique instance). While the

system internally treats custom types just like other types, custom types do not need to be created by the user as a type; rather, the user need only manually fill in the necessary fields for a process specification. Process types specify most process structure and construction information which may be applied to an instance in one of two ways. The information may either be mandatory for all instances

- source code
- port names

or it may be default information in those cases such as

- compiler flags
- target environment (§4.7)
- pragma information (e.g., weight, priority)
- runtime argument list

where particular process instances may need to differ without affecting the basic identity of the process. The main piece of information which cannot be included in a process type is the binding of ports to channels. Note that without the communication interface abstraction provided by ports, process types would not be possible since each process would have to name its unique external channels.

Making some information only a type default and recognizing that other information must be specified as part of the process instance, permits the type specification to be as complete as possible without being as rigid as those systems that only permit exact replication. By using types creating a new process specification can be as quick as specifying the process (instance) name, choosing an existing type from a menu, and binding its ports to channels, thus satisfying the objective of easy replication. Furthermore, modifications to type information immediately apply to all processes of the type making modification quick and reliable.

4.5 Process Groups

Grouping objects is a successful technique for controlling complexity. Both the module mechanism and the process type mechanism provide system support for groupings with pre-defined meanings. However, in many cases structure, module, or application specific groupings will need to be defined.

Our system provides a basic grouping mechanism for processes available to both other functionalities (e.g., parameterized process networks) and directly to users. The underlying representation is the same in either case. Functionality is provided for the user to create, rename, or delete named groups, which processes can be added to or deleted from.

The current primary role for groups is to serve as an alias for a set of processes. For instance, support is provided for binding communication ports to a particular port existing in each member group of processes. As members are added or removed from the group, the actual port-to-port bindings are updated automatically. For example, consider a data collector process that wants to receive results from all the leaves of a tree of processes. Not only is the specification of the initial connections made easier by this functionality, but maintaining the connections if the dimensions of the tree change is automatic instead of a tedious manual process. This latter functionality is necessary for parameterized process networks (§4.6). Currently, for this binding operation to be valid, all members of the group must be of the same process type. Other possible alternatives include simply requiring that the port name used by group members for the connection be the same or, if port typing (§7.3.2) were to be supported, that these ports have the same type.

Another possible use for groups is to simplify user modifications of a set of nodes such as changing process weights or display characteristics. Other, not yet implemented, uses for groups will include aiding and simplifying interactions with other tools such as those performing trace analysis and visualization.

In other systems, the only close parallel to our system's process groups exists in ParaGraph. ParaGraph nodes/processes possess named attributes whose values are usually derived as part of the construction process. Information such as files can bound to nodes/processes based on the values of attributes or some function of them. A looser relationship exists with process groups in Via [Wi89] which are directly supported by the runtime system and whose membership and utilization are more dynamic than ours. One of the main functions of process groups in Via is to transparently increase throughput. In both this purpose and in their implementation they more closely resemble process clusters in distributed operating systems.

4.6 Parameterized Process Graphs

A parameterized process graph is some regular interconnection of processes whose dimensions are parameters. Each parameterized graph is a graph family. These interconnection networks may vary from the simple, such as rings and chains, to the moderately complex, such as trees and meshes, to the even more complex, such as hypercubes, butterflies, and shuffle exchange networks. This functionality is critical to providing scalable modules. As the size of networks grow, it will become increasingly important for developers to build and test programs on a minimal network and then easily scale the program to take advantage of the full set of hardware [BaCuLo90]. A similar need to rescale may exist when porting a program from one hardware platform to another since each is likely to have distinct resource limits.

In our system a parameterized process graph instance is created by selecting a topology type from a list of over a dozen frequently used interconnection patterns. Each topology type has one or two parameters which control the structure of the network (e.g., for a tree the parameters are degree and depth). These parameters may either be explicitly entered by the user or bound to an environment parameter (§4.3). Once the parameters are specified the network can be instantiated. In addition to the implicit description, the instantiation process inserts an explicit representation into the module process network. Changing the implicit description causes the old explicit representation to be removed and replaced with the one specified by the new implicit description. The explicit representation may be modified directly, although any changes are lost if the implicit representation is changed. In addition to providing this ability to tweak the parameterized network, having an explicit representation permits a uniform representation of the module network to be maintained. From its point of view the parameterized network facility is simply a macro operation on the module network.

The availability of process groups is necessary for parameterized process networks. In addition to parameters, each topology type has a set of predefined, frequently overlapping, process groups based on conventional labeling schemes. For instance, for a tree the groups are: root, leaves, last (leaf), internal (i.e., not root or leaf), all, and remainder (i.e., all nodes not in another selected group). The membership of the ‘remainder’ group will vary depending on which other groups are selected for use and, thus, provides a mechanism to insure that the set of nodes can be cleanly and completely partitioned regardless of which of the other groups a

user wishes to use. In general, all topologies have the basic groups of first, last, all, and remainder. By binding communications from outside the parameterized network to groups inside the network rather than specific processes, correct bindings can be preserved even if the structural parameters are changed. These groups also provide an easy means of binding process types to individual processes in such a way that process type assignments will also be preserved in spite of parameter changes.

While this feature in our system addresses the same scalability concerns as ParaGraph, there are a number of significant differences. In ParaGraph network scalability is intrinsic to the network construction mechanism. Networks are specified in ParaGraph by specifying the productions of a graph grammar and iteratively applying those productions to construct the members of the graph family specified with the grammar [BaCu87]. This mechanism is certainly more powerful and general than the one employed by our current implementation. However, it does have at least two drawbacks.

The first is that constructing a graph grammar is not always an easy or intuitive process. While a family of trees is relatively easy, even other simple structures such as meshes require a considerable amount of effort and understanding. However, there are other approaches to the generation of graph families such as the use of graph operators (e.g., like those in [Ha69]). In many cases these operators offer a much simpler means of defining a graph family than do graph grammars. For example, the specification of a hypercube is almost trivial using the graph-theoretic cross product operator ($\otimes K_2^n$). Many of these operators have the advantage of mapping very simply to graphical interfaces, unlike graph grammars. For example, to build a hypercube a user could draw a graph consisting of two connected nodes, select these nodes, and then invoke the cross product operator as a menu command. Repeating the process of selecting the original nodes and those produced by the command and invoking the command would allow a user produce in n steps a 2^n hypercube. This type of functionality is available in a few existing graph construction and editing packages. While such operators are not supported in our current implementation, similar support for quickly building complete graphs or subgraphs is provided (holding down one key when creating a node causes edges to all other nodes to be automatically generated, holding down a different key causes edges to all selected nodes to be created).

The second is that of the considerable implementation complexity. It is not clear that the generality of ParaGraph is required by most users, particularly when the generality

significantly increases the difficulty of taking advantage of the functionality. It is far from clear that a set of predefined graph families frequently used will not satisfy the needs of the vast majority of users. The considerable gains in both implementation and usage simplicity made possible by taking this approach make testing its validity a worthwhile exercise before proceeding to more complex solutions. Both graph grammars and operator based approaches could be used to provide a more general and powerful implementation foundation than the current C codings. Ultimately, however, the goal must be to provide the required generality with the simplest user interface or combination of interfaces possible.

4.7 Target Environments

A parallel application will, in many, if not most cases, run in a heterogeneous environment. For example, even relatively homogeneous machines such as a hypercube or transputer network will be connected to some host system. This means that different process types, or even different instances of the same process type will need to be compiled for different machines with different architectures. Alternatively, different process types may be written in different languages or require special support (e.g., extra header files and library files for a process with a graphical interface).

The abstraction of a target environment configuration allows the creation of named, reusable configurations which may then be specified for each process type and overridden for individual process instances. Information included in a compiler configuration includes a compile host, compiler name, source and target file suffixes, include file paths, library file paths, basic libraries, and standard compiler flags.

4.8 Files

A relatively simple type of object dealt with by the system is files. Currently the system supports three categories of files:

- internal source code
- external (module interface) source code
- libraries

Files are all currently maintained on the host's network filesystem. As a consequence filesystem specific path information is part of the file object specification and naming must be compatible with host conventions. An important feature related to managing files is supporting reuse at the file level. By doing so code updates are minimized and the impact of updates is well defined and can be automatically managed as well.

Ultimately it would be best to move files into the environment database once limitations and performance issues are resolved. Such a move should prevent many problems with accidental file corruption or loss, locking, and portability. For example, the Inversion filesystem feature of Postgres supports storing large objects such as files in the database and preserves all critical database functionality (e.g., transaction protection, access via the query language) for these objects, while also providing an alternate interface which appears to access a filesystem [Postgres92].

4.9 Summary

This chapter has discussed a number of objects built on top of the process-port-channel primitives presented in Chapter 3 in order to satisfy many of the design objectives discussed in the first two chapters. Some of these objects such as projects and modules are structural units of organization for aggregates of primitives. Modules also facilitate reuse as does the process type object. The related objectives of scalability and performance tunability are clearly addressed by objects such as module parameters, parameterized graphs, and process groups. The next chapter will discuss user interface design issues in which both the primitives of Chapter 3 and the modules described in this chapter will play a role.

5 Interface Issues

5.1 Base Program Representation

A user interface and particularly a graphical user interface needs a base representation. The existence of a base representation provides a number of advantages. It can define or at least help define the minimal information needed to specify a complete project. It also provides a default representation which is always applicable and valid. The existence of such a representation also gives a user's interactions with the interface a sense of rootedness since one will always begin and normally return to the base view. Finally, it allows all other views to be as specific or general as desirable and to added or removed from the system without serious consequences. These additional views will be referred to as auxillary views.

A basic choice when graphically representing a distributed program is whether to represent the program as a process graph or a control flow graph. While these representations are by no means mutually exclusive, each has its inherent strengths and weaknesses which tend to strongly influence the programming idiom being implemented.

Process graphs are a common high level representation for multi-process programs such as ParaGraph [BaCuLo90]. The nodes in such a graph represent processes and the edges represent the existence of a communication/data dependency between two or more processes. Edges may be directed or undirected, depending on the amount of information which one desires to reflect in the graph. A key aspect of process graphs is that they provide no information about when or how often communications take may take place and, in the case of undirected graphs, the direction of the communication.

Control flow graphs (also called dependency graphs) are commonly seen in existing graphical interfaces for parallel programming environments such as CODE [Br89] and HeNCE [BeDo91a]. The nodes in a control flow graph represent sequential units of computation while directed edges represent data inputs to or outputs from the unit of computations. While many systems restrict their graphs to be DAGs for simplicity or do not support loops well (e.g., CODE), the control flow graph model and more sophisticated systems do support loops (cycles) within graphs (e.g., HeNCE). A key feature of the control flow graph is that it pushes communication specification to the graph level. That is, the direction, timing, and to some extent the frequency of communications are exposed since computations can only receive inputs when they begin and produce outputs when they terminate.

The process graph is particularly well suited to process oriented paradigms such as CSP or client-server, it can represent all of the paradigms we are aware of with equal ease. In contrast, the control flow graph is only simple for the generalized dataflow paradigm. While its full form has the same representational power as the process graph, in most cases the control flow graph will be considerably more complicated both in terms of the number of nodes and edges and because of the various types of edges that must be employed. For example, most persistent, non-trivial processes must be represented by a number of nodes in a flow control graph connected in a loop structure.

This is not merely a representational complexity issue, but also a descriptive problem in that the control flow graph provides no clear representation of what will become a process in the final implementation. Thus, not only is an additional, non-trivial task created for the environment, but also a human comprehension problem. How can a person assist in final instantiation tasks such as mapping when there is no clear relation between his input and the objects which the instantiation task is manipulating? In contrast, process graphs, while not in any way precluding dynamic assignment schemes, directly map onto static physical resource allocations in distributed programs with the nodes/processes assigned to processors and the edges/channels assigned to communication pathways. While static resource assignment is not the best policy in all situations, the costs of dynamic allocation are too high for smaller granularities (e.g., single processes) in MIMD architectures where, at least from a numerical point of view, most of the structural complexity lies.

The simplicity of the process graph representation is also its weakness. A process graph specifies the minimum amount of information needed to specify a distributed program;

i.e., its processes and the existence of dependencies between them. Much information which might be required or useful to complete the program or advise tools (e.g., temporal ordering information, details about dependencies) is not present in this representation while it is present in the control flow graph. Thus, the control flow graph is better suited for time dependent descriptions and is widely used as an input to scheduling algorithms.

We have adopted the process graph as the primary graphical representation because its simplicity makes input, display, and user understanding easier. Furthermore, the majority of graph objects in any program (i.e., the lower levels with the smallest granularity) will be statically scheduled in most cases. Finally, the control flow graph can be utilized as an auxiliary representation in those cases where it would be advantageous to represent the additional information.

5.2 Controlling Display Complexity

The concept of the subgraph is a useful tool in dealing with complicated graphs. However, the general concept of a subgraph can itself be quite complex since a single node may be a member of any number of subgraphs. Restricting subgraphs to be strict partitions of the master graph does, on the other hand, provide a useful mechanism for controlling representational complexity in graphs. Furthermore, the nested use of such subgraphs creates a simple, easy to grasp hierarchical grouping of nodes in the graph. While most graph packages (e.g., EDGE [Ne88]) support the subgraph abstraction, most graphical parallel programming tools do not. The notable exception is CODE.

The subgraphs of a larger graph can serve the two related, but distinct, functions of display organization and program decomposition. This difference is analogous to the difference in programming languages between creating code blocks by the use of comments and blank lines as opposed to making a code block a function or procedure. In other words, the former is purely a superficial decomposition to aid readability and programmer understanding while the latter is a structural decomposition recognized by the language and compiler. CODE does not make this distinction and treats each visual subgraph as a decomposition of the program. In contrast, subgraphs in our interface may be either a view or a module, each of which is designated with a distinct symbol.

A view is a logical grouping of nodes and edges which has been contracted into a subgraph to represent a logical relationship and/or to control the complexity of the parent view. Views may of course contain other views. The term view is used to imply that the subgraph primarily exists as a display convenience and has little or no meaning with respect the construction of the program (a view may be used as a pragma for contraction tools).

A module is essentially a view which has been identified as a distinct structural unit. As described in §4.2 modules have an external interface and are a unit of reuse. Since modules are also views there is no conflict between the two definitions. The fact that a module is simply a special kind of view is reflected in the fact that new modules (as opposed to imported module instances) are created by modifying an existing view. Modules may contain any number of views and other modules. Thus, as described in §4.1 a project consists of a hierarchical graph of modules which may themselves be logically partitioned into a logical graph of views (see Figure 5.1).

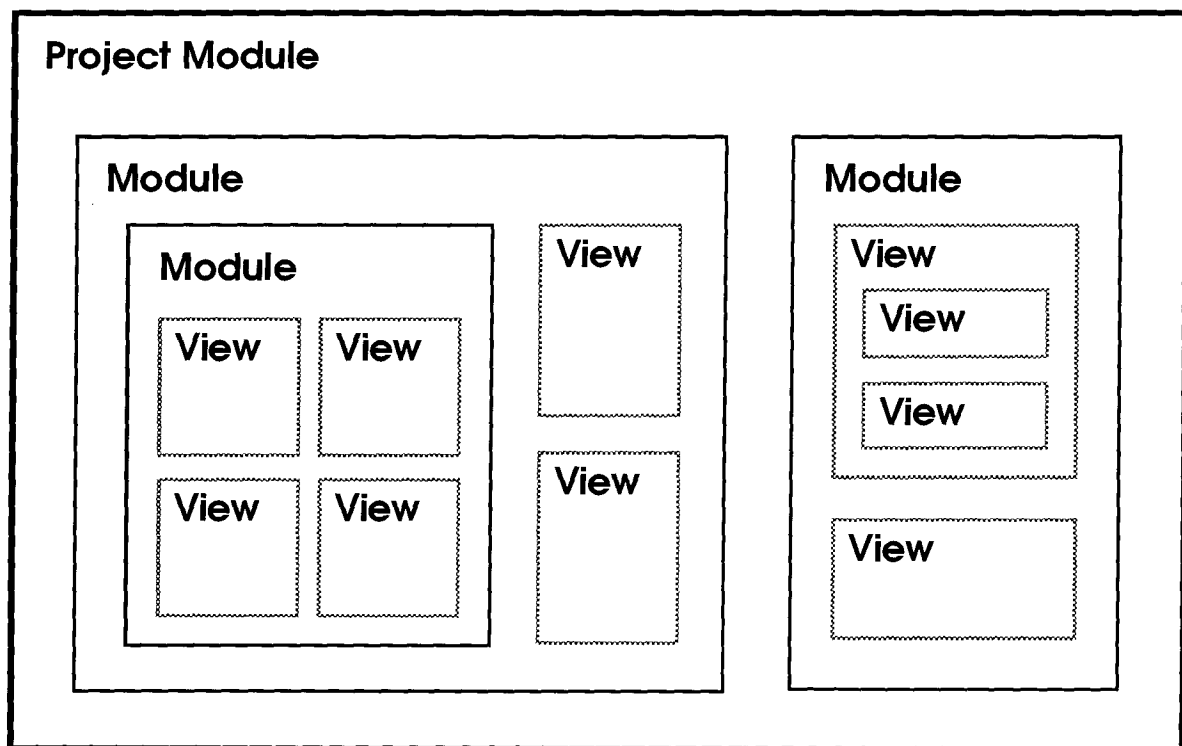


Figure 5.1 Example of hierarchical structuring using modules and views

Making the distinction between views and modules is useful in several ways. First, while it is possible using the CODE scheme to have subgraphs that are only for logical purposes and others that are actual program subunits and/or units of reuse, there is no representational or internal distinction of this real structural difference. This failure to make a distinction between unique and reused program components contributes to the problem in CODE of propagating updates to the master version of a reusable module. Finally, making the distinction provides for greater functional flexibility since not every subgraph must be a reusable program module. This is particularly important in our system because of the extra features associated with modules (e.g., their interface) in addition to their additional descriptive and internal overhead.

While the utilization of subgraphs can control graph complexity, it does not solve the problem in all cases. One basic aid provided by our system is the preservation of layout information, even in cases of reuse, so that manual effort invested in a layout is not wasted. Also, because all modules are subgraphs there is no problem with mixing a newly created module instance graph with that of the parent. In addition, our system provides a standard layout with each of the parameterized graph structure types. This support is particularly important for our system since these subgraphs are the most likely to be large as well as frequently structurally modified.

Nevertheless, there will inevitably arise large and complicated subgraphs which are not logically decomposable. In these cases techniques for automatically laying out large and complex graphs are required. Many examples of such techniques can be found in graph tools such as EDGE [Ne88]. Each of these techniques attempts to optimize one or more particular “good” graph properties such as the number of edge crossings or proximity of connected nodes. However, most techniques are restricted to or only work well for particular classes of graphs since optimizing graph layout is in general an NP complete problem. ParaGraph intends to exploit its knowledge about the underlying structure of a graph (derived from the AR grammar productions for the graph family) to choose techniques which will work well for that particular graph's family. In order to handle these cases, we also provide a general purpose graph layout algorithm which has been found to be successful in producing visually attractive layouts for a wide variety of graphs [FrRe91]. This algorithm models the graph as a many body problem where the edges are springs connecting the nodes which repel each other. The layout produced is an equilibrium point in the many body simulation.

5.3 Interface Levels

As noted in §1.2 parallel program development can be viewed as consisting of tasks at several distinct levels of abstraction. While the number and specific definitions of these levels is an open issue, the concept of supporting more than one such level in a programming system is not novel [Sn91]. However, there is little or no precedent for designing a smooth transition between such levels or actually implementing such support, particularly using a graphical interface. Of course any system providing modularity and easy reusability can be said to provide support for “higher level” programming in the sense that a C application programmer can avoid writing system code by using the standard Unix libraries. However, there is a difference between providing a set of utilities such as those in a Unix library and providing a complete program skeleton which is only missing a few task specific routines. Furthermore, there is a significant difference in abstraction between building an application out of language primitives and functions provided by libraries and building an application out of a set of skeletal programs joined by communication channels.

As was also noted in Chapters 1 and 2 the concept of paradigm based programs is not new, and, as noted in §2.3.6 the PIE project proposed the integration of paradigm based programming into their environment as early as 1985. However, such integration has not been seen until recently (e.g., the Linda Program Builder [AhCaGe91]) and has yet to be seen in a graphical environment. Public STMs provide this functionality in our system. By simply importing an existing public skeleton module, filling in the interface files, and setting the interface parameters a user can quickly and easily create a complete parallel program incorporating considerable optimizations without knowing anything about parallel programming in general or the system on which he will be executing the program in particular. By importing multiple modules and connecting them via their external ports one can easily create an integrated, multi-function parallel program. Since a user can perform all of these functions to create complete programs in the system without entering the lower level (the contents of the modules) or possessing a significant amount of the expertise required to program at that level, Parsec does provide two different levels of interface. Furthermore, it does so transparently within a unified framework.

5.4 Auxillary Views

The need for and establishment of a base representation and display in no way diminishes the role of or need for auxillary views, in fact, they are increased. A primary role for such views is to provide specialized interaction with other tools in the environment such as schedulers and mappers. For example, the control flow graph rejected as a base representation is precisely what is needed for specifying time orderings and interacting with a scheduler.

5.5 Summary

This chapter has discussed several user interface design issues. The chapter began by motivating the need for a base representation and the choice of process graphs for this representation. The need for auxiliary, special purpose representations was also noted. Restricted subgraphs were presented as the means of hierarchically organizing a program within the user interface. These subgraphs correspond either to the modules presented in the previous chapter or to logical groups of processes called views. It was also noted that hierarchically organized module subgraphs provide for multiple levels of programming abstraction with smooth, consistent transitions between the levels. The following chapter will, in the context of a brief example, provide an overview of how the elements from the previous three chapters have come together in the current implementation.

6 Implementation and Example

This chapter presents as an example the specification of the processor farm STM described in [FeSrWa92] and briefly describes how it may be included in an application. This STM provides optimal performance on a balanced, rooted k-ary tree and has parameters for tree depth and granularity. The granularity parameter controls the aggregation of individual task data units into larger units for the purposes of communication and assignment to worker processes. The degree of the tree is also a potential parameter but is of less interest since for this STM it is a system parameter rather than an application parameter because the optimal value is the largest that can be physically realized on a given system.

We will begin the process of defining the processor farm by defining its parameters and then the parameterized graph controlled by one of the parameters. This parameterized graph constitutes the main structural component of the module. The following step will be to complete the process type specifications for the processes in the parameterized graph. We will then add a custom process to serve as the ‘master’ process for the processor farm and as part of its definition establish its communication bindings to the parameterized graph. Finally, we will see how the processor farm can be turned into a reusable module and included in other projects.

6.1 Defining Parameters

Figure 6.1 shows an empty project frame with its menu buttons and its ‘Graph’ menu pulled down. Most of the important functions used in this example are available from this menu. A logical place to begin defining a STM is its parameters since several other constructs

utilize them. Figure 6.2 shows the parameter dialog accessed by selecting 'Parameters...' from the 'Graph' menu. This dialog utilizes a basic user interface construct employed throughout the interface – the add-change-delete list. In this case the list contains all of the parameters within the module from which it was activated (noted in the footer of the window). Items may be added by filling in or setting the attribute fields below the list and then pressing the 'Add' button. The attributes for an existing item are displayed by selecting the item in the list. These items may be changed and the changes committed by pressing the 'Change' button. A new item derived from an existing item may be easily created by selecting the original, making the changes, and then pressing the 'Add' button. Pressing the 'Delete' button simply deletes the selected item in the list.

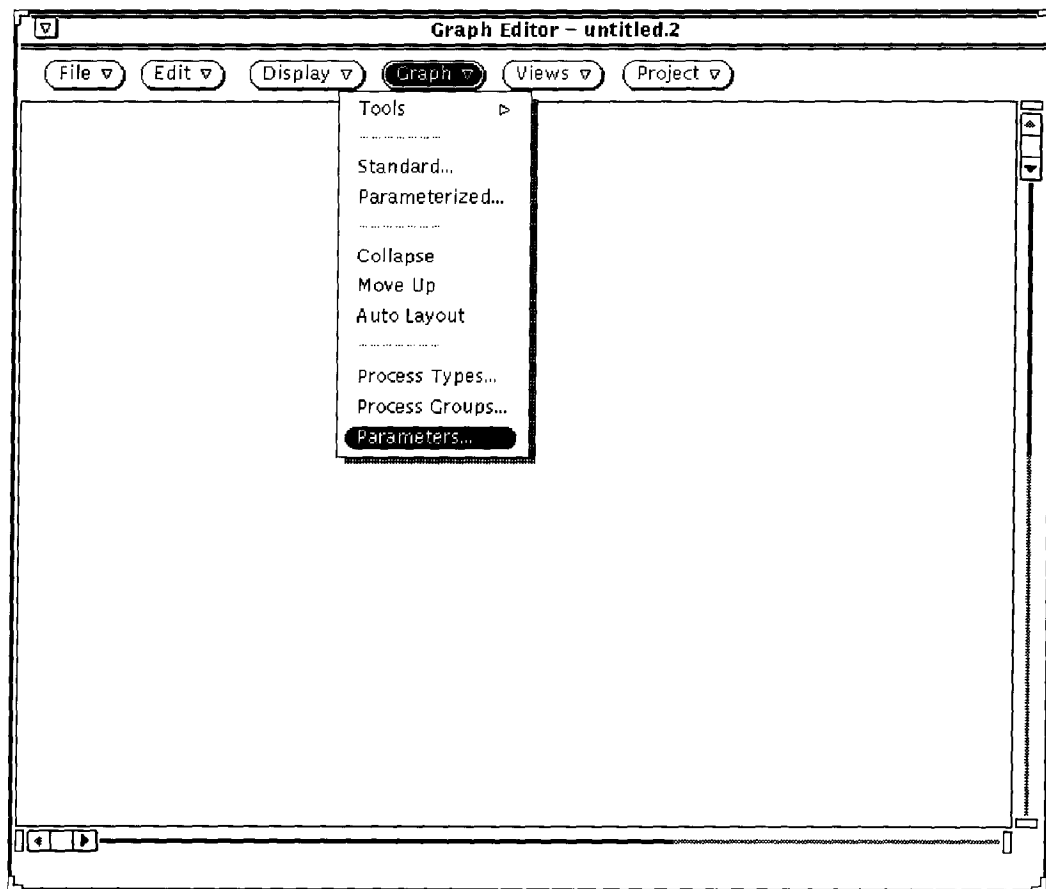


Figure 6.1 Empty frame and 'Graph' menu

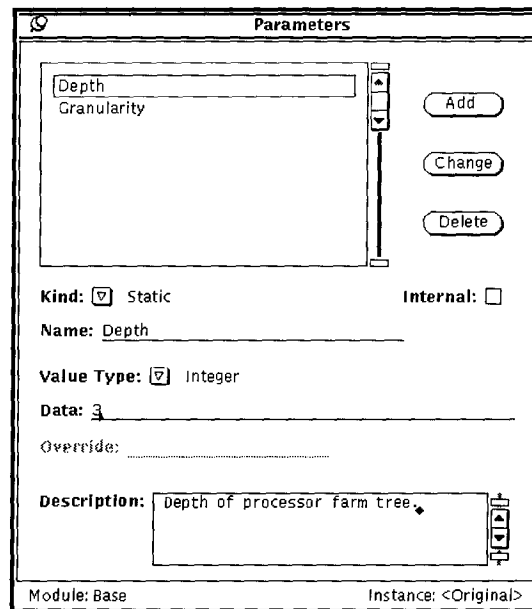


Figure 6.2 Parameter dialog

The list in Figure 6.2 shows the two parameters for the STM and the attribute values for the selected parameter, 'Depth.' For the purposes of this example, both parameters are specified as being of kind 'static.' However, in the actual STM they would be specified as values provided to the environment by an external program. For this STM that program is a performance analysis tool written by the author of the STM and based on the underlying analytic model. This tool provides, depending on its arguments, the appropriate parameter value based on statistics stored in a file by a previous execution of the application. The name of the analysis tool and its arguments would be specified in the field where the value of a static parameter is specified (here '3').

6.2 Defining a Parameterized Graph

Once the parameters have been specified, the next logical step is to specify the parameterized tree that will be the main part of the STM. Note that the STM design calls for the tree to consist of only workers, with the master process connected to the root of the tree. Figure 6.3 shows the dialog for specifying parameterized subgraphs, once again based on an add-change-delete list.

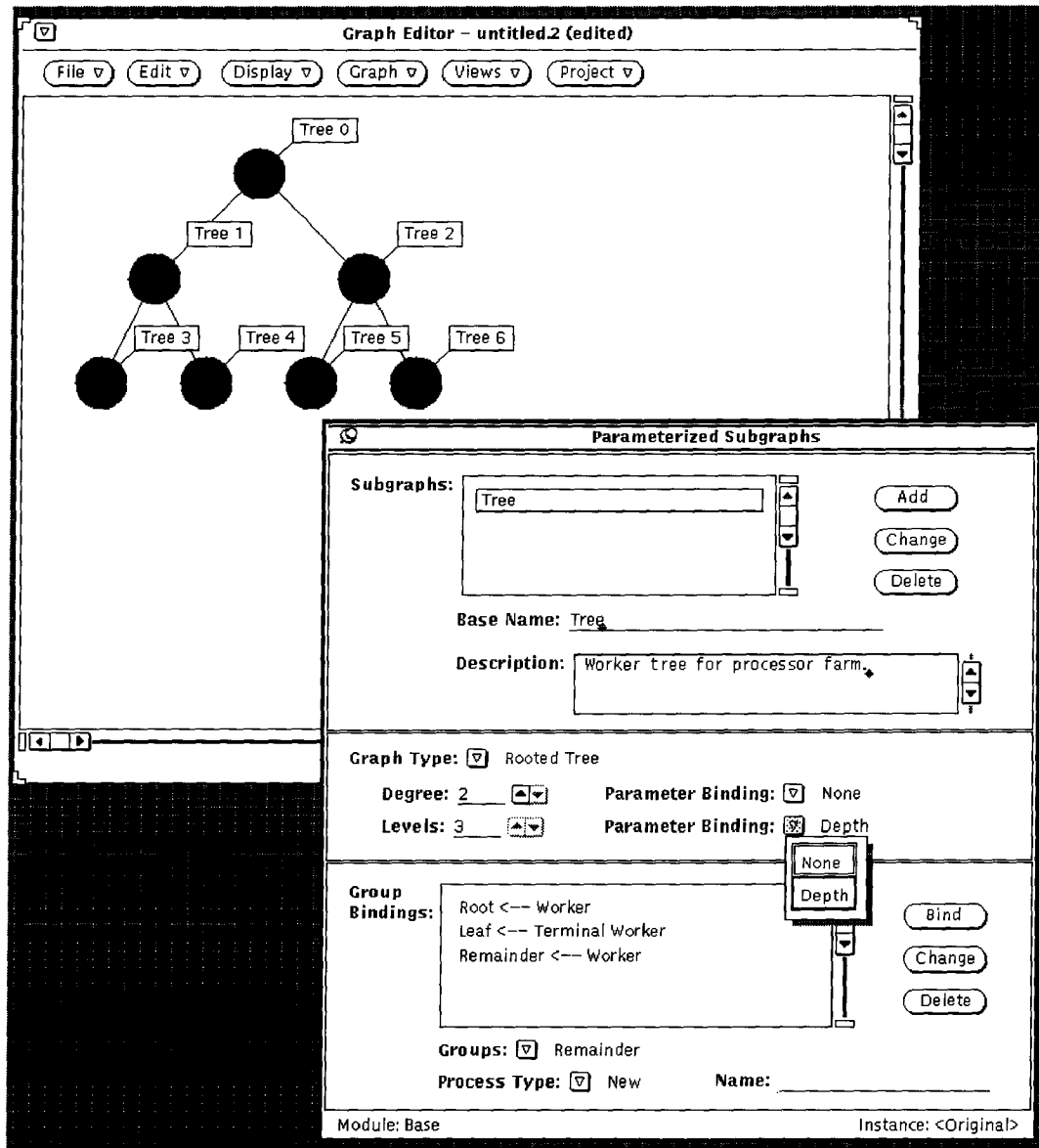


Figure 6.3 Parameterized subgraph dialog and its result for the processor farm

Below this list is the first group of attributes for a parameterized subgraph. These specify the structure and dimensions of the graph family. The first field is a menu stack which currently lists over a dozen standard interconnection networks. Here we have selected 'rooted tree.' There are two other pairs of fields for specifying the parameters of the selected graph family (only one pair may be visible if the graph family only has one parameter). The labels

for the pairs depend on which graph family has been selected. The first item in the pair is the current value of the parameter which may be set here manually or may be derived from the value of the second item. It is used to bind the graph family parameter to a module parameter such as the one we created in the last step. If such a binding is made the first field shows the current parameter value but is not editable. Here we bind the depth parameter to the module parameter ‘Depth’ we previously created but leave the degree dimension as a manually set value of ‘2’ to specify a binary tree. A parameterized subgraph will immediately reflect changes implied by a change to a module parameter to which one of its parameters is bound. Manual changes to unbound graph family parameters are made by changing the value and pressing the ‘Change’ button.

The second group of parameterized subgraph attribute fields comprise a variant of the add-change-delete list used for establishing bindings. Here the function is to bind process types to groups of nodes within the subgraph. There is a menu stack which lists the groups for the current interconnection structure (for a rooted tree: root, last, leaves, remainder, and all). The other menu stack lists existing process types. New types may be created by selecting ‘New’ from the menu and filling a new name in the adjacent field. A group and process type is bound by pressing the ‘Bind’ button. For the processor farm there are two different process types for the workers in the tree. The workers for the leaves must be slightly different since they cannot pass work on to children. Those groups used here will be also available for communication bindings. Thus, we need to specifically note the ‘Root’ group for our example even though it has the same process type as group ‘Remainder.’

Once the specification is complete, the ‘Add’ button may be pressed. This causes a number of actions to take place. First, the group to process type bindings are validated to make sure every node in the graph has received a single process type binding. If this is correct, the system proceeds to create any new process types, their ports, and then the graph for the current dimension settings as is seen in the main window in Figure 6.3. Currently, process types not created by the parameterized subgraph mechanism must already have the documented canonical port names for the group the type is bound to. A future version will allow the user to map the canonical port names to arbitrary port names in existing parameter types.

Process Types

Types:

Terminal Worker	▲ ▼ Add Change Delete
Worker	

Name: Terminal Worker Files...

Ports:

Child0	▲ ▼ Add Change Delete
Child1	
Parent	

Name: _____ Flow: ☒ In ☒ Out

Description: _____

Constraints: ☐ Anti-social ☐ High Priority

Weight: 1 ▲▼

Type Description: Processor farm worker process without children. ▲▼

Module: Base Instance: <Original>

Figure 6.4 Process type dialog

6.3 Defining Process Types

The next step in specifying the processor farm is to complete the process type specification for the process types automatically created by the parameterized subgraph mechanism. This is done by selecting 'Process Type...' from the 'Graph' menu to display the dialog in Figure 6.4. Initially the process types are already present in the list and the ports for each of them already specified (ports are listed and modified in the first group of attributes

below the main list). We may still, however, want to fill in some of the information the second group of attributes to elaborate and refine the process specification.

Type Creation Information - Terminal Worker

Internal Files:

- prog.h
- slave.c

Buttons: Add, Change, Delete

Path: /project/transputer/tips/tests/test2

Source File Name: slave.c

Description: Core C implementation of processor farm STM worker for transputer under Trollius.

Arguments:

- <Granularity>

Value: _____

Buttons: Add, Constant (processor id), (process name), <Depth>, <Granularity>

Target Environment: ☒ Transputer (Trollius)

Compiler Configuration: ☒ Default

Compiler Flags: -DLAST -DDEGREE=2

Buttons: Apply, Reset

Module: Base Instance: <Original>

Figure 6.5 Process type creation information dialog

As noted in the discussion of process types, process type information is divided into two categories: descriptive and construction. The dialog in Figure 6.4 deals with the descriptive. The dialog in Figure 6.5 deals with the construction information, including source files, and is accessed by pressing the “Files...” button in Figure 6.4. The first panel in the dialog is for specifying the files used by the process type: internal, external (interface), and library. The second panel is for specifying the parameter list for processes of the type. The ‘Add’ button has an attached menu which list of module parameters (those items in <>’s), special values to be filled in at load time (those items in ()’s), and the default item ‘Constant’ which installs the string in the ‘Value’ field. The final panel contains compilation and target information.

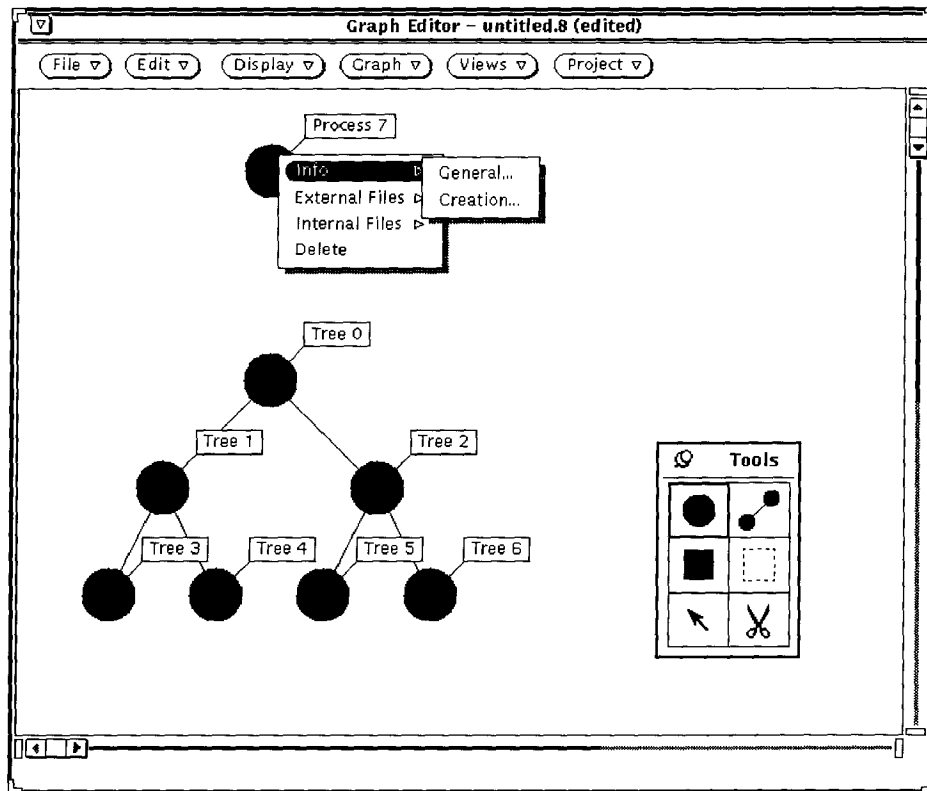


Figure 6.6 Manually creating a new process

6.4 Adding Processes

Once the process type information for the parameterized tree is complete, the next step is to add the master process. Nodes may be added manually by selecting appropriate tool from the tool palette shown in Figure 6.6 and then clicking the mouse where you want the icon placed. This can be obtained either from the 'Graph' menu or by pressing a mouse button on an empty area of canvas. By clicking on an existing process with the appropriate mouse button, you can obtain the process menu which allows you to alter or delete the process.

6.5 Establishing Port Bindings and Other Descriptive Information

By selecting the general information option you can access the process information dialog in Figure 6.7. This dialog allows you to name the process and specify its type if you

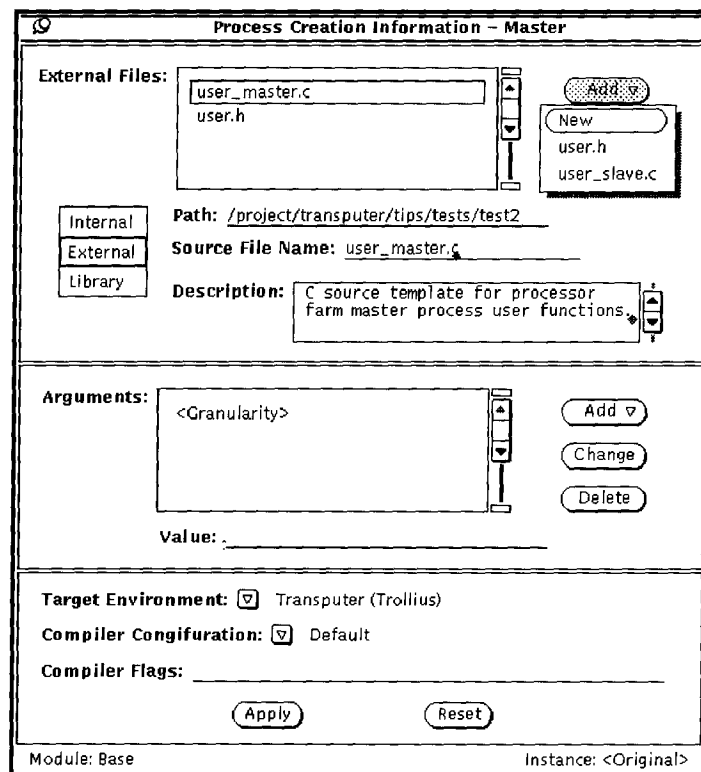
wish it to be of a standard type. Since this dialog includes the information contained in the process type structural specification, that information will be reflected here if you make that choice. Unbound ports are displayed in the port binding list as bound to the special channel 'None.' If, on the other hand, you desire a custom type, as in this case, type information can be filled in here (changing type information for a non-custom process will cause that process to become custom). In addition, this dialog allows specification of information such as group memberships and, most importantly, port bindings that cannot be specified as part of a type.

The image shows a 'Process Information' dialog box for a process named 'Master'. The 'Type' is set to 'Custom'. The 'Groups' list is empty, with 'Add' and 'Delete' buttons. The 'Port Bindings' section contains two entries: 'Interface == External' and 'Worker == <Tree.Root::Parent>'. Below this is a 'Channel' button and a 'Group' button. The 'Bind' section shows 'Local Port: Worker' and 'Flow: In'. The 'to' section shows 'Group: Tree.Root' and 'Port: Parent'. A dropdown menu for 'Port' is open, showing 'Child0', 'Child1', and 'Parent'. The 'Description' field contains the text: 'Communication of tasks and results with root worker of parameterized worker tree'. The 'Constraints' section has 'Anti-social' unchecked and 'High Priority' checked. The 'Weight' is set to 2. The 'Process/Type Description' field contains the text: 'Master process for processor farm STM. Responsible for generating tasks for workers, receiving results, and communicating with STM parent.' At the bottom are 'Apply' and 'Reset' buttons. The status bar shows 'Module: Base' and 'Instance: <Original>'.

Figure 6.7 Process information dialog for master process

Two different categories of binding are possible here: channel and group. For our example, we need both a binding to the root node of the parameterized tree and to the STM interface. The latter is relatively trivial and can be done by specifying a port name and binding it to the special channel 'External' (a menu of available channels is displayed in place of the 'to

Group' menu if 'Channel' mode is active). The 'to Port' menu is inactive in channel mode unless there is more than one channel to the same node in which case the destination port names are needed for unique identification. For the connection to the root process of the tree we could have drawn in a channel and used a channel binding. However, the channel would have been lost if the parameterized tree were regenerated because of a change. To obtain a permanent binding we can use the 'Group' mode to bind this process to port 'Parent' of the group 'Root' using the group menu and port menu (which only displays the ports for selected group) available in this mode. Channel edges in the graph are automatically maintained for group bindings.



The dialog box is titled "Process Creation Information - Master". It contains several sections:

- External Files:** A list box containing "user_master.c" and "user.h". To the right is an "Add" button and a small menu with "New", "user.h", and "user_slave.c".
- Path:** A text field containing "/project/transputer/tips/tests/test2".
- Source File Name:** A text field containing "user_master.c".
- Description:** A text area containing "C source template for processor farm master process user functions." with a vertical scrollbar.
- Arguments:** A text area containing "<Granularity>" with a vertical scrollbar. To the right are "Add", "Change", and "Delete" buttons.
- Value:** A text field containing ".".
- Target Environment:** A checked checkbox next to "Transputer (Trollius)".
- Compiler Configuration:** A checked checkbox next to "Default".
- Compiler Flags:** An empty text field.
- At the bottom are "Apply" and "Reset" buttons.
- At the very bottom, it says "Module: Base" on the left and "Instance: <Original>" on the right.

Figure 6.8 Process creation information dialog for master process

6.6 Defining Process Creation Information

Figure 6.8 shows the process creation information dialog for the master process which is accessed by selecting the creation information option from the process menu. This is identical to that used for non-custom types. Notice that the files entered earlier are now

available as items on the 'Add' button menu. Thus, we can with almost no effort re-use files and prevent manual duplication errors. In this case we need to re-use the external file 'user.h.'

Figure 6.9 shows the completed processor farm graph. Also shown is a file access menu item for the master process. These menu items are maintained automatically based on the information supplied to the creation information dialog. From here you can directly access a specific file or even function within the file in the environment's editor.

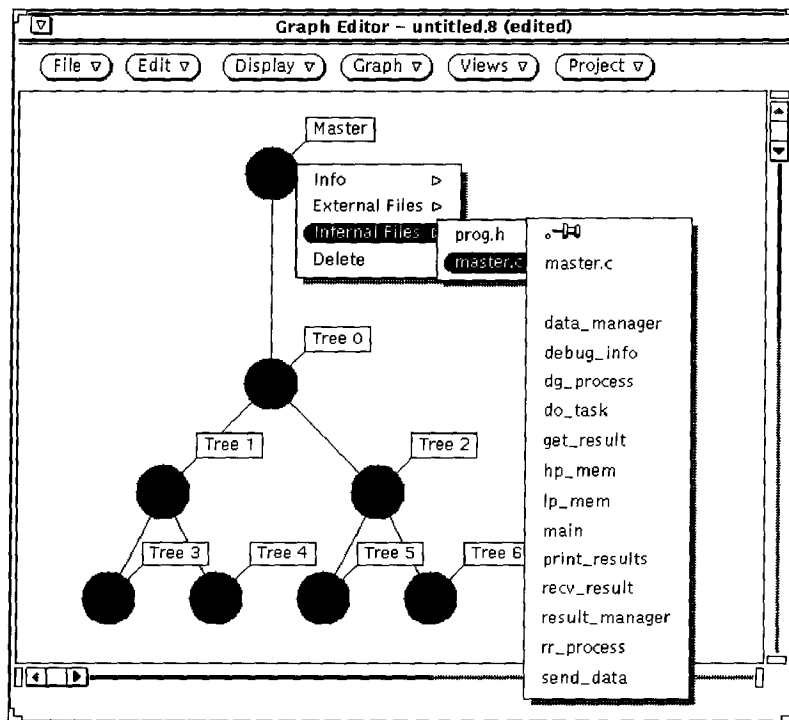


Figure 6.9 Completed processor farm process graph

6.7 Creating a Reusable Module

Once the description is completed the next step is to turn it into a reusable module. The first step in this process is to turn the graph into a view subgraph. This is done by selecting a set of nodes (in this case all of them by using 'Select All' from the 'Edit' menu) and then selecting 'Contract' from the 'Graph' menu. Figure 6.10 shows the resulting display. Like processes, views have a menu accessible by clicking on the view icon. We could use the 'View Subgraph' option to see the original graph, but right now we want to turn this view into a module. To do this we select the 'Info...' option from the menu to get the dialog also shown

in Figure 6.10. To achieve the conversion we change the boundary type from 'Logical' to 'Module' and fill in the appropriate name fields. Once this is done we have a custom module which can be made public by pressing the 'Export' button (which is only enabled for custom modules). At this point we have completed our initial task and have a new module icon and access to its menu as displayed in Figure 6.12.

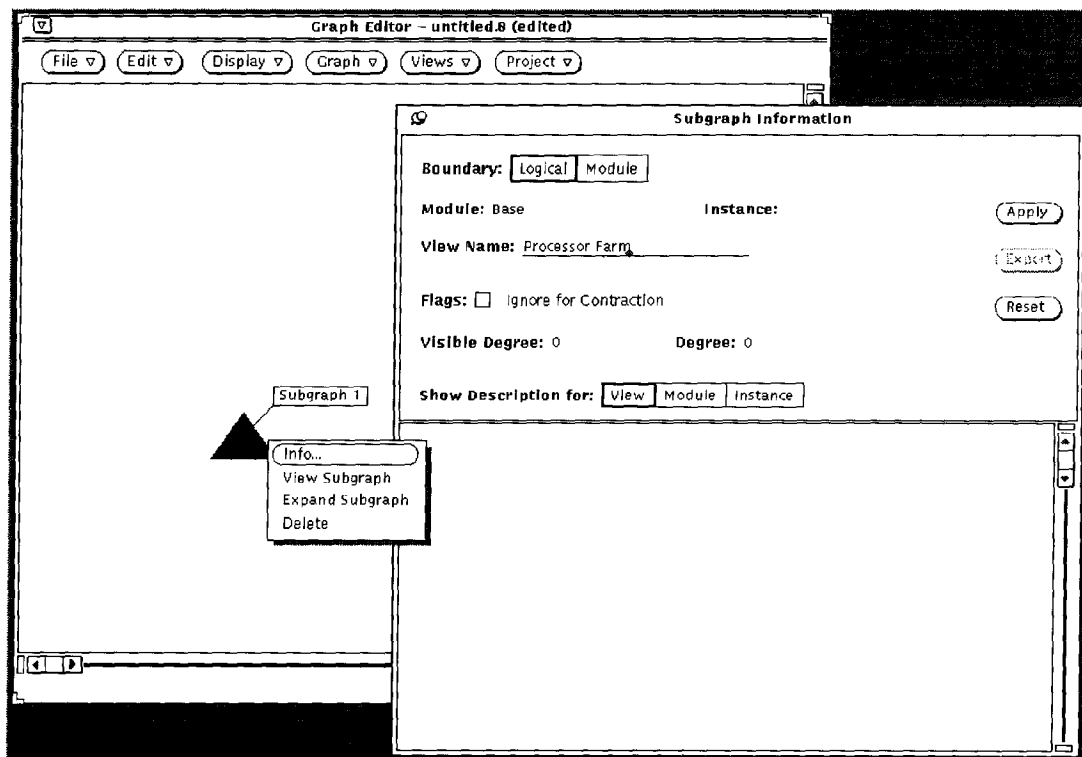


Figure 6.10 Processor farm converted to a view subgraph and the subgraph dialog

6.8 Instantiating a Reusable Module

Now, suppose that we wish to use this STM in an application. We can do this by creating an empty project (e.g., by using 'New' from the 'File' menu) and then selecting the module instance tool from the tool menu (the square icon in Figure 6.6). Clicking on the desired location on the canvas will cause the dialog in Figure 6.11 to be displayed. Selecting processor farm from the list and filling in the desired names will cause the icon in Figure 6.12 to be displayed. As noted earlier, modules have their own distinct menu attached to their icons.

As an application developer the interface options are the most important. The port bindings are important for connecting processes or other templates to this one.

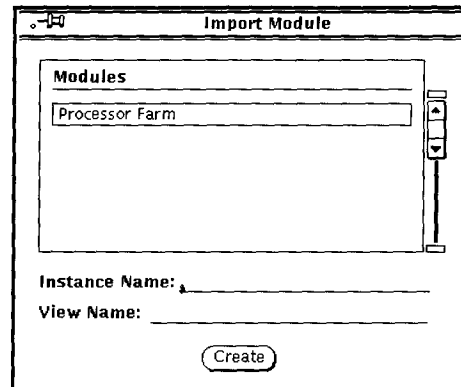


Figure 6.11 Import module dialog

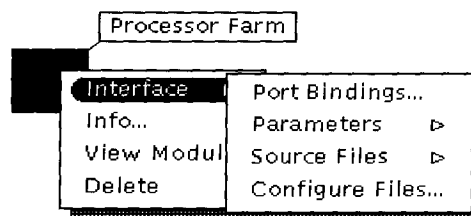


Figure 6.12 Module symbol and menu

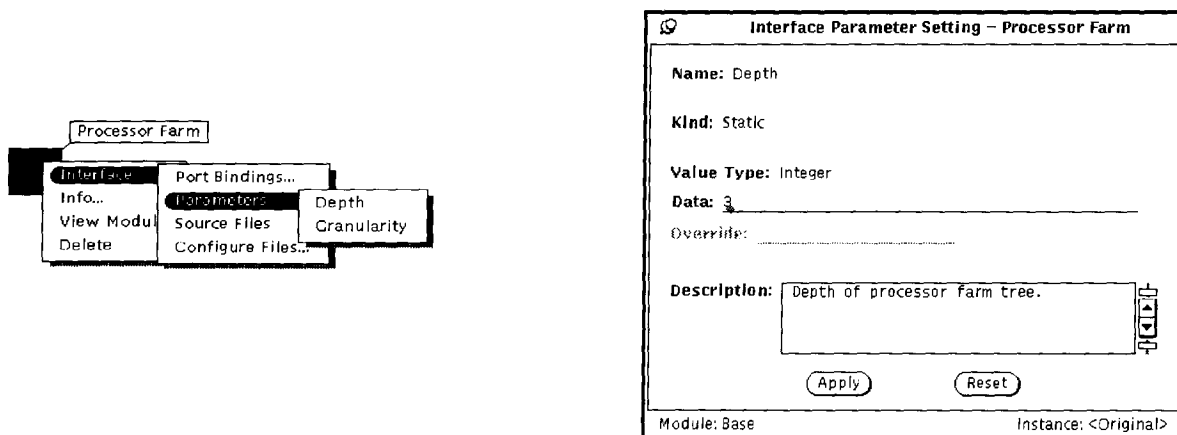


Figure 6.13 Interface parameters submenu and dialog

The parameters menu lists those module parameters created for the module and not declared external. This menu and the dialog accessed by selecting one of the parameters are shown in Figure 6.13. Note that only the parameter value is editable at the interface. This is also the main target of the object description since the person importing a module may have nothing to do with its implementation.



Figure 6.14 Editor window with interface file

The file menu lists those files declared as 'external' for the module in a menu just like those for processes shown in Figure 6.10. Insuring clean reuse at the specification stage insures no duplications here. Figure 6.14 shows one of these files as brought up by selecting it from the menu. The 'Configure Files...' option brings up a dialog very similar to the file specification panel of the creation information dialog except that it has only two modes. One is for adding libraries and one for source files. A user may utilize this dialog to change the path of a file supplied to him and thus get his own copy of template file or to attach additional source files to a particular interface file. These 'attached' files will be included in the make process everywhere the interface file is so that it will become a part of all of the same processes.

7 Conclusions

7.1 Synopsis

This thesis began by noting the need to reconcile the desire to provide simplifying abstractions with performance requirements. Virtual machines implementing parallel programming paradigms were presented as a type of abstraction which does provide significant simplification and can be made compatible with performance objectives. Compatibility is achieved by analyzing a virtual machine to determine the design and implementation attributes which can be tuned to improve the performance of a particular application of the virtual machine and then parameterizing the virtual machine in terms these attributes. We have proposed the skeleton-template-module (STM) as a parameterized virtual machine, corresponding to a particular paradigm, which a user can employ to create a particular application by filling its skeletal or template structure.

It was also noted that to be truly useful a construct such as an STM must be supported by and fully integrated into a viable programming environment. Such a system must support both the construction of STMs and their use to create applications as part of a general programming framework. Other objectives for such a system, such as “ease of use,” reuse, and scalability, common to both sequential and parallel environments, were also discussed.

We have designed and built such an interface and environment, along with prototype mapping and loading tools. The system utilizes the process-port-channel model of parallel program description and supports these primitives with a graphical interface. This interface provides arbitrary hierarchical organization within a program by the use of STMs and logical subgraphs within STMs. In addition, objects are provided to both support STMs and further

enhance “ease of use.” Some of these objects such as projects and modules are structural units of organization for aggregates of primitives. Modules also support the objective of reuse as does the process type object. The related objectives of scalability and performance tunability are clearly addressed by objects such as module parameters, parameterized graphs, and process groups.

7.2 Current Status

An alpha version of the user interface and database component of the environment implementing all of the major features discussed in this paper exists at this time. It consists of nearly of 30,000 lines of C and SQL source code for the Oracle V6 database. Prototype implementations of the mapper, loader, and runtime support systems have also been implemented. Our system also includes a slightly modified version of the Tmon monitor [JiWaCh90]. In addition to more advanced versions of these tools, initial versions of tools for multi-module scheduling and communication pattern debugging are currently under development.

The tools and database have been developed and currently run on Sun SPARC workstations under SunOS 4.1.X. However, they have been designed and implemented to be portable to any Unix platform with access to a high-performance SQL database. The runtime system is built on top of Trollius 2.1 running on a network of 70+ transputers and a SPARC host. We are investigating extending the runtime system to support networks of workstations by using PVM. The system is currently undergoing testing and has been used to implement the processor farm STM and a port of a many body simulation originally implemented directly in Trollius and using our original mapper, Tmap [Go91].

7.3 Future Work

7.3.1 Communication Support and Typing

There are a number of ways in which communication functionality can be simplified and enhanced. One obvious enhancement is providing a network data representation system such as Sun’s XDR [XDR87] or OSI’s ASN.1 [ISO87]. Doing so will solve several problems

which must currently be dealt with by the user. The first of which is the existence of different byte orderings on different systems. Even when using relatively homogeneous systems this can be a very annoying problem, and can be serious impediment to portability. The second problem is that of requiring the user to pack data structures into and unpack data from message data units. This is an inescapable task, but tedious and error prone. Data structure compilers which automatically create these routines are available for both of the mentioned data representation systems. Of the two, XDR is preferable in performance sensitive contexts like parallel programs since it carries much less overhead than ASN.1 [PaRo89].

Another potentially useful mechanism is providing some sort of typing for ports. This would provide a more explicit specification of communication and, thus, make a greater degree of structural validation possible. Furthermore, by tying port types to types declared with the data representation support mentioned above, most of the coding overhead involved in dealing with message passing systems could be eliminated. What are not clear are issues such as how strict type checking should be, whether multiple types should be supported on a port, or, if that is the case, whether time ordering information should be recognized as part of a 'port type.' The latter, for example, brings the whole problem into the domain of protocol specification and verification.

7.3.2 Data Decomposition

One of the persistent parallel programming problems is that of data decomposition and its complement, result aggregation. The nature of these tasks is such that they are highly application specific and, as such, are not able to be incorporated into the specification of a virtual machine or the implementation of an STM. In fact, two of the three most common pieces of code required to instantiate an STM are function implementing these two tasks (the third being a computation on the decomposed data). Reducing the amount of programmer effort involved in dealing with these tasks would, thus, be another major step in simplifying parallel application implementation.

There do exist some well defined standard decomposition strategies for some data structures, particularly arrays (e.g., [MeVR91]), although they are not nearly as general as virtual machines. However, even here there is the question of how these strategies can be presented to the user and parameterized into a relatively general form. Even more challenging

is the problem of how to deal with those situations where such strategies are not available or applicable.

7.3.3 Derived Objects

Although reusable objects such as modules and process types save large amounts of effort, both in construction and maintenance, a significant duplication of effort is still required to create types and modules that are almost, but not quite, the same. In the case of process types we had initially considered having fields for individual processes to override process type information to help address this problem. However, this solution is inadequate for precisely the same reasons that process types are needed in the first place – reuse and maintainability. The same variant may occur a number of times and the information defining the variation may be need to be updated at some point in the future.

As a result, we are now considering the concept of derived objects to solve this problem. A derived object is defined as a base object plus some set of differences. This is closely related to the concept of inheritance in object oriented systems. Applying inheritance this domain raises a number of questions regarding the semantics for different objects (some of which may be compositions of objects including derived objects), whether derivation chains should be permitted, and whether such modifications should encourage or discourage the relaxation of object scoping rules. Furthermore, any non-trivial definition would present a number of challenges with regard to how it can be efficiently implemented in an environment such as ours.

7.3.4 Advisors

Even with the best tools and environment for constructing parallel applications possible, converting a sequential application into a parallel application or writing one from scratch are unlikely to become trivial tasks. There remains a significant amount of highly specialized expertise involved in such tasks as identifying the best virtual machine for an algorithm, choosing a good data decomposition scheme, or determining the optimal value for a parameter. This is fertile ground for technologies such as expert systems and user modeling which can incorporate such expertise and use it to guide and advise the user.

Such a software ‘advisor’ was proposed as one of components of PIE [GrSe85]. However, the utilization of virtual machines which implement restricted models of computation

should assist and simplify such an effort since the restricted models clearly delineate different areas of expertise and the developer of a particular virtual machine is likely to be an expert with respect to its particular attributes. Also, using restricted models makes it more possible that 'expertise' can be reduced to analytical models instead of general heuristics and techniques.

Furthermore, while it may seem attractive, an interactive tool may not always be best or simplest solution. For example, as already supported by our environment, a virtual machine implementor utilize his knowledge to identify and parameterize performance sensitive aspects of an implementation and then write programs to analyze information from a monitor to compute the optimal values for these parameters. In such cases the application developer need never deal with or even be aware of the expertise at work making adjustments to improve the execution of a program.

References

- [AhCaGe91] D. Gelernter, S. Ahmed, and N. Carriero, "The Linda Program Builder," in A. Nicolau, D. Gelernter, T. Gross and D. Padua, ed., *Advances in Languages and Compilers for Parallel Processing, Research Monographs in Parallel and Distributed Computing*, MIT Press, Cambridge, Massachusetts, 1991, pp. 71-87.
- [Al91] G. Alverson, *Abstractions for Effectively Portable Shared Memory Parallel Programs*, Dept. of Computer Science and Engineering, University of Washington, PhD Thesis, Oct. 1990.
- [BaAl91] Ö. Babaoglu, L. Alvisi, *et al*, *Paralex: An Environment for Parallel Programming in Distributed Systems*, University of Bologna, Dept. of Mathematics, Technical Report UB-LCS-91-01, Feb. 1991.
- [BaCu87] D. Bailey and J. Cuny, "Graph Grammar Based Specification of Interconnection Structures for Massively Parallel Computation," *Proc. of the Third Int'l. Workshop on Graph Grammars*, Springer-Verlag, Berlin, 1987, pp. 73-85.
- [BaCuLo90] D. Bailey, J. Cuny, and C. Loomis, "ParaGraph: Graph Editor Support for Parallel Programming Environments," *International Journal of Parallel Programming*, Vol. 19 (2), 1990, pp. 75-110.
- [BeDo91a] A. Beguelin, J. Dongarra, *et al*, *Graphical Development Tools for Network-Based Concurrent Supercomputing*, Oak Ridge National Laboratory, 1991.
- [BeDo91b] A. Beguelin, J. Dongarra, *et al*, *HeNCE: A Users' Guide*, Oak Ridge National Laboratory, 1991.
- [BeSt89] F. Berman and B. Stramm, *Prep-p: Evolution and Overview*, Department of Computer Science, University of California at San Diego, Technical Report CS89-158, 1989.
- [Br85] J.C. Browne, "Formulation and Programming of Parallel Computations: A Unified Approach," *Proc. of the Int'l Conf. on Parallel Processing*, CS Press, Los Alamitos, Calif, 1985, pp. 624-631.

- [Br89] J.C. Browne, M. Azam, and S. Sobek, "CODE: A Unified Approach to Parallel Programming," *IEEE Software*, Jul. 1989, pp. 10-18.
- [BuPf88] G. Burns, A. Pfiffer, *et al*, "The Trillium Operating System," in *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, ACM Press, Jan. 1988.
- [Ca89] D. Gelernter and N. Carriero, "Linda in Context," *Communications of the ACM*, Vol. 32 (4), Apr. 1989, pp. 444-458.
- [ChGo91] S. Chanson, N. Goldstein, *et al*, "TIPS: Transputer-based Interactive Parallelizing System," *Transputing '91 Conference Proceedings, Sunnyvale, Calif.*, IOS Press, Apr. 1991.
- [Co89] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*, MIT Press, Cambridge Massachusetts, 1989.
- [EiBl91] R. Eigenmann and W. Blume, "An Effectiveness Study of Parallelizing Compiler Techniques," *Proceedings of the ICPP*, 1991, pp. 17-25.
- [EnWe91] A. Endres and H. Weber, eds., *Software Development Environments and CASE Technology*, Lecture Notes in Computer Science No. 509, Springer-Verlag, Berlin, 1991.
- [Epcc92a] Edinburgh Parallel Computing Centre, *PUL-TF Prototype User Guide*, EPCC-KTP-PUL-TF-PROT-UG 1.4, 1992.
- [FeSrWa92] D. Feldcamp, H. Sreekantaswamy, A. Wagner, S. Chanson, "Towards a Skeleton Based Parallel Programming Environment," *Transputer Research and Applications 5*, IOS Press, Apr. 1992.
- [Fl79] R. Floyd, "Paradigms of Programming," *Communications of the ACM*, Vol. 22 (8), Aug. 1979.
- [FoKe91] G. Fox, K. Kennedy, *et al*, "A Static Performance Estimator to Guide Data Partitioning Decisions," *Proc. 3rd ACM Symp. on Principles and Practice of Parallel Programming (PPOPP)*, Apr. 1991, pp. 213-223.
- [FoOv91] I. Foster and R. Overbeek, "Bilingual Parallel Programming," *Advances in Languages and Compiler for Parallel Processing, Research Monographs in Parallel and Distributed Computing*, MIT Press, Cambridge, Massachusetts, 1991, pp 24-43.
- [FrRe91] T. Fruchterman and E. Reingold, "Graph Drawing by Force-Directed Placement," *Software Practice and Experience*, Vol. 21 (11), Nov. 1991, pp. 1129-1164.
- [Go91] N. Goldstein, *A Topology Independent Parallel Development Environment*, Dept. of Computer Science, University of British Columbia, Masters Thesis, April, 1991.

- [GrSe85] F. Gregoretti and Z. Segall, *Programming for Observability Support in a Parallel Programming Environment*, Department of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-85-176, November 18, 1985.
- [Ha69] F. Harary, *Graph Theory*, Addison-Wesley, Reading Mass., 1969.
- [HeEt91] M. Heath, J. Etheridge, *et al*, *ParaGraph: A Tool for Visualizing Performance of Parallel Programs*, Oak Ridge National Laboratory, Technical Report.
- [HeGi91] M.T. Heath, G.A. Giest, *et al*, *PICL – A Portable Instrumented Communication Library*, Oak Ridge National Laboratory, ORNL/TM-11130.
- [ISO87] CCITT, Recommendation X.208, “Specification of Abstract Syntax Notation One (ASM.1),” Geneva, Switzerland, 1987.
- [JiWaCh90] J. Jiang, A. Wagner and S. Chanson, “TMON – A Transputer Performance Monitor,” in D. Fielding, ed., *Transputer Research and Applications NATUG 4*, IOS Press, Oct. 1990.
- [Ke92] K. Kennedy, “Software for Supercomputers of the Future,” *The Journal of Supercomputing*, Kluwer Academic Publishers, Boston, 5 (1992), pp. 251-262.
- [LC89] *Transputer Toolset*, version 89.1, Logical Systems, Corvallis, Oregon, 1989.
- [LeLi88] T. Lee and C. Lin, *ROPE User’s Manual: A Reusability-Oriented Parallel Programming Environment*, Technical Report, Dept. of Computer Science, University of Texas at Austin, 1988.
- [LeER92] T. Lewis and H. El-Rewini, *Introduction to Parallel Computing*, Prentice-Hall, 1992.
- [LeSeVr89] T. Lehr, Z. Segall, D. Vrsalovic, *et al*, “Visualizing Performance Debugging,” *IEEE Computer*, October 1989, pp. 38-51.
- [Lo89] F. Long, ed., *Software Engineering Environments*, Proc. of the Int’l Workshop on Environments, Chinon, France, Sept. 1989, Springer-Verlag, Berlin, 1990.
- [LoRa90] V. Lom S. Rajopadhye, *et al*, *OREGAMI: Software Tools for Mapping Parallel Computations to Parallel Architectures*, Dept. of Computer Science, University of Oregon, Technical Report CIS-TR-89-18, Jan. 1990.
- [MeVR91] P. Mehrotra and J. Van Rosendale, “Programming Distributed Memory Architectures Using Kali,” *Advances in Languages and Compiler for Parallel Processing, Research Monographs in Parallel and Distributed Computing*, MIT Press, Cambridge, Massachusetts, 1991, pp. 364-384.
- [Ne88] F. Newbery, “An Interface Description Language for Graph Editors,” *Proc. of the IEEE Workshop on Visual Languages*, 1988, pp. 10-12.

- [RaSeVr90] M. Rao, Z. Segall, and D. Vrsalovic, "Implementation Machine Paradigm for Parallel Programming," *Proceedings Supercomputing '90*, New York, NY, Nov. 1990, pp. 594-603.
- [OL89] Sun Microsystems, Inc., *Open Look Graphical User Interface Functional Specification*, Addison-Wesley, Reading, Massachusetts, 1989.
- [PaRo89] C. Partridge and M. Rose, "A Comparison of External Data Formats," in E. Stefferud, O. Jacobsen, and P. Schicker, *Message Passing Systems and Distributed Applications*, Elsevier Science Publishers B.V. (North-Holland), 1989, pp. 233-245.
- [Postgres92] *Postgres Reference Manual*, Version 4.0, 1992, available via anonymous ftp from postgres.berkeley.edu.
- [SeRu85] Z. Segall and L. Rudolph, "PIE: A Programming and Instrumentation Environment for Parallel Processing," *IEEE Software*, Nov. 1985, pp. 22-37.
- [SiScGr91] A. Singh, J. Schaeffer and M. Green, "A Template-Based Approach to the Generation of Distributed Applications Using a Network of Workstations," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2 (1), Jan. 1991, pp. 52-67.
- [Sn84] L. Snyder, "Parallel Programming and the Poker Programming Environment," *IEEE Computer*, July 1984, pp. 27-36.
- [Sn91] L. Snyder, "The XYZ Abstraction Levels of Poker-like Languages," in A. Nicolau, D. Gelernter, and D. Padua, ed., *Advances in Languages and Compilers for Parallel Processing, Research Monographs in Parallel and Distributed Computing*, MIT Press, Cambridge, Massachusetts, 1990.
- [SrChWa91] H. Sreekantaswamy, S. Chanson and A. Wagner, *Performance Prediction Modeling of Multicomputers*, TR 91-27, Department of Computer Science, University of British Columbia, Vancouver, Canada, Nov. 1991.
- [Su90] V. Sunderam, "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice and Experience*, Vol. 2 (4), December 1990, pp. 315-339.
- [Wa89] A. Wasserman, "Tool Integration in Software Engineering Environments," in F. Long, ed., *Software Engineering Environments*, Proc. of the Int'l Workshop on Environments, Chinon, France, Sept. 1989, Springer-Verlag, Berlin, 1990, pp. 137-149.
- [WiWo89] A. Wileden and A. Wolf, "Object Management Technology for Environments," in F. Long, ed., *Software Engineering Environments*, Proc. of the Int'l Workshop on Environments, Chinon, France, Sept. 1989, Springer-Verlag, Berlin, 1990, pp. 137-149.
- [Wi91a] G. Wilson, *Via: A Structured Message-Passing System for Parallel Computers*, Edinburgh Parallel Computing Centre, TR91-16, 1991.

- [Wi91b] G.Wilson, Seminar at the University of British Columbia, Dept. of Computer Science, 1991.
- [XDR87] Sun Microsystems, Inc., *XDR: External Data Representation Standard*, (RFC 1014), in Internet Working Group Requests for Comments, no. 1014, Network Information Center, SRI International, Menlo Park, California, Jun. 1987.
- [Va90] L. Valiant, "A Bridging Model for Parallel Computation," *Communications of the ACM*, Vol. 33 (8), pp. 103-111, 1990.
- [Za89] P. Zave, "A Compositional Approach to Multiparadigm Programming," *IEEE Software*, Sept. 1989, pp. 15-25.