CMU-CS-85-128

PIE - A Programming and Instrumentation Environment for Parallel Processing

Zary Segall and Larry Rudolph

Department of Computer Science

Carnegie-Mellon University

26 April 1985

Abstract

The issues of the efficient development of performance efficient parallel programs is explored. The Programming and Instrumentation Environment for Parallel Processing (PIE) system's concepts, designs, and preliminary implementation results are presented. The key goal in PIE is semiautomatic generation of performance efficient parallel programs. In PIE, a system intensive rather than programmer intensive programming environment is promoted for supporting users with different experience in parallel programming. Three levels of such support are provided, namely the Modular Programming Metalanguage, the Program Constructor, and the Implementation Assistant. In order to facilitate the task of parallel programming, each component employs a set of new concepts and approaches to integrate functionality with performance concerns. Some of them are: programming for observability, semantic monitoring, relational parallel program representation, constraint driven abstract data types (frames), frame unfolding, programming with templates, and parallel implementation assistant. This paper presents the results of PIE 1, the first of a three phase project.

This research has been supported in part by the Ballistic Missile Defense Advanced Technological Center under contract DASG-60-80-C-0057. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of BMDATC, Carnegie-Mellon University or the U.S. Government.

26 April 1985 at 11:21

Table of Contents

1. Introduction

A recurrent problem plaguing computer science is the underestimation of the software effort. Even from the earliest days, the hardware component of a major computer project has been viewed as requiring the most research and creative energy; only once the hardware is completed does the enormity of the software effort become apparent. The recent emergence of parallel processing is no exception. For example, an overwhelming number of research efforts have been directed toward describing various aspects of designing and implementing parallel processors, while there is a paucity of research addressing how one will actually program these machines.

Parallel processing is a radical departure from the traditional sequential processing. It promises machines that can execute extraordinarily large numbers of instructions per second. The main challenge now is not whether these machines can be built but whether they can be programmed in such a way as to make effective use of the increased computing power. It is not the raw computing power that is important, it is the effective computing power. Experience indicates that the future programmer of parallel processors will require assistance in order to make effective use of the machine and to efficiently produce efficient parallel programs. This paper describes one such environment, called the Programming and Instrumentation Environment (PIE).

Historically, the programmer of a parallel processor has been a rather knowledgable scientist or engineer who was burdened with the task of creating parallel applications using rudimentary environments. Detailed multiprocessor architecture and operating system knowledge as well as intricate ability to manually map parallel algorithms into a parallel virtual architecture and an "extended" sequential languages, are just some of the hurdles such users had to overcome. A correct parallel program is not even the end of the task. Often, the only reason for developing a parallel program is for the real time performance. The difficult task of performance debugging and the interpretation of the feedback in the context of a rudimentary program environment requires an even more specialized and highly knowledgable programmer.

The criteria of success for a parallel programming environment should be evaluated against two metrics. First, the user interaction with the system should put the burden on the system rather than on the programmer. That means, realizing a system intensive programming environment rather than a programmer intensive programming environment. Second, as noted above, efficiency in terms of the real speed obtained by parallelization of the problem, mapping it into and executing it onto a parallel processor, is the reason for applying a parallel solution to a computational problem. Accordingly, a programming environment oriented toward high performance parallel programs is of essential importance.

The first problem is not unique to parallel processing. The same goal is laudable in the context of sequential processing. Experience shows, however, that the increased number of dimensions present in a parallel program makes the need for system support urgent. Amplification and adaptation of current techniques from sequential programming environments to the parallel programming environment could be the first step in the way of satisfying this need.

The second problem is particular to parallel programming, and therefore, new techniques are required to fulfill the critical goal of efficient parallel programming. The PIE system attempts to address the above two problems and their related issues.

In PIE, we concentrate on those aspects relevant to parallel processing. It is worthwhile noting the distinction between parallel and distributed or concurrent processing. In concurrent processing, there is a set of processes whose executions are interleaved. The processes may share memory or may communicate via a message-passing mechanism. The emphasis is on correctness and only some slight improvement in execution speed is achieved due to overlapping I/O with computation. In distributed computing, there is a set of processes executing on distinct machines. Often the programming goal is to coordinate the activities that occur at physically-disjoint locations and these activities usually involve data-base applications. Occasionally, the goal of distributed processing is an increase in execution speed, however, since the communication overhead between processes is high, only a coarse grain parallelism can be supported. Communication is usually performed through a highly structured interprocess communication mechanism (IPC). In parallel processing there is a set of processes executing on a tightly coupled set of processors that either all share access to some shared memory or employ a very fast and efficient communication mechanism. The primary goal of parallel processing is a performance increase proportional to the number of processors. An orthogonal goal is reliability. In PIE, we concentrate our effort toward programming for performance efficient parallel programs.

The PIE project is organized in three phases. The first one, PIE 1 , is geared toward concept formation and feasibility study. It has been implemented using a number of VAX 11/780 computers and professional workstations. PIE 4, to be executed in a multiprocessor VAX 11/784 and MicroVAX workstations, is under advanced design and implementation. PIE 100, the third project phase, is geared toward the Supercomputer Workbench (a large multiprocessor system under development at CMU). Currently, PIE 1 is essentially complete and hence most of the material in this paper relates to the PIE 1 phase.

The paper is organized as follows. Programming environments and, in particular, parallel programming environment are first reviewed. The next three sections are concerned with the three components of the PIE system: the modular programming metalanguage, the program constructor, and the implementation assistant. The components are organized according to their level of abstraction and user support with the highest level providing expert system support for parallel program development. The concepts described in these sections are illustrated by a sample application presented in section 5. Finally, section 7 presents the conclusion and current status of the PIE project.

2. Background and Related Research

This section serves as a general introduction to programming environments. We review both sequential and parallel programming environments.

2.1. Sequential Programming Environments

At the foundation of a programming environment there are, in general, three types of components. First, a set of one or more programming languages is required . The programming languages may be of general purpose flavor or may support specific types of applications. Second, a program constructor facility provides an integrated interface between the programming languages, the user and the system utilities. Aside from supporting the individual user, the program constructor may facilitate multiple programmers' project support. Recently, there are notable efforts to enrich the program constructor's functionality with task specific information. Examples of program constructors are syntax-oriented editors and some recent efforts on semantic-oriented editors. Third, run-time debuggers of different flavors are essential for following and correcting a program's run-time behavior.

Currently, there are a few efforts aimed at building sequential programming environments. Although the emphasis is somewhat different in each of these systems, interactive system support for program development seems to be a common goal.

The Cornell Program Synthesizer [20] has a program constructor based on a syntax-oriented editor. Using attribute grammars to describe the program constructor generator output, this system could be used for a number of programming languages. The COPE [2] system, also from Cornell, uses an editor coupled with an error correcting parser to automatically correct users' program constructs, as they are entered. Undo, redo, editing and execution features provide a minimal connection and feedback mechanism between program development-time and run-time.

A major effort at CMU on programming-in-the-small and programming-in-the-large has produced the GANDALF programming environment [8]. The program constructor is again a syntax or language oriented editor - ALOE. The internal structure of programs in an ALOE is a parse tree, and the editor produced is template-oriented.

At Xerox PARC a number of programming environment prototypes have been explored. Notable is the Smalltalk environment [13, 5]. Smalltalk is object and message oriented and makes use of graphics to facilitate the manipulation of objects. The Smalltalk environment is language dependent and makes heavy use of the capability of interactive graphics display.

Another programming environment based on Smalltalk is CEDAR [12]. This system supports an algebraic, compiled language with interactive graphics-oriented programming interface using multiple windows.

Recently at Brown University, a graphic-oriented programming environment effort, PECAN [16], has been started. In PECAN, the emphasis is. on programming for observability, multiple views, and exploration of graphical programming.

2.2. Parallel Programming Environments

As previously stated, at the foundation of a programming environment is a class of programming languages. Almost all programming languages currently supporting concurrent processing (e.g. Ada, Modula2, CSP, OCCAM, Edison, Concurrent Pascal, etc.) provide constructs for sequential programming, as well as constructs for supporting concurrency control, communication and synchronization. When moving the domain of application into parallel processing, additional features are needed for achieving efficient mapping of parallel machines. This include constructs for:

- parallel entities manipulation
- virtual machine mapping
- efficient use of shared memory (when relevant)
- programming for observability
- performance/correctness debugging

Such support could be integrated in the programming language itself by extended the semantics, or could be provided as a set of procedures callable from the programming language itself. Given the fact that this support will most likely be language independent, an alternative is to provide the parallel language support in language independent manner. This support could be in a form of a metalanguage. The metalanguage is then the medium in which to express and manipulate all the parallel constructs. The sequential constructs stay in an existing programming language. When the programming language is not a concurrent programming language (ex. Fortran), the metalanguage covers synchronization and communication primitives as well.

Currently, there are a number of research efforts in the area of parallel programming environments. A distinct class is related to the vectorization of existing languages. Most notable is the Paraphrase [11] automatic vectorizing compiler research at the University of Illinois. Although well-suited for taking advantage of vector programs, the Paraphrase compiler does not provide support for creating new superior parallel software for a general class of computer structures.

Another interesting research effort is POKER at Washington University [19]. POKER provides a program constructor and is specialized to mapping a virtual application architecture into a configurable VLSI parallel architecture.

A set of relevant, programming environment efforts come from the area of distributed programming. ADL/ADS [6] developed by TRW is a language with an interactive graphic user interface, targeted toward mapping a distributed application into a distributed system. The language has constructs enabling the user to describe both virtual and physical system structure and to map a distributed program over the described structure. Significantly, an instrumentation system provides run-time information of the system behavior. Matchmaker [10], at Carnegie-Mellon University, is a program providing for distributed program construction by hiding some of the more cumbersome details of interprocessor communication from the user.

3. The Modular Programming Metalanguage

This section describes the modular programming metalanguage, which is the first of the three components of the PIE system. The other two components, the program constructor and the implementation assistant, are described in subsequent sections. The goal of the modular programming (i.e. MP) metalanguage is to provide support for the efficient manipulation of *parallel modules*, fast parallel access to shared data constructs, and *programming for observability* in a language independent fashion. The MP metalanguage allows the programmer to specify most of the code in his or her favorite programming language. Moreover, MP assumes a run-time environment that supports its abstractions and its monitoring directives. In this section, we describe the MP metalanguage that has been implemented in PIE 1.

3.1. The Composition of an MP Program

In MP, the programmer views the computation as a sequence of serial and parallel operations. Each parallel operation may consist of another set of computations that are executed in parallel. Since each computation itself can be composed of a sequence of serial and parallel operations, the programmer may not know how many processes are executed in parallel. To realize the goal of a clean separation of concerns, the programmer must be able to code or specify parts of the program without a detailed knowledge of the whole computation. To this end, a set of logical entities have been developed so that the programmer can concentrate on the correct functioning of each module; the system automatically converts the logical entities and their specifications into an executable parallel program. It is envisioned that additional optimization, both compile-time and run-time, of concurrency control conditions will minimize the synchronization overhead.

MP provides the ability to observe or monitor the execution of a program. This is accomplished through the *sensor* module. A sequence of serial operations is referred to as a *process* and a parallel operation is referred to as a *task*. A set of tasks that share the same protection envelope is referred to as a *task group*. Access and management of shared memory is specified in a *frame*.

3.2. Processes

A process is a sequential program written, at least in theory, in any sequential programming language. The process construct includes the set of local declarations, local data, the process body, and instrumentation. Sequential languages are extended by the MP environment to give support for synchronization, initiation of parallel operations, access to shared data, and monitoring directives associated with process functions. The motivation for these extensions is to allow the programmer to concentrate on the limited set of concerns at hand. For example, Figure 3-1 shows sample process code for a tree insertion routine. Note that the code is written without specifying locks or synchronization. These issues are relegated to the frame constraint code.

We have found it useful to code using frame operations that modify shared data structures based on the results of some test of shared data structures. In this way, concurrency constraints can be easily satisfied and the process code need not be concerned with the details of the shared data structures. For example, in the Insert process of Figure 3-1, a frame operation is used that adds an item to the tree only if the process is at a leaf node of the tree. The frame operation also indicates whether this occurred.

```
Process Insert(x:item) =
Begin
        $MoveToRoot;
        Repeat
                Switch $Compare(x) Begin
                        Case -1:  $MoveToRightChild;
                        Case +1:  $MoveToLeftChild;
                        Case  0:  Exit;
                End;
        Until $AddIfLeaf(x);
End
```

Figure 3-1: Process code for insertion in a binary tree. Note that the code is written without knowledge of other parallel activity. The *frame* operations, identified by the symbol "$", provide access to shared data structures.

3.3. Frames

Communication between processes on a tightly-coupled multiprocessor is often more efficient when shared memory is manipulated directly by the processes than when they exchange information solely through message passing, interprocess communication mechanisms (IPC). Experience has shown, however, that indiscriminate use of shared memory promotes 'hard to find' bugs. MP attempts to alleviate this problem by providing facilities for controlled access to shared memory. Separation of concerns is achieved through shared data encapsulation and shared abstract data types. MP thus introduces the notion of a *frame.* Logically, a frame consists of declarations of shared data, a set of operations on shared data, constraints on the execution of these operations, and monitoring directives associated with the shared data and its access. An interprocess communication mechanism (IPC), which is supported in many operating systems, is an example of frame operations. The IPC mechanism is usually implemented as a queue in which a send is a queue insert and a receive is a queue delete with the queue hidden from the caller and with a strictly enforced synchronization protocol. Although a frame is logically a single entity, in reality its operations are 'unfolded' automatically into the process code with appropriate coordination code where necessary. Moreover, frames give a clean extension to serial programming languages. From the user's point of view, access to shared data is accomplished through procedure calls that appear to the programmer as monitors [9, 3]. It is important to note that frames are not monitors; monitors provide strict mutually exclusive access to shared data whereas frames allow potentially unrestricted accesses to occur in parallel.

Frames consist of abstract shared data types, shared data encapsulation, and *constraints.* Associated with the data types are a set of operations, some of which are visible to the process code. The frame code for these operations may refer to any of the encapsulated data as well as to any of the internal manifest variables (explained later in this section in detail) such as the caller's process name or index described below. Frame operations may also refer to other frame operations. Figure 3-2 shows sample frame specifications for the tree insertion process presented in Figure 3-1. A local variable is declared for the use by each process. Visible functions and procedures can be used in process and other frame code.

Coordination and synchronization required for correct parallel access to shared memory is specified by constraints on the parallel execution of these operations. The constraints can be regarded as a generalized form of path expressions [7, 4] or path predicates [1] on the frame operations. These expressions specify both what sequences are required and what parallel actions are forbidden. Anything left unspecified is assumed to be unrestricted and may occur in any order or in parallel. The constraints can control access finer than just frame operations. For example, parallel access to different elements of an array may be allowed provided that there is no parallel access to any particular element and that no element can be referenced until it has been initialized.

Figure 3-3 shows sample constraint specifications for these operations. The operation names are listed with free or bound variables as parameters. The symbol "." indicates that the parameter value plays no role. The parameters following the operation name and placed between the brackets ("[" and "]") indicates which variables, of type either local or other, are required to match in order for the constraint to hold. In the example, each different *TreeNode* value has a separate constraint. The system can realize these constraints, for example, by automatically allocating a lock with each occurrence of a *TreeNode* and inserting the proper lock protocols.

Figure 3-4 shows the frame code unfolded into the process code. This is performed mechanically, and since the AddIfLeaf operation is exclusive on each node in the tree, the system automatically assigns a lock to each node. Moreover, from the specification, it is derived that the other operations can occur in parallel with each other but not with AddIfLeaf and hence readers/writers coordination is automatically inserted.

Divide and conquer, a common algorithmic paradigm, is supported by the MP environment. Here, a part of shared data is often subdivided with each subpart processed independently and in parallel. A frame partition operation and the notion of a hollow frame support decomposition into subproblems and allow the process code to be written without knowing whether it is solving a problem or a subproblem. The partition operation records how a shared array is to be divided. A hollow frame maps a part of a shared data array into an entire virtual shared data array. A particular instance of divide and conquer is when the subproblems are solved recursively. This style is supported by MP

```
Frame Tree Manipulation

Type  TreeNode = Record Begin
                        Item : int;
                        Left, Right : ^TreeNode;
                    End;

Local  P:^TreeNode;

{ Allocation of Shared Memory Data Structure }
Shared       Root : ^TreeNode;
                 RootLeaf : TreeNode;

Initialization:
         Begin
                 Root := RootLeaf;
                 Root.Item := Nil;
                 Root.Left := Nil;
                 Root.Right := Nil;
         End;

Visible Function $Compare(x : int): Boolean
Begin
         If x < P.Item Then $Compare := -1
         Else If x = P.Item Then $Compare := 0
         Else $Compare := 1;
End

{ Some operations are omitted }


Visible $MoveToLeftChild
Begin
         P := P.Left;
End;

{ The following operation modifies the tree, inserts the item, if we
are currently pointing at a leaf node. The results of this test are returned. }
Visible $AddIfLeaf(x : int): Boolean
Begin
         If P.Item != Nil Then Return(False)
         Else Begin
                 P.Item := x;
                 P.Left := New(TreeNode);
                 P.Right := New(TreeNode);
                 Return(True)
         End
End;
```

Figure 3-2: Sample frame specifications for the tree insertion process presented in Figure 3-1.

```
Constraint
Forbidden:   (ForAll t in {TreeNode})
                             $AddIfLeaf(.)[TreeNode: t]
                             with {$Compare(.)[TreeNode: t] ∪
                                          $MoveToRoot[TreeNode: t] ∪
                                          $MoveToLeft[TreeNode: t] ∪
                                          $MoveToRight[TreeNode: t] ∪
                                          $AddIfLeaf(.)[TreeNode: t] }
End Constraint
```

Figure 3-3: Sample constraint specifications.

and the run-time system maintains a hierarchical, dynamic, naming scheme for MP entities.

3.4. Tasks

A task contains all the necessary specifications for a parallel operation. A process wishing to initiate a parallel activity just invokes a task. A parallel operation consists of a set of processes that cooperate in some fashion, often through shared memory. A task therefore consists of a set of processes, a set of frames, and control information. In addition, tasks specify monitoring directives for the whole task as well as resource allocation. Resource allocation is usually dependent on the resources supplied by the target machine and often includes specifications as to how processes are to be mapped onto processors, what scheduling policies should be used, and how shared memory is to be mapped onto physical memory.

The MP metalanguage enforces scoping in a manner similar to that found in block-structured sequential programming languages. Part of the specification of a task includes process definitions, frame definitions (abstract shared data type declarations), and task definitions. Each logical entity can reference other logical entities that are within its scope. That is, entities that are defined in the same task or in an outer task can be referenced. (See, for example, Figure 5-1.) A library of predefined abstract shared data types and some standard tasks and processes are visible throughout the program and are arranged in system libraries in much the same way as are I/O packages.

3.5. Sensors

In addition to specifying the actions and constraints of a parallel program, MP also provides the ability to observe or monitor the execution of a program. This is accomplished by *instrumenting* a program by inserting *sensors* into the program. The implementation of the sensors is automatic and may be in hardware or software, depending on the system. MP supports four types of sensors: time consumed by a task or process, time required to execute a block of code, the value of a variable, and user defined. The first type is always inserted automatically by MP. Figure 3-5 shows the monitoring of a block of process code as well as the monitoring of the value of the parameter. The PIE system

```
Program Insert =

{ The frame's data structure declarations are unfolded into each process
and for each frame that it references. Note the lock allocation that has
been inserted with each TreeNode. }
Type   TreeNode = Record Begin
                          $Semaphore001 : Lock;
                          Item : int;
                          Left, Right : ^TreeNode;
                       End;

{ Access to shared memory is achieved in Pascal by allocating a pointer to a
record that specifies the shared memory structure. }
          SharedMemory = Record Begin
                          Root : ^TreeNode;
                          RootLeaf : TreeNode;
                          End;

var  P : ^Treenode;
         SharedMemoryPtr001 : ^SharedMemory;
         x : int;

{ The frame operations are unfolded into the Pascal code for the process
along with the inclusion of synchronization code realizing the constraints.}

Function $AddIfLeaf(x : int)
Begin
   $BeginWriters(P.$Semaphore001);
        If P.Item != Nil Then Return(False)
        Else Begin
                P.Item := x;
                P.Left := New(TreeNode);
                P.Right := New(TreeNode);
                Return(True)
        End;
   $EndWriters(P.$Semaphore001);
End
        . . .

{ Note that in the following body code that the frame operation has been
unfolded as well as the inclusion of synchronization code. }

                Switch $Compare(x) Begin
                          . . .

                        Case +1:  Begin
                                    $BeginReaders(P.$Semaphore001);
                                        P := P.Left;
                                    $EndReaders(P.$Semaphore001);
                                    End
                          . . .
                End;
```

Figure 3-4: Frame and synchronization code are unfolded into the process code.

integrates the job of monitoring and does all the associated bookkeeping. MP and the run-time system generate and maintain unique identifiers for each of the sensors as well as associating the point in the program specification where the sensors have been inserted by the programmer. Thus, when viewing the results of a particular sensor, the programmer can also see surrounding the code and specifications. In Figure 3-5, the block sensor requires the insertion of two sensors each with the same identifier.

```
Process Insert(x:item) =
Begin
~Begin_Process_Sensor(17);
        ~Variable_Sensor(12,x,"item");
        $MoveToRoot;
        ~Begin_Block_Sensor(7);
        Repeat
                Switch $Compare(x) Begin
                        Case -1:  $MoveToRightChild;
                        Case +1:  $MoveToLeftChild;
                        Case  0:  Exit;
                End;
        Until $AddIfLeaf(x);
        ~End_Block_Sensor(7);
~End_Process_Sensor(17);
End
```

Figure 3-5: Process code of Figure 3-1 with sensors inserted.
The `Begin/End_Process_Sensors` are always inserted and the 17 is associated with this process. The Block sensors were inserted at the direction of the programmer with the identifier 7 generated by MP and associated with this block.

3.6. Naming in MP

The ability to recursively invoke tasks and to create distinct instantiations of the various entities requires particular care be given to the naming issue. Each entity of type task, process, or frame, has three names: local, static, and dynamic. The local name is the one assigned by the programmer and is a single alphanumeric identifier. (For example, in Figure 3-1, the local process name is "Insert".) The static name is based on the static scoping of MP. It consists of a list of local task names separated by the delimiter "/" and terminated by the entities local name (For example, "Outer/Master/Slaves"). Since there can be multiple copies of a process, the dynamic local process name is the local name of the process followed by the delimiter "." and the index of the process instantiated (for example "slave.4"). The dynamic naming structure is similarly structured in a hierarchical fashion but reflects the sequence of task invocations leading to its instantiation. The dynamic name is a list of alternating task and process dynamic local names again separated by the delimiter "/" and terminated by the dynamic local name (for example, if "Outer" and "Master" are task names, and "Init" and "Slave" are local process names then

"Outer/Init.3/Master/Slave.5/Master/Slave.4" could be a dynamic process name).

3.7. Synchronization and Coordination

The problem of synchronization and coordination is a major cause of concern in parallel processing. In MP, the problem is mitigated in three ways: Tasks, Frames, and Events. Frames deal with synchronization and coordination details related to access to shared memory. Tasks deal with the initiation and termination of sets of processes. Although these two mechanisms are sufficient to handle all synchronization problems, a third kind of support is often useful. Specific execution points within a process are labeled and marked as events. Within the task specification, the user can specify constraints using these events in order to synchronize the processes within a task. These expressions or predicates are used to control processes as well as frame operations.

The constraints require the use of certain run-time status values. These *manifest variables* may or may not be actually maintained during run-time; their main use is to allow the programmer to form constraints in terms of these variables; the system may be able to realize the constraints in a simpler fashion. The manifest variables are similar to the *shadow* auxiliary variables introduced by Owicki and Gries [14, 15]. Some examples include: the `Process_Dynamic_Name` which is unique for each instantiation of a process and the associated `Process_Id` is a unique integer. The `Process_Index` can be derived from the `Process_Dynamic_Name` and indicates the unique instantiation of process when multiple copies have been initiated. The `Process_Depth` is an integer that identifies how many times a process has been recursively invoked. This value is derived by counting the number of times the process name occurs within the `Process_Dynamic_Name`. Finally, an event or execution count, `*_Count` where * is an event or frame operation identifier, is available that records the number of times the event or frame operation has been executed for each unique process instantiation Associated with each of these manifest variables is a set containing the universe of values for these variables.

3.8. MP Summary

Figure 3-6 reviews the basic "modules" of a task in MP. It describes the various specifications of a task as well as the other modules comprising a task. Note the distinction between definitions and references. A process may be defined in one task but only included in tasks that are defined within the task.

- Task
  - Definitions. (Statically scoped).
    - Process Definitions. A complete program in an extended serial programming language. Monitoring directives concerning aspects of the process state. Points in the program marked as events with associated names.
    - Abstract Shared Data Type Definitions. Describes the shared data declarations, the hidden and the visible operations, and the access control of these operations. A data initialization specification can also be supplied. Monitoring directives concerning the values of the shared constructs as well as monitoring the time spent waiting for synchronization.
    - Task Definitions. This definition provides static scoping of the logical entities.
  - Body
    - Processes. The set of processes that are to be executed in parallel when the task is invoked along with a repetition count indicating that there are to be multiple copies of the process.
    - Frames. Binds names to instances of the shared data types. This represents the data that is common to all processes and frames in this scope.
  - Control
    - Process Control. Generalized path expressions specifying initiation and termination conditions for the processes and for the task. Constraints are specified in terms of instances of the entities, global variables, and events.
    - Resource Allocation. Mapping process instantiation to processors: for each instance of a process either a specific processor is suggested, any one in a class of processors are suggested, or the system is free to allocate as it pleases. Where shared memory is to be allocated and whether or not the pages can be swapped is specified.
    - Monitoring Directives. Specifications as to whether any particular configuration of the processes should be noted. The system, by default, monitors the initiation and termination of each instance of a task.

Figure 3-6: A summary of the modules and their meaning for the specification of a parallel program in the MP environment.

4. The Program Constructor

The Program Constructor (PCT) is the second component of the PIE system. It provides a higher level of abstraction and support for the generation of efficient parallel programs. The PCT builds upon the concepts introduced by MP and frees the user from much of the detailed specification of an *instrumented* parallel program.

The PCT is unique in that it provides mechanisms and policies for combining and transforming both the development time (static) and the run-time (dynamic) information into meaningful program representations. The static information is combined with status and performance information to provide semantically meaningful performance feedback which, in our view, is the key for generating performance efficient parallel programs.

The PIE PCT can be decomposed into three main components:

- An MP oriented editor - MPOE
- A status and performance monitor
- A relational representation system

The transformation and integration of dynamic and static information is based on using the relational model approach. The MPOE internal representation is in the form of both a parse tree and relational model, with the former required for the syntax directed editor and code generation part of MPOE and the latter to convey static program information to the two other components of PCT .

The Status and Performance Monitor is also based on the use of the relational model [18, 17] to select, filter, and collect run-time status and performance information. Accordingly, the relational model will contain static and dynamic information along with the semantic and temporal relations between them. The Relational Representation System then extracts and selects the views relevant to a specific performance or status goal.

The rest of this section describes each of these components in detail.

4.1. MP Oriented Editor - MPOE

The user (or programmer) of the MP environment, specifies and codes the parallel program through a syntax directed editor. While the study of structured editors is still a very active area of research, the PIE system requires only a few salient features that are common to most of them. The editor is the interactive interface between the user specifying a solution to an application problem, the extended programming language, the system utilities, and the rest of the PIE system. Our current implementation employs a GANDALF-type of syntax-directed editor [8] refered to as MPOE. The user is presented with a partially constructed parse tree consisting of nonterminal and terminal nodes. As the user enters program information, the nonterminals are expanded, the parse tree is refined, and some semantic action affecting the contents of the relational model may occur. For example, when positioned at the nonterminal node for the identifier of a process module, the name entered by the user causes the parse tree to be further refined, the screen updated, and an addition to the relational model indicating the existence of this process along with other information such as where in the parse tree the specification occurred.

The MPOE consists of two parts or grammars, one for describing and specifying most of the modules of MP and one for each of the programming languages in which the process code can be specified. The root of the parse tree is a task and the children are (i) the process and frame definitions, (ii) the process and frames instantiations that are the run-time composition of the task, (iii) the control specifications, (iv) the task monitoring directives, and (v) the tasks that are within the scope of this task.

In MP, separation of concerns is achieved through the use of many modules. Although this allows the programmer to concentrate on only a narrow range of program aspects, a typical program may contain a large number of instances of these modules whose maintainance and organization may become a problem. The static scoping rules of the definitions helps to isolate disjoint parts of the program; entities not within the outer scope can not be referenced. The MPOE enforces this restriction in an interactive fashion. The MPOE, however, does not just tell the programmer when he can not reference an entity; it also helps the programmer to see what entities are available at each point in the specification. A roadmap of the program that dynamically displays the scoping structure of the modules is part of the graphical user interface.

As each module is specified, the MPOE notifies the rest of the PIE system. This causes an update to the PIE central relational model and may also update some of the graphical views of the program specification. A link is created between the user's source code or specification, the internal objects of the PIE system, and the final executable code. This is useful, for example, when the user queries the result of a monitoring sensor. At the same time as the value is related to the user, the PIE system can also present the user's specification of the sensor and its surrounding context in MP.

The MPOE also provides a mechanism of selective viewing. It is often the case that event specification within the process code or the monitoring directives are intrusive and clutter the users view of the program specifications. Too many events or monitoring directives make the control flow of the program hard to follow and often make the display of a simple procedure span multiple screens. Similarly for frame operations that have long static names and make the display of statements span multiple lines. The user can request that event and monitoring specifications not be displayed and that frame operation names be shortened.

Finally, syntax-directed editors have much knowledge about the program as it is being specified. This knowledge can be used with another component of the PIE system in order to give advice to the user about the specification. For example, in a Master/Slave implementation, the knowledge of the entities of the partially specified program enable the Implementation Assistant, described in the next section, to advise the user about where to place the code to handle fatal exceptions raised by a slave process.

4.2. The Status and Performance Monitor

The development of a correct and efficient parallel program is accomplished through a series of phases such as source editing and specification, compilation, correctness and performance debugging, and optimization. In PIE, the role of monitoring is to observe the later phases of the program development and to provide significant information to the user or to the system. The monitoring environment is composed of sensors, monitoring directives, and presentations of sensor recordings. Sensors are currently implemented mainly in software although research is underway to build hardware sensors. There are two types of sensors, system and user defined. The user specifies monitoring directives throughout the MP specifications indicating which of the system sensors are to be enabled and the composition of the user defined sensors. After the parallel program is executed, the results of the sensors can be seen through various graphical views that are under the user's control.

The monitoring "philosophy" in PIE is based on the insertion of special objects, named sensors, into the user code during program development. It has the following main characteristics:

- It detects a precisely defined event, which may be the execution of a statement, a procedure call, an access to a variable, or a more complex, user defined function of the computational status of the program.

- It causes a transfer of information from the program to an external environment in an automatic and controlled way.

- It is transparent to the correctness of the program and makes every attempt to be transparent to the performance of the program.

- It can be enabled or disabled under the control of a standard, external mechanism. This allows a standard and possibly automatic instrumentation of a program at development time and a customization of monitoring requirements at run-time.

Sensors are inserted into the application code in a controlled way, using the MPOE which is also responsible for storing all the information about their names, type, and location in a central relational database. The run-time environment also records the information gathered into the relational model. The user views the information from the relational model in a graphical and interactive fashion.

Logically, a sensor is composed of three parts: Detection, isolation, and notification. There are four types of sensors: Start/stop object, start/stop block, variable monitoring, and user defined. The first two record the spatial and temporal status of the starting or stopping of a task, process, frame, or block of code within a process or frame operation. The variable monitoring sensors record the contents of variables at run-time. Different sub-types are provided for each basic data type of the language. The user-defined sensors allow for greater flexibility and filtering. The user provides a set of routines that are executed whenever a sensor is enabled and invoked. There is a standard format for transferring information to the relational model.

The ability to instantiate multiple copies of a task or process and, in particular, the ability to recursively instantiate tasks makes the enabling/disabling of sensors difficult. Sensor names are based on their location (i.e. the name of the logical entity within which they were specified) and have static and dynamic names. Sensors can be dynamically or statically enabled. A sensor enable table (SET) is maintained which is checked each time a sensor is encountered. The table contains both dynamic and static names as well as a specification of a class of names based on a wildcard specification. Dynamic enabling of a task is accomplished by inserting appropriate commands in the user's application code. The static enabling of sensors is most easily performed by first executing the application with all system supplied sensors enabled and then pointing to parts of the views of the computation to enable or disable sensors nested deep within the computation.

Based on previous experience with parallel processors, the monitoring system is the conceptual backbone in providing the compound event observability, a critical issue in the process of efficient parallel programming. The PIE system is angling towards a prototype parallel processor that will dedicate a substantial part of the multiprocessor hardware specifically to instrumentation. Main expected advantages of hardware instrumentation are non-intrusiveness (minimal cost or zero cost associated with detection, isolation, composition and notification of an event), wide range (from low grain hardware events to large grain events), and adaptability through programmability. These aspects provide the incentives for intensive use of instrumentation not only in performance evaluation, but also in operating systems design and programming environment. Such a parallel processor (i.e. the Supercomputer Workbench) is under development at CMU.

4.3. The Relational Representation System

We wish to give the parallel program developer all the help possible. The MP metalanguage provides support for specifying the program. In order for the developer to produce a correct and efficient program, the PIE system supports the notion of *programming for observability*. Multiple representations of the syntactic and semantic information are made available at both development-time and at run-time and are supported in an integrated fashion.

The syntactic MP information is maintained in the MPOE parse tree. The modules defined in MP, such as Process, Frame and Task, are directly visible and can be manipulated through MPOE. The MPOE parse tree contains the relevant development-time instrumentation information, such as sensor location, code, and associated data structures. In order to integrate the specification with the rest of the PIE system, a relational model (from data base domain) is used. Information from MP as well as information accumulated during run-time is maintained in the database. When relational operations are applied to the relationally structured information, multiple views of the same information are obtained. These views all help the programmer to understand the subtle interactions and bottlenecks of the program.

A representation is a form of expressing a parallel program and/or its behavior. Examples of representations are:

- Communication graph and communication tree
- Process graph and process tree
- Process-resource graph and process-resource tree
- Dependency graph and dependency tree

The graph presentations are the development-time representation, whereas the tree refers to the same representation at run-time. For example, the specification view of a recursive, divide-and-conquer application will be a graph with self-loops indicating recursive invocations whereas, the dynamic run-time view will be a tree with each node representing a particular instantiation of a module. A representation integrates the graphical screen presentation control together with the behavior and observability information (e.g. selective monitoring of relevant events).

In PIE, the MPOE parse tree is dynamically restructured into an INGRES-like relational model as the user performs the MPOE program development actions. At run-time, a relational monitor will produce relationally organized monitor information, stored in the same relational data structure. The relational data structure integrates the development-time information and the run-time information. By using the view feature of the relational model, along with presentation control information specific for each representation, a number of different representations can be displayed.

The integration of development-time and run-time information in one representation provides the basis for the performance/correctness debugging process. This is accomplished by providing:

- Selective representation. The user can selectively observe compound events by specifying a representation definition relevant to a performance or correctness goal.
- Integration between development-time and run-time information. The user is getting usable feedback which is directly applicable to the process of program development and performance tuning.
- High-level and automated performance debugging. The user describes the parallel program and the required representation at high levels of abstraction. The system is generating all low level interactions.

The PIE system currently provides four views: roadmap, process execution times, invocation tree and frame usage tree. The complete filtering envisioned has not yet been incorporated into the system. It is expected that the user will be able to interact with the graphical views and indicate the objects that are not to be displayed. A major problem in the graphical presentation of the actions of a parallel program is the large amount of information that must be sifted through until the appropriate information is found. The relation model approach should make this job easier.

4.3.1. Roadmap

This is a view of the specification of the program. As the user specifies a task, process, or frame with the MPOE, a corresponding object is displayed on the screen. The view also communicates with the MPOE so that a user can point to an object in the roadmap view and the MPOE will position the user to that part of the code. In this way, the user can get the "big picture" of the whole prgram structure while concentrating on only a small part of the specifications.

4.3.2. Process Execution Times

This view displays the execution times of processes in the form of a bar graph with a separate indicator for each instance of a process. The graph is dynamically drawn with the total execution time of the experiment divided into a number of "slots". For each slot during which a process is active the bar is blackened. The analysis of the database proceeds chronologically through the execution of the experiment giving the global effect of a visual tracing of the execution of all the processes in "slow motion". The bar graph gives an immediate global view of the parallelism of the application as a function of time. A useful feature is the ability to *zoom* into part of the view in order to see a finer grain of interaction.

4.3.3. Invocation Tree View

This view represents the dynamic invocation sequence of tasks, processes, and frames of the experiment. This view can be either static or dynamic. In the static case, each object remains in view once it is created and in the dynamic case, the objects vanish from the screen once they terminate. As an object is created, it is displayed on the screen and a line is drawn indicating the parent/child relationship. Once again, the view is displayed in "slow motion" thus presenting a motion picture as the computation unwinds. At any point, the user can select an object and in another window see the MPOE specifications of the object.

4.3.4. Frame Usage View

This view displays the dynamic reference to shared memory by processes or other frames. The actual implementation is part static and part dynamic. Frames and processes are displayed statically. Once again, the view proceeds chronologically through the database and as each reference to a frame is performed a line is drawn connecting the two entities. This view helps discover bottlenecks due to contention to shared objects.

5. A Simple Example

In this section we demonstrate various features of the PIE system by describing an example application. A naive parallelization of quicksort is the chosen application due to its familiarity and simple recursive structure. Quicksort starts with a set of numbers, divides the set into two subsets, and then recursively sorts them. The subsets are formed by choosing a candidate median value and placing all elements smaller than this value into set SMALL, those equal to or greater than into set BIG. The naive parallelization recursively applies quicksort to the subsets SMALL and BIG in parallel.

There are a few MP modules that are specified: a process either sorts the array if it contains only a few elements or divides the array into subsets and recursively applies quicksort to each subset. The process code can be written in such a way as to not know if it is executing in parallel with another process. Each process accesses the array through frame operations. A hollow frame is specified that makes part of the array (either the set SMALL or the set BIG that was created by the parent task) appear to be the whole array. After the process rearranges the array into SMALL and BIG sets, it partitions the array and invokes the task QT. Task QT contains the hollow frame and two copies of the process. One copy of the process is mapped onto the set SMALL and the other is mapped onto the set BIG. The hollow frame makes this distinction based on the index of the processes and access the data through the outer levels frame operations. Figure 5-1 shows the outermost part of the MP specification of the quicksort program. There is a master process that reads the items to be sorted and places them into the shared array through the operations provided by Frame global. No constraints were supplied so that the master process will not wait for the parallel task it invokes to complete. Also for simplicity, no extra task sensors have been inserted, however, the system will supply a set of default sensors for the tasks.

```
TASK quick

    PROCESS m DUPLICATION 1
        >Body<

    FRAME global
        >Body<

    TASK QT
        >body<
```

Figure 5-1: Part of the MP Specification of a naive parallel quicksort. The definition and reference to the processes and frames are combined.

In Figure 5-2 the inner task specification is presented. There are to be two copies of the process q instantiated whenever task QT is invoked.

```
TASK QT

    PROCESS q DUPLICATION 2
        >Body<

    FRAME shared
        >Body<
```

Figure 5-2: Part of the MP Specification of a naive parallel quicksort. This is the recursive quicksort task definition.

In Figure 5-3 some of the frame definitions are shown. The variables Low, Hi, NewLow, and NewHi are used to keep track of the remapping of the array. The first two record how the array was partitioned by the invoking process and the latter two are used to record the information of the partition operation that may be executed by the processes. The ReadArray and WriteArray operations are used if a process decides to directly sort the array and the other operations are used for division of the elements of the array into two groups. The initialization code makes use of one of the *manifest* variables recording the current level of the recursive task invocation. If this is the first level the the partitioning information is not needed.

```
FRAME shared

{ No Comment }


TYPE
    PartitionInfo = ARRAY[NProcesses] OF ArrayIndex;
    NewPartitionInfo = ARRAY[NProcess, 1 .. 2] OF ArrayIndex;

VAR
    Low, Hi : PartitionInfo;
    NewLow, NewHi, NewPartitionInfo : NewPartitionInfo;
    size : INTEGER;

VISIBLE FUNCTION $ReadArray(i : INTEGER) : INTEGER;
{>Documentation<}
    >Body<
VISIBLE FRAME PROCEDURE $WriteArray(i : INTEGER);
{>Documentation<}
    >Body<
VISIBLE FRAME PROCEDURE $Partition(MidPoint : INTEGER);
{>Documentation<}
    >Body<
VISIBLE FUNCTION $CompareAndSwap(i, j : INTEGER) : INTEGER;
{>Documentation<}
    >Body<

  *** Initialization of Data ***

BEGIN
    IF $Frame_Depth <> 1 THEN
        GetParentPartition(Low, Hi, Size)
END.
```

Figure 5-3: Part of the MP Specification of a naive parallel quicksort. This is the Frame definition part of the recursive part of quicksort. The Read and Write operations are for the Bubble Sort Part.

Figure 5-4 shows the roadmap view of the program. This is the definition scoping view of MP. As each module is specified in the MPOE, the roadmap view is updated. The user can point to a part of the roadmap with the mouse and MPOE will automatically position the cursor at the corresponding point in the parse tree.

Figures 5-5, 5-6, and 5-7 show the dynamic invocation tree indicating the processes invoking the tasks. Note the dynamic names of the MP entities as opposed to the static names in the roadmap view. In the first figure, a frame module has been highlighted indicating that it has been selected. As a result, another part of the display will show the specifications associated with the selected frame.

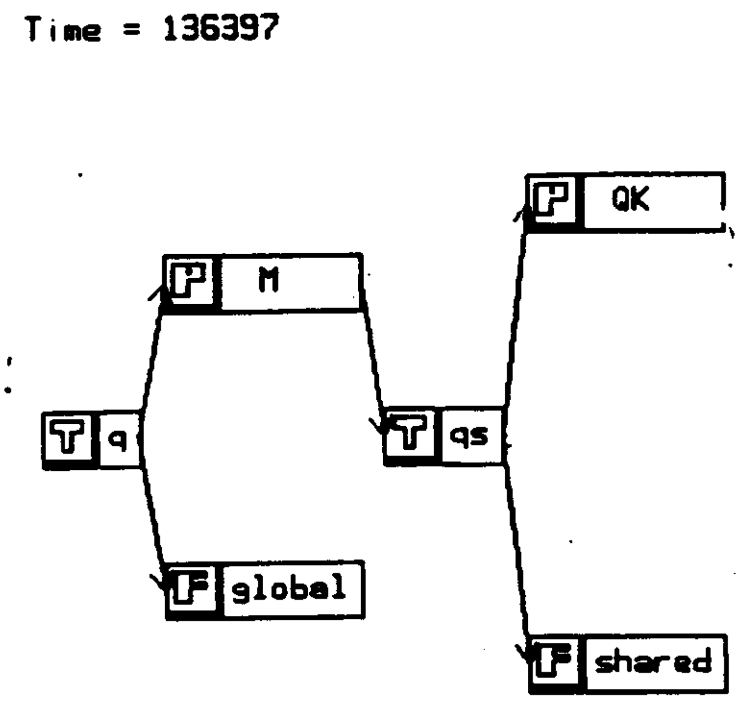

Figure 5-4: The Road Map View of Quicksort

Figure 5-8 shows a view of the frame usage graph. This view shows that at time 29733, one of the processes is accessing the frame. As time progresses, the view is updated and other processes and frames will be added. Frames are always placed on the left and processes on the right. THe view does not indicate frames referencing other frames.

Figure 5-9 is the process execution time bar graph. Since the application has been implemented on a Vax 780 using a message-passing scheme to simulate shared memory, the results are somewhat surprising. The times of no system activity are due to the operating system overhead initiating the processes and frames. The first recursive invocation of the task require a extraordinarily long time due to contention to the array. As the execution continues, the data is almost all sorted and there are not as many references to it.

6. The Implementation Assistant

The Implementation Assistant is the third PIE component. The goals of the Implementation Assistant are to provide semantic support in the parallel program development cycle. Specifically, the goals are:

- Parallel program performance prediction before extensive implementation investments.

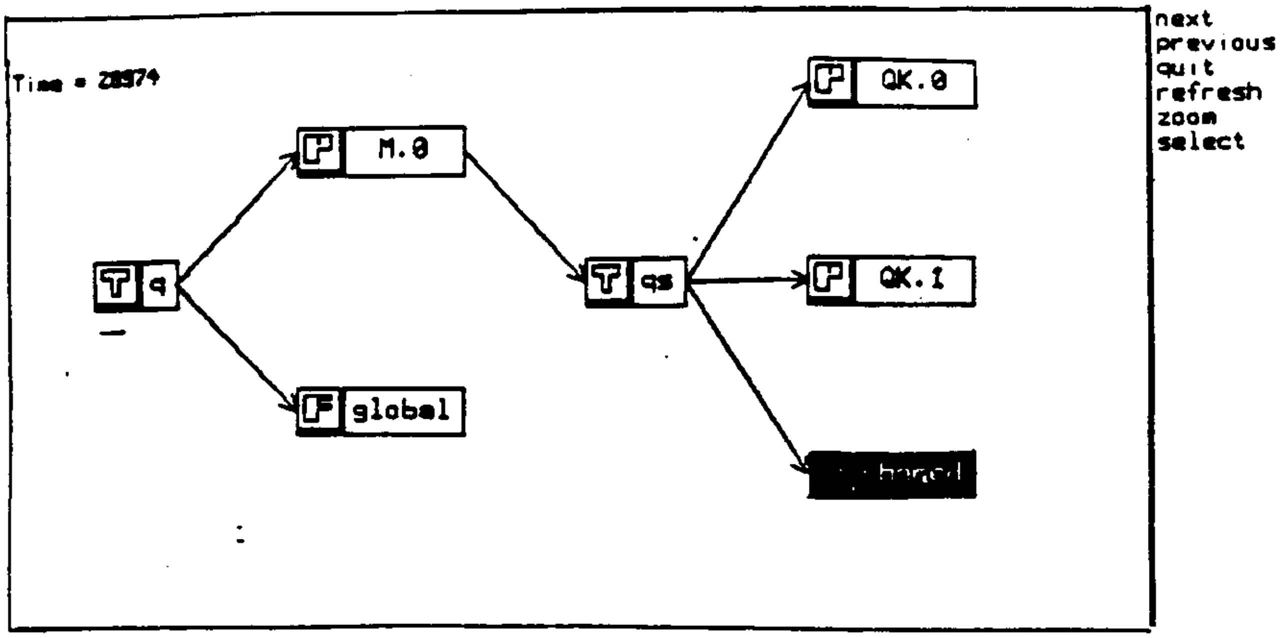

Figure 5-5: The Dynamic Invocation Tree - An Early View

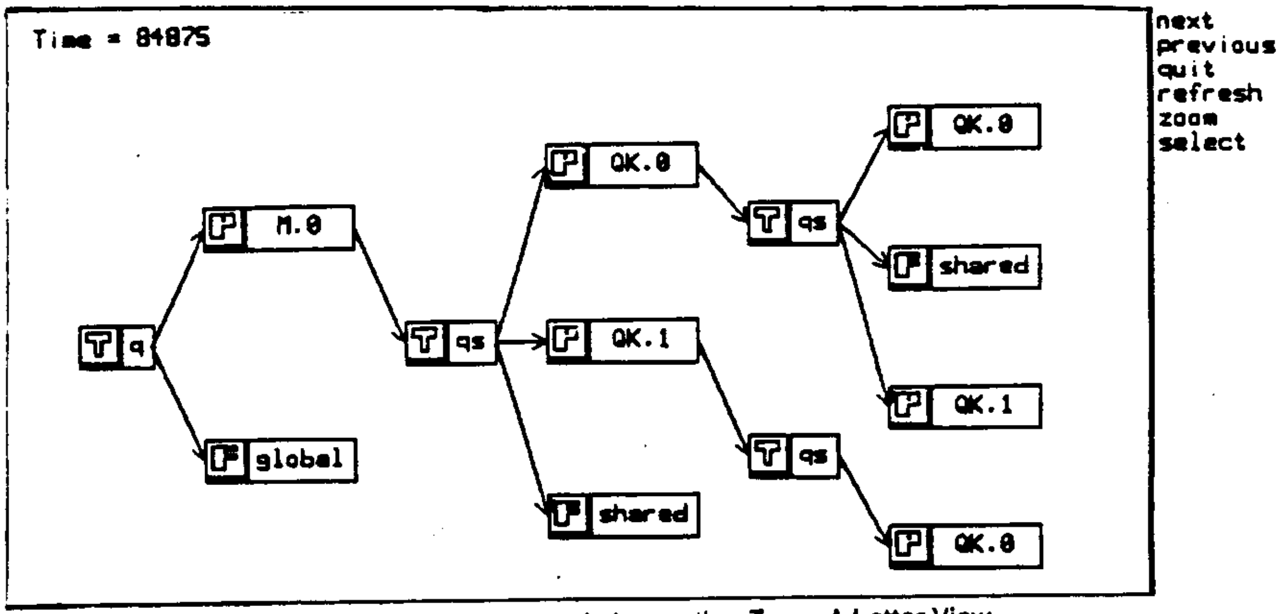

Figure 5-6: The Dynamic Invocation Tree - A Latter View

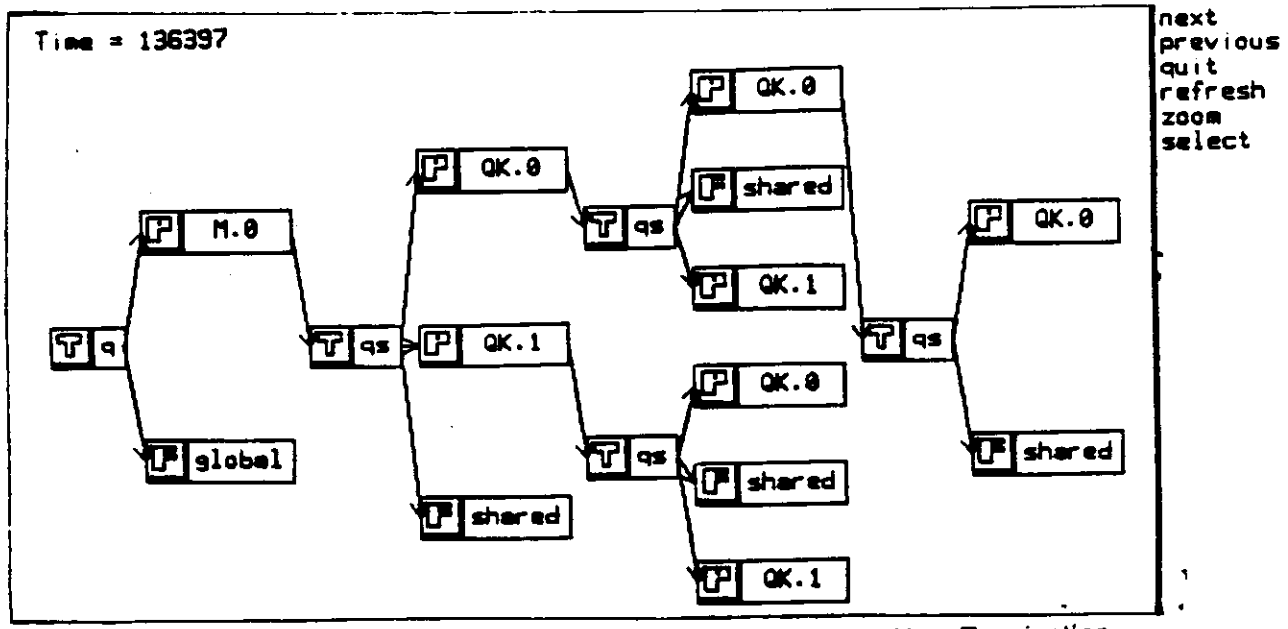

Figure 5-7: The Dynamic Invocation Tree - A View Near Termination

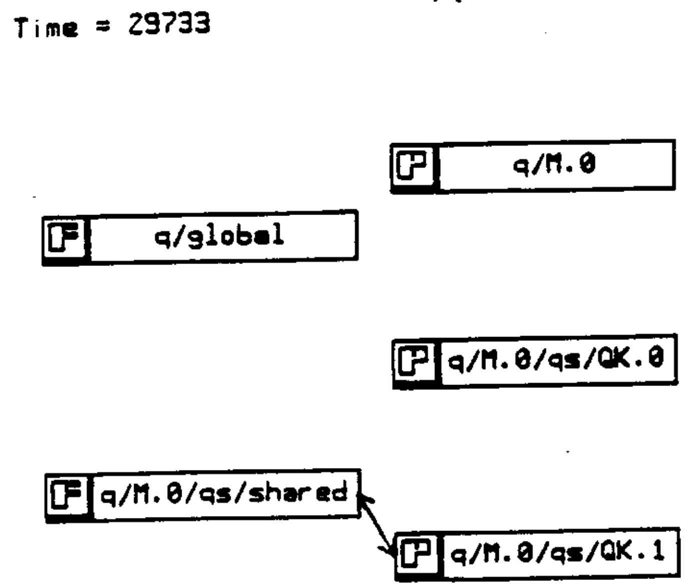

Figure 5-8: A Frame Usage View

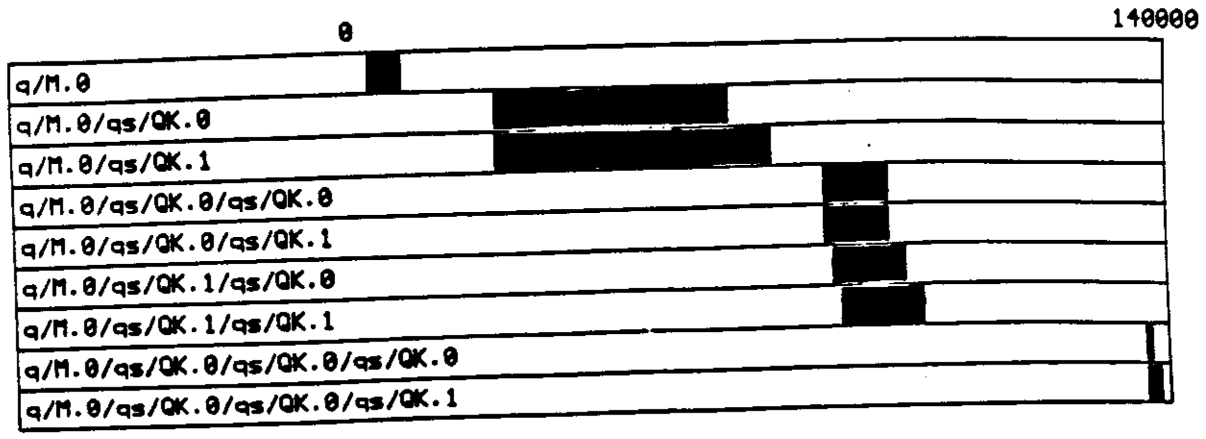

Figure 5-9: The Bar Graph View of Quicksort

- Assist the user in choosing between implementations.
- Semiautomatic parallel program generation by supporting a set of well-defined implementations.
- Support for semiautomatic semantic instrumentation.
- Support for a set of predefined representations related to the provided implementations.

An implementation, in the context of a parallel program, is a specific way to decompose the parallel computation processes and data, as well as the way they are controlled. Examples of parallel implementations are:

- Master-Slave
- Recursive Master-Slave
- Heap organized problem
- Pipe-line
- Systolic multi-dimensional pipe-line

The high level of abstraction of the implementation concept, allows one to characterize classes of problems by their relative performance/implementation measure. The performance/implementation ratio refers to the parallel application performance, when implemented using two or more relevant implementations. For example, a parallel program for a molecular dynamic problem could be implemented as a Master-Slave or as a systolic pipe-line. When applying a parallel solution to the problem, the user is interested in finding out what is the most suitable way to partition the problem data and decompose the computation in parallel processes, in the context of the architecture and operating systems of a specific parallel machine.

Encapsulating the knowledge pertinent to a set of frequently used implementations in the Implementation Assistant is, in our view, a first step toward fulfilling the above desiderata. The separation of concerns between implementation and algorithm allows, aside from programming in the context of predefined implementation semantic knowledge, performance prediction before full program development. In this case, the performance debugging process is supported by providing specific representation per each implementation.

The implementation performance prediction step is supported in PIE by the iteration model [21, 22]. An application is described in terms of its computation intensive and communication intensive behavior. The model allows the user to specify the process decomposition, the data partition and virtual parallel architecture characteristics. Currently, the model is limited to classes of application which are not heavily data dependent. A performance/implementation measure could be predicted per each relevant implementation. The user, assisted by an Implementation Discriminator, could then choose a suitable implementation.

As an implementation is chosen, the precoded structure (in terms of control, communication, synchronization, process decomposition and data partition) is made available to the user in the form of a modifiable template. Under the assistance of a specific Template Assistant, the user can fill in the template or alter it. From previous experience, we infer that in most cases, the user will wish to employ one of the provided implementations.

For illustrating this concept, consider an application to be implemented as a Unidirectional Pipeline (UPL) template. Such an environment will start by presenting the user with the UPL.Master template (see Figure 6-1)

```
UPL.Master <name>

        Input

        Pipeline <repeat>

        Output

        Terminate <terminating.agent>

End UPL.Master
```

Figure 6-1: The UPL.Master Implementation template

The components of the UPL.Master template are the pipeline processing nodes (Input, Pipeline, Output) and the termination condition agent. The terminating agent will indicate the processing node in charge of the termination condition (with the default agent being the User Interface).

The UPL.Master template is controlled through a UPL oriented editor (UPLOE). The Pipeline processing node could be split into multiple Pipeline processing nodes, each one specifying different code and a different replication count.

Under UPLOE control any one of the processing nodes template could be displayed. For example, Figure 6-2 shows the template obtained by selecting a Pipeline processing node.

```
Pipeline <name>

        >body<

        receive.west <west.buffer.name>

        >body<

        send.east <east.buffer.name>

        >body<

        send.terminate <terminate.buffer.name>

        >body<

End Pipeline
```

Figure 6-2: Pipeline Processing Node template

The Pipeline template provides send/receive operations to the process' adjacent neighbors and a send terminate communication with the terminate node. The user has to provide only the sequential application code. Once all the nodes have been defined, the system will automatically generate all the process management, memory management, synchronization, communication, instrumentation, and resource allocation required for the UPL implementation. Moreover, the implementation being semantically instrumented, specific representations relevant to the performance debugging of UPL are available at the user interface level.

In summary, by programming in the context of a preprogrammed template, the user expertise could be limited to its own application. The system is automatically generating most of the support required for parallelization of the application. One result in this context is the availability of the preprogrammed semantic instrumentation support. Implementation specific representations are made available at both development-time and run-time. The task of parallel programming and performance debugging is greatly simplified and facilitated.

7. Status and Conclusions

The issues of performance efficient parallel programs and system support for parallel programming are the main focus of the PIE system. The research method is pragmatic and is oriented toward practical results. Currently, the PIE 1 phase is mostly done and feasibility studies of the main components are completed. The MP Metalanguage has been defined. The prototype MP Metalanguage System is currently supporting Pascal. Other languages are under consideration. The Program Constructor components, namely the MPOE, the Status and Performance Monitor, and the

Relational Representation System are in the prototype form and are experimentally operational.

Sample examples have been run through the PCT, and some of the results are presented in this paper. Observations from the running system show the performance bottlenecks, of the PIE1 system, to be located in the relational model and in the software monitoring. Accordingly, instead of using an available relational data base system (INGRES as used in PIE1), we are exploring the use of the relational model in ways specifically suited to PCT. One of the approaches under consideration is a memory resident relational model with optimization for often executed PCT relational operations. Related to the alleviation of the software monitoring performance bottleneck is the definition and the design of a "programmable general purpose multiprocessor architecture." This multiprocessor, the Supercomputer Workbench dedicates a large amount of hardware to the task of non-intrusive monitoring providing direct support for PCT and programming for observability.

The Implementation Assistant is currently in the feasibility study state. The performance predictor is operational. The model covers applications which are not heavily data dependent. Sample implementation templates are under construction and and a set of templates with application to real-time processing have been evaluated. The concept looks promising, however more flexibility in terms of conveying performance requirements to the system is required. Work employing production systems techniques to solve this problem has been started. Most of this work is the object of the second research phase, i.e. PIE 4.

In terms of the open and unsolved research issues in PIE, one of the main open problem is the use of formal specifications in the process of parallel program development. A desideratum will be to integrate formal specification for both functionality and performance requirements into the programming environment. The current state of art in this domain does not provide enough of a practical approach to enable us to consider the integration of such techniques in PIE 1. A second open question is how to provide support in PIE for different styles of programming languages such as LISP and Prolog. This problem has not been explored in detail in PIE 1, and is being postponed to the PIE 4 design cycle. In PIE 1, the issue of what is the most suitable graphic interface has not been considered as the the main goal. It our belief that a lot more experimentation and user feedback is necessary in order to make a determination in this domain. Initial PIE 1 results, however, show that associating graphic presentations with the relational representations is attractive.

In summary, this paper presents a snapshot of a relatively large research project on the move. An overall programming environment structure reflecting the goals and the characteristics of parallel programs has been designed and implemented. A set of concepts have been introduced and evaluated, including programming for observability, semantic monitoring, relational parallel program representation, constraint driven abstract data types (frames), automatic frame unfolding, programming with templates, and parallel implementation assistant. Our initial results and feasibility studies are very encouraging.

8. Acknowledgment

A research project having the goals and size of the PIE can succeed only by the cross-pollination of a group of talented researchers. The authors wish to acknowledge the contribution of the following researchers. Dalibor Vrsalovic has been responsible for the performance prediction model part of the Implementation Assistant, and the design of the User Interface; Robert Whiteside has been instrumental in providing the first version of the Relational Representation System; Francesco Gregoretti has provided the first version of the Status and Performance Monitor; Beth Bottos has been responsible for the MP.Frames design; Eddie Caplan provided the implementation of the graphical User Interface. As part of her doctorial dissertation research, Beth Bottos will continue exploring the power of the constraint specifications as well as the automatic insertion of coordination code to realize these constraints. Apart from these specific contributions, we wish to acknowledge their general contributions during numerous general designs PIE Project meetings. Dan Siewiorek deserves special acknowledgment for his continuous encouragement and wise suggestions.

References

[1] Andler, S.
Predicate Path Expressions.
In *Principles of Programming Languages*. 1979.

[2] Archer, J., and Conway, R.
COPE: A Cooperative Programming Environment.
Technical Report TR81-459, Cornell University, June, 1981.

[3] Brinch-Hansen, P.
Distributed Processes: A Concurrent Programming Concept.
Commun. ACM 21(11):934-941, November, 1978.

[4] Campbell, R. H.
Path Expressions: A Technique for Specifying Process Synchronization.
Technical Report, University of Newcastle, August, 1976.

[5] Deutsch, L. P. and Taft, E. A.
Requirements for an Experimental Programming Environment
CSL-80-10 edition, Xerox, 1980.

[6] Ellis, J., Hooper, J. and Johnson, T.
An Architecture Description Language, ADL for Describing and Prototyping Distributed Systems.
In *System Science Conference*. January, 1984.

[7] Habermann, A. N.
Path Expressions.
Technical Report, Carnegie-Mellon University, June, 1975.

[8] Habermann, A. N.
The Gandalf Research Project.
Computer Science Research Review, Carnegie-Mellon University, 1979.

[9] Hoare, C. A. R.
Monitors: An Operating System Structuring Concept.
Communications of the ACM 17(10):549-557, October, 1974.

[10] Jones, Michael B., Richard F. Rashid, ary R. Thompson.
Matchmaker: An Interface Specification Language for Distributed Processing.
In *Principles of Programming Languages*, pages 225-235. ACM, January, 1985.

[11] Kuck, D. J., R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe.
Dependence Graphs and Compiler Optimizations.
In *Proc. of the 8th ACM Symp. on Principles of Programming Languages*, pages 207-218. ACM, ACM, January, 1981.

[12] Kuck, D. J., R. H. Kuhn, B. Leasure, and M. Wolfe.
The Structure of an Advanced Retargetable Vectorizer.
Tutorial on Supercomputers: Design and Applications :168-178, 1984.

[13] Mitchell, J. G., Maybury, W. and Sweet, R.
Mesa Language Manual
CSL-79-3 edition, Xerox, 1979.

[14] Owicki, S. S. and Gries, D.
An Axiomatic Proof Technique for Parallel Programs.
Acta Inform. :319-340, 1976.

[15] Owicki, S. S. and Gries, D.
Verifying Properties of Parallel Programs: An Axiomatic Approach.
Commun. ACM 19(5):279-285, May, 1976.

[16] Reiss, S. P.
PECAN: Program Development Systems That Support Multiple Views.
In *Proceedings of the Seventh International Conference on Software Engineering*. March, 1984.

[17] Segall, Z., Singh, A., Snodgrass, R. T., Jones, A. K. and Siewiorek, D.
An Integrated Instrumentation Environment for Multiprocessors.
In *IEEE: Transactions on Computers*. January, 1983.

[18] Snodgrass, R.
Monitoring Distributed Systems: A Relational Approach.
Technical Report, Carnegie-Mellon University, December, 1982.

[19] Snyder, L.
Introduction to the Poker Programming Environment.
In *Proceedings of the 1983 International Conference on Parallel Processing*, pages 289-92. Purdue University, August, 1983.

[20] Teitelbaum, T. and Reps, T.
The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.
Technical Report TR 80-421, Cornell University, May, 1980.

[21] Vrsalovic, D., Siewiorek, D., Segall, Z. and Gehringer, E.
Performance Prediction and Calibration for a Class of Multiprocessor Systems.
Technical Report, Carnegie-Mellon University, June, 1984.

[22] Vrsalovic, D., Gehringer, E. F., Segall, Z. Z. and Siewiorek, D. P.
The Influence of Parallel Decomposition Strategies on the Performance of Multiprocessor Systems.
In *The 12th Annual Symposium on Computer Architecture*. June, 1985.