Programming Environments for Parallel Programming

Andrew W Kwan

Department of Information and Computer Science
University of California Irvine
Irvine, California 92717

Technical Report #89-06
January 1989

Programming Environments for Parallel Programming

Andrew W Kwan

Department of Information and Computer Science
University of California Irvine
Irvine, California 92717

abstract

Programming environments are used to bridge the gap between actual computers and development of their application programs. Most parallel programming environments currently in use focus on a specific parallel programming tool. This paper examines programming environments, languages, tools, and techniques used for programming of parallel computers.

In this paper, several topics are examined. First, a brief survey of parallel computer architectures and typical application programs is performed. Then, a survey of available environments, languages, and tools is conducted to determine how parallel programming is currently performed. Finally, by considering architectures, applications, and environments, an attempt is made to find desirable characteristics for a parallel programming environment and a useful set of parallel programming tools.

Programming Environments for Parallel Programming

Andrew W Kwan

Department of Information and Computer Science
University of California Irvine
Irvine, CA 92717

1. Introduction

Programming parallel computers is a difficult skill. A programmer must not only develop an algorithm to solve a given problem, but must also implement that algorithm as a program to run on a machine composed of multiple processing elements. Success is measured by how quickly this program runs, compared to a program that solves the same problem on a sequential, single processor computer.

The purpose of a programming environment is to support and aid the development of programs. The characteristics of an environment for parallel programming are different than that for sequential programming, and are not as well known. Most literature on parallel programming environments focuses on a specific tool. There is little emphasis on tool sets and an overall environment architecture to support parallel programming.

A survey of parallel programming environments, tools, and techniques was conducted. The purpose was to determine how parallel programming is performed and to find desirable characteristics of an environment architecture and a useful set of tools (over and above that for sequential computer programming). The simple "waterfall model" of the software lifecycle was used as a basis for program design and development, and steps for parallel program development were derived.

2. Parallel Computer Architectures and Applications

In this section, a brief introduction to parallel computer architectures and application programs is presented. Parallel computing is unique in that it represents a whole new field within computer science, yet is being driven by a specific set of applications. Programs written for sequential (Von Neumann-type)

computer architectures have wide application to many fields (e.g., the sciences, business, etc.). But programs written for parallel computers are generally either simulations of physical systems (e.g., weather forecasting, fluid flow, etc.) or numerical analysis problems (e.g., image and signal processing, matrix multiplication, etc.). Such applications derive benefit from parallel computers by running more quickly, thereby allowing finer simulation resolution or larger problem sizes.

2.1 Parallel Computer Architectures

Computer architectures were classified in [Fly66] (also see [SMS88]) by the number of instruction and data streams executing on a computer, where an instruction stream is a sequence of instructions, and a data stream is a sequence of data values utilized by an instruction stream. Four classes of machines were defined:

- SISD single instruction stream, single data stream
- SIMD single instruction stream, multiple data stream
- MISD multiple instruction stream, single data stream
- MIMD multiple instruction stream, multiple data stream

Sequential, single processor computers are considered to be SISD computers. When an SISD computer executes, it executes a single application program on a single set of data values.

Parallel, bit-serial computers fall into the SIMD computer category. A typical SIMD computer consists of a control unit and a number of interconnected processing elements (PEs) (each with a local memory). The same program is executed on each PE over different data items. Communications between PEs occurs over the interconnection network. The control unit controls sequencing of the program on each PE, so that each PE is executing the same instruction (of the same program) at any given time.

Typical SIMD-type computers use hundreds of simple, small PEs, interconnected as a grid, cube, or other regular structure. These computers work very well for applications that can be partitioned into smaller tasks that fit well into the computer's interconnection structure. Typical applications include

problems using matrices, and image processing.

There are no commonly used MISD-type computers. Such a computer would execute different programs (simultaneously) over the same set of data. There are few problems of this type, and could be executed on more flexible MIMD computers.

MIMD-type parallel processing computers typically consist of several PEs and several memory modules, connected by an interconnection network. Each PE can execute its own program on the data assigned to it. MIMD computers with relatively few PEs often have the PEs interconnected by a bus. MIMD computers with larger numbers of PEs are usually interconnected into a regular structure. Just as with SIMD computers, performance of the MIMD computer on a problem depends on how well the problem can be partitioned to fit the interconnection structure.

There are two basic types of MIMD computers: shared memory and message passing computers. In a shared memory MIMD computer, many, if not all, memory modules are organized into a global memory which is accessible by all PEs (although access time to any memory location may not be the same for all memory locations). PEs communicate by leaving data stored in global memory. Message passing MIMD computers do not have a global memory. Instead, each PE has a local memory which is not accessible to any other PE. PEs communicate by sending messages across the interconnection network to other PEs.

2.2 Parallel Computer Applications

As mentioned previously, applications will have the best performance when partitioned to match the structure of the parallel computer architecture. In [Sto87], several important problem areas are noted which have need for high performance computation. These are:

- *Highly structured numeric computations* — weather modeling, fluid flows, finite element analysis;
- *Unstructured numeric computations* — Monte Carlo simulations, sparse matrix problems;
- *Real-time multifaceted problems* — speech recognition, image processing, and

computer vision;

- *Large-memory and input/output-intensive problems* — database systems, transaction systems;
- *Graphics and design systems* — computer-aided design; and
- *Artificial intelligence* — knowledge-base-oriented systems, inferencing systems.

A search through the current literature on parallel processing algorithms finds that most research in applications is concentrated in the first three areas listed above, and in particular the first one.

Most simulations of physical systems can be classified as highly structured numeric computations. This is due to the fact that since they are physical systems, they are usually models of two or three dimensional systems in space, varying over time. Each spatial dimension and time usually can be discretized into regular intervals, and so these models become implemented as highly structured numeric computations.

[Hos86] and [Sto87] note that physical computation models fall into one of two categories: continuum models and particle models. The continuum model is used to capture continuously varying quantities (e.g., force, temperature) over space and time, and is usually implemented by dividing up space and time into discrete regions and assigning the desired quantity a value by taking an average over the region. Further, continuum models typically are models of systems which follow from partial differential equations, so that changes in a quantity's value for a region are dependent on the values in neighboring regions. An application which can be classified as a continuum model can be matched to an architecture by assigning a region to a PE such that it's neighbors (over the interconnection network) are assigned regions whose values the (first) PE needs.

The particle model is used to capture quantities associated with discrete particles of the system. Typically, the quantity associated with each particle will be dependent upon the influence of several (and possibly all other) particles in the system. The particle model is difficult to match to a non-fully connected architecture (in a fully-connected architecture, each PE can communicate directly, and at minimum cost, with every other PE), since if a particle is assigned to a PE, the PE will need to communicate with some

(or all) other PEs to receive information on their assigned particles. Fully connected architectures typically are very expensive (due to the large number of interconnections), and so are not in great use.

It should be noted that even though the continuum model and the particle model are different, they have one very important factor in common. That is that the interactions of one region with others in the continuum model, and of one particle with others in the particle model, usually can be described in the same manner for all regions or particles. In other words, only one program need be written for a partition of regions or particles over PEs. Both models can be described (programmed) relatively easily for SIMD computers. In considering parallel programming environments, and realizing that most applications will be implementations of physical computation models, a general parallel programming environment should provide a large amount of support for computations typical for SIMD computers.

Computations typical for SIMD computers have been termed Single Program, Multiple Data (SPMD) computations, in order to distinguish them from the term "SIMD," which relates to an architecture. Although SIMD computers usually have their PEs execute programs in a lock-step manner, SPMD computations are not so rigidly executed. SPMD computations can be executed on MIMD computers. Since MIMD computers have PEs which can operate independently of one another, SPMD programs may contain conditional paths which allow somewhat different processing for different data on PEs (e.g., data for boundary regions, instead of interior regions). However, at some point PEs have to become synchronized with each other.

In a similar vein, computations typical for MIMD computers will be referred to as Multiple Program, Multiple Data (MPMD) computations. An MPMD computation would have a different program assigned to each processing element, operating on different data. An MPMD computation might consist of a collection of different tasks. A program assigned to a PE would consist of a subset of those tasks being scheduled and mapped together. With such a model for MPMD computation, by casting an SPMD computation as a collection of tasks (even though many tasks would be the same, just operating on different data items), an SPMD computation can be transformed into an MPMD-like computation.

However, the converse cannot be performed.

3. Parallel Programming Tools and Environments

This section provides a brief survey of tools and environments currently available to support parallel programming. These tools have been classified by function.

3.1 Task and Task Interface Design

Tasks are defined to be a part of the program that performs a certain function. Task interfaces are defined to be the structures used to transfer information (i.e., communicate) between tasks.

Polylith [PRG87] is an environment which supports prototyping of algorithms in an architecture independent manner by separating the specification of task interfaces from the underlying architecture. To implement a program, the programmer first specifies the tasks and task interfaces. Then the programmer specifies the implementation of the computer's architecture by specifying the PEs, memory modules, and the interconnection structure. This allows for portability of programs across different architectures, and allows experimentation with different architectures for determining optimal program performance, by modifying the architecture specification.

POKER [SnS86] is a parallel programming environment for message passing, MIMD computers. It requires a set of interprocessor communication primitives and a sequential language for task description. To program an application using POKER, the programmer specifies a task graph, where each node corresponds to a task, and edges correspond to communications. The programmer then binds a task to each node and an (inter-task) message to each edge. Additionally, graphs may have "dangling" edges, which provide for data input and output to the graph. The complete graph description is called a phase. A program is composed of a sequence of one or more phases (phases communicate with each other via the dangling edges) and an execution scheduler that describes how tasks are assigned to PEs. A significant

feature of the POKER environment is that it provides a graphical view of the program with interactive graphical program editing. The environment also provides a trace facility which allows the programmer to start, stop, and single step the program while viewing the values of trace variables.

MUPPET [MKL87] is an environment targeted for the West German SUPRENUM supercomputer project. Programs are written in the Concurrent MODULA-2 language for an abstract message passing computer by specifying processes and inter-process communications. The abstract computer is then mapped onto the SUPRENUM architecture to create the program. MUPPET also provides a graphical interface, GONZO, which provides for graphical program specification and animation of program execution.

HYPERTOOL [WuG88] is a programming environment for hypercube message passing computers. Programs are written in a subset of C by specifying tasks and their communications. HYPERTOOL then constructs a data flow graph of the program, schedules tasks, and maps the tasks onto PEs. The program is then executed on a hypercube simulator, which provides some execution trace information. An interesting feature of HYPERTOOL is that the program is initially debugged on a sequential computer.

Examination of these environments shows some common features. Each of these is intended for use on an MIMD computer. Programming is performed by specifying tasks and their interfaces, with no mention of the intended target architecture. The tasks are then scheduled and mapped from an underlying abstract computer onto a real computer. Graphical interfaces are used to edit the task graph, which is really a data flow graph with tasks as nodes and inter-task communications as edges, and to provide visualization of program structure.

3.2 Languages and Compilers

Most programming languages for parallel processing currently in use are modified versions of sequential programming languages. The most widely used is FORTRAN, with C as a distant second favorite. There have been some alternate languages proposed. A brief summary of the language choices available are summarized below.

3.2.1 FORTRAN

[KaB88] provides a survey of FORTRAN dialects available for use on commercial parallel computers. Most of these dialects merely provide extensions for parallel synchronization and control, such as cobegin and coend statements, fork and join statements, barrier synchronization statements, lock and unlock statements for shared memory locations, and facilities for sharing data in shared memory systems. It is left to the programmer to control task and data partitioning, scheduling, and synchronization. This is not a satisfactory state of affairs, as partitioning, scheduling, and synchronization are what make parallel programming difficult. The programmer is given little support.

FORTRAN compilers provided by Alliant, Cray, and IBM go a step beyond by providing for some automatic parallelization of code. The Alliant FORTRAN compiler can vectorize some types of loops, that is, remove data dependencies between loop iterations so that each iteration's results can be calculated independently, organize the data required by the loop iterations into vectors, and schedule the vectors for processing. The Cray and IBM FORTRAN compilers can support parallelization of some types of loops (called microtasking) and parallelization of subroutine calls (called multitasking). These two features allow for some detection of parallelism, and semi-automatic scheduling. However, much work is still left to the programmer.

3.2.2 Other algorithmic languages

HYPERTOOL [WuG88], as previously mentioned, allows programming using a subset of C. The programmer partitions the program into tasks, and identifies the input and output parameters of each task. HYPERTOOL then compiles the program, schedules the tasks, and inserts the communcation and synchronization primitives automatically. This is a definite improvement, as proper handling of communication and synchronization is the cause of many parallel programming errors, and generally such errors cannot be found until the program is executed.

ADA, Concurrent PASCAL, and Concurrent MODULA–2 are other algorithmic programming languages which provide for concurrent program execution. These languages are interesting because they supposedly allow for architecture independent parallel processing. However, programs must still be scheduled and mapped efficiently onto an actual architecture. This is a difficult problem, and currently must be performed manually, which removes the primary advantage of using these languages.

3.2.3 Single assignment languages

In a single assignment language, memory locations may have a value assigned only once, although they may read multiple times. This provides for program sequencing by data dependence, only allowing program statements and procedures to be executed when the data required is available. As a result, race conditions and data coherence problems are avoided. Programming for parallelism occurs as a result of data usage and availability, and does not have to be explicitly programmed. In some ways this method is similar to the task-oriented programming methods previously described, except at a lower level.

SISAL [BoG87] is a single assignment language originally intended for description of data flow computation. It is used in within an environment which includes a compiler, several computer simulators, and a graphical program execution monitor. The program execution monitor can display a procedure call graph (a display of the program's procedures and data dependences as they are executed) and can display PE utilization statistics over time. These help the programmer to visualize their program's execution over time.

3.2.4 Functional languages

Id Nouveau [ArE87] is a new version of the Irvine Dataflow language, and is based upon a functional programming approach, that is, program code is written as a set of functions (in a manner similar to LISP programming), where each function has a task and a clearly defined set of input and output parameters. Id Nouveau also utilizes single assignment. Program sequencing in Id Nouveau occurs as a

result of function calls and data dependences. This can be considered another method where the programmer separates the tasks to be performed, and the task interfaces, from the underlying architecture.

The Gibbs Project [Nic87] [Wil86] is an effort to program parallel computers in a method similar to how physical scientists describe systems. For example, a physicist might describe the motion of a set of objects by writing down the general equations of motion for each object, recording the attributes of each object, recording any initial and boundary conditions, and then describing the fineness of resolution desired (e.g., a time step between each iteration). Computer scientists might write a program for such a system by first describing a data structure to represent the system, then breaking up the overall problem into tasks and their interfaces, and then finally describing exactly the equations of motion, initial and boundary conditions, and resolution. Clearly, there is a great difference in how each approaches a problem. Research is being performed to find a language suitable for this kind of problem expression.

3.3 Static data flow analysis tools

Static data flow analysis tools perform data flow analysis of a program based on the program's source code only (dynamic analysis tools make use of information from program execution). Such tools are typically used to uncover data dependences and find parallelizable code.

The *CAMP* tool [PGW87] is used to analyze loops and remove data dependences between iterations, so that individual loop iterations may be executed in parallel. A performance estimator is also included so that the programmer can analyze the effect of various data partitioning schemes on loop performance.

Similar analysis tools were embedded in some of the FORTRAN compilers previously mentioned.

These tools typically do not work when the loop contains procedure calls, since such calls may contain implicit data dependences to other loop iterations. In [Tri87], analysis of interprocedural data flow information is also performed. This provides the additional information necessary to analyze the effects of a procedure call in a loop, possibly allowing the data dependences between loop iterations to be removed and the loop iterations to be executed in parallel.

Another use for static data flow analysis is described in [TaO80]. Here, analysis may uncover programming anomalies, that is, errors in variable definition or use due to race conditions. Since the analysis is not dynamic, actual instances of race conditions cannot be found, but potential instances can be located.

3.4 Scheduling and mapping

There are two basic methods available for scheduling and mapping tasks onto processing elements: static and dynamic. Scheduling of a task refers to the assignment of an execution time for a task. Mapping of a task refers to the assignment of a task to a certain processing element for execution. Static methods must know or have an estimate of the execution time of each task, and must know the information that must be present to execute a task (i.e., the data dependences). The tasks can then be scheduled into time slots. Once all tasks are scheduled, collections of tasks are assigned to individual PEs. Static scheduling is done at compile-time.

HYPERTOOL [WuG88] utilizes a static scheduling and mapping method. A data flow graph is constructed, with nodes corresponding to tasks and edges to messages. Nodes are assigned a weight equal to the corresponding task's execution time, and edges are assigned a weight equal to the corresponding message's transmission time. The graph is then traversed, the critical path identified, and nodes assigned the earliest possible execution times. The nodes are then scheduled onto a virtual PE. After all nodes are scheduled, each virtual PE is assigned to an actual PE to minimize communication time.

In dynamic scheduling, task execution times are not known, and tasks are scheduled and mapped at run time onto a PE when its data dependences are satisfied. In [Pol88], a data flow graph is constructed, and communication primitives are inserted before and after each task's program code. Tasks are initially placed into a queue, depending on their depth in the graph (root nodes are located at the front of the queue). A host processor administers the queue, examining tasks in the queue to see if their data dependences have been satisfied, and if so, the tasks are send to a PE for execution. PEs that are not executing tasks are available

for task assignment by the host.

3.5 Execution monitoring and debugging

There are few techniques available for debugging parallel programs. Most are based upon execution monitoring and profiling the programs as they run. In *Belvedere* [HoC87], a multiprocessor simulator is used to execute a parallel program, and an execution trace is produced. The trace contains a list of time-stamped events, such as message transmission and reception, and task initiation and completion. This list is treated as a database, and queries are made by the programmer for certain patterns of events. The results of these queries can be graphically displayed and animated. For example, one might look for certain communication patterns corresponding to information transfer between neighboring PEs. Belvedere would find the corresponding events and display the PEs, the interconnection network, and source and destination of the messages. This can provide the programmer with a visualization of how the program executes, which can be compared to how it should be executing.

In [AlP87], a dynamic data flow analysis tool is presented which utilizes program source code and execution trace information to uncover actual and potential race conditions. Static data flow analysis is utilized to find potential race conditions. Analysis of the execution trace can determine if any of the potential race conditions were actually realized (for a particular set of input data). The analysis can also check for potential race conditions which are possibly hidden, due to the effects of the actual race condition, although in general no conclusions can be drawn until the actual race condition is eliminated.

Other previously mentioned execution monitoring tools are in POKER, MUPPET, and the program execution monitor for SISAL.

3.6 Performance analysis

Parallel program performance analyzers, like debuggers, have few techniques available, and are derive their analysis from execution information. In [SBD87], static and dynamic data flow analysis is used to

produce a weighted data flow graph (similar to that used for scheduling in HYPERTOOL) which represents the parallel program. This graph is then scheduled and mapped over a range of numbers of PEs to show program speedup versus number of PEs used. This can be used by the programmer to improve program performance and to find a number of PEs that provides good program performance.

COSMIC [CaF87] uses timed Petri nets to analyze program performance. A weighted data flow graph is produced from the program, and is simulated as a Petri net. Several Petri nets can be produced, which contain some or all of the constraints associated with the program, such as resource access, memory access, etc. The Petri nets can demonstrate how the execution time and critical path can vary with different overhead costs.

3.7 Summary of parallel processing tools and environments

There are many tools and environments available to support parallel programming. However, none of the environments seems to be complete, in that the environment can support all aspects of program development. This is examined more in Sections 4 and 5. There was much use of graphics to visualize program execution, which programmers can use to compare with their own idea of how the program should be executing. Visualization is a powerful concept, because it is very difficult to put together a picture of program execution from textual information. Time, communication and synchronization, tasks, and processing elements combine into a picture with too many dimensions for a programmer to uncover in all but the simplest cases.

Many of the programming techniques support a task oriented, architecture independent approach to program development. This approach is useful because there are many different parallel computer architectures available, none of which has emerged as a standard. However,this approach is troubling because it runs counter to the intuition developed in Section 2 — that programs must be well matched to the architecture for best performance. Presently, good automatic techniques for transforming programs written for abstract parallel computers to those for real computers have not been found.

4. Programming Environments and the Software Lifecycle

In this section, we examine how development of sequential and parallel programs are different, and the support that should be provided by in a parallel programming environment.

4.1 Sequential Programming Environments

Figure 4–1 shows the common, simple view of the software lifecycle. The parts of the lifecycle that correspond to the steps involved with development of program code are illustrated. These steps can be described in a more detailed manner, as shown in Figure 4–2. It is in the detailed design phase that the program is broken up into modules, where each module performs some task. The actual program code will have an implemention of each module, and so will be a reflection of the detailed design. The detailed design also reflects the (as yet unwritten) program code, since the design will consider details of the program's implementation. Thus, we can say that program code development starts in earnest in the detailed design phase. The purpose of a programming environment is to support program code development, and so a logical place to start to provide support would be in (the latter stages of) the detailed design phase.

In the program coding phase of the software lifecycle, the detailed design of the program becomes implemented in a chosen programming language. However, it is rare for a program to become implemented without debugging and testing. Program debugging and testing is considered to be a part (the initial stages) of the maintenance phase of the software lifecycle. Debugging and testing occurs for each module of the design as it is coded, and as it is integrated into the overall program. Thus, there is much feedback between the program coding phase and the initial stages of the maintenance phase. A logical place to end support of program development would be in the early stages of the maintenance phase.

A programming environment for sequential programming should provide support in the detailed design phase for module design and interfacing, the program coding phase, and in the maintenance phase for debugging and (some) testing.

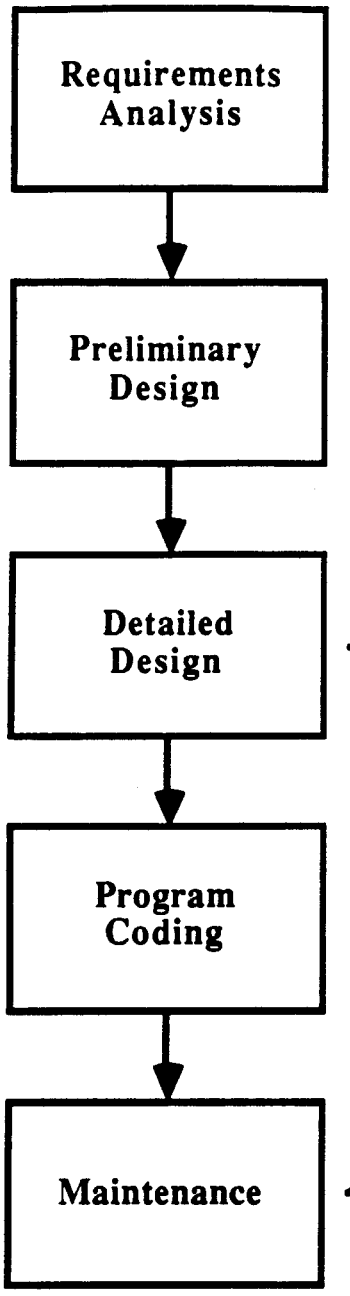

Figure 4–1
The Basic Software Lifecycle

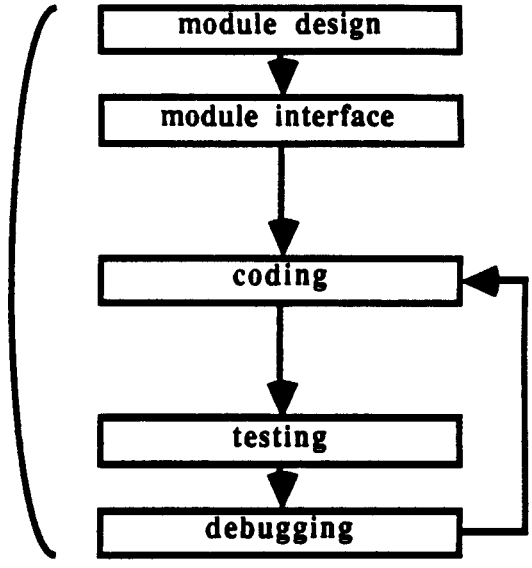

Figure 4–2
Sequential Programming Environment

4.2 Parallel Programming Environments

The software lifecycle can also be applied, in general, to the development of parallel programs. However, parallel programming is more complex. Figure 4–3 illustrates the steps to be performed in parallel programming. Tasks to be performed, and the interfaces between tasks, are developed in the detailed design phase. This is similar to the sequential detailed design phase.

In the program coding phase, tasks and task interfaces are implemented as program code. However, in parallel programs, tasks must be scheduled in time and then mapped for execution on specific processor elements. This is one difference that distinguishes parallel program development from sequential program development. A parallel program is not complete until it is scheduled and mapped.

Debugging and testing of parallel programs (in the maintenance phase) is also more complex than for sequential programs. For sequential programs, debugging and testing (for program development) is performed to remove coding errors, uncover errors in functionality, and to integrate the pieces of the design into a whole. For parallel programs, debugging and testing also includes the uncovering of errors due to improper sequencing of tasks, execution monitoring of the program, and and performance analysis for the improvement of program performance. Improper task sequencing is more likely to occur in parallel programs than in sequential programs, because control is more difficult. Tasks may execute simultaneously in a parallel program, and in an arbitrary order, whereas in a sequential program tasks execute one at a time, and in a specific, fixed order. Improper task sequencing can lead to race conditions, where a data item might have values stored by multiple tasks, and the execution sequence of future tasks will depend upon the order that the values are stored. Some race conditions can be very difficult to uncover when the time between possible value storages is small, because the amount of time for task execution may vary for reasons not under the programmer's control (such as the operating system or resource access). Debugging, testing, and execution monitoring tools are necessary for detecting possible race conditions, and for monitoring task sequencing, task interfacing, and processing element utilization.

Improvement of program performance is another difference that distinguishes parallel from

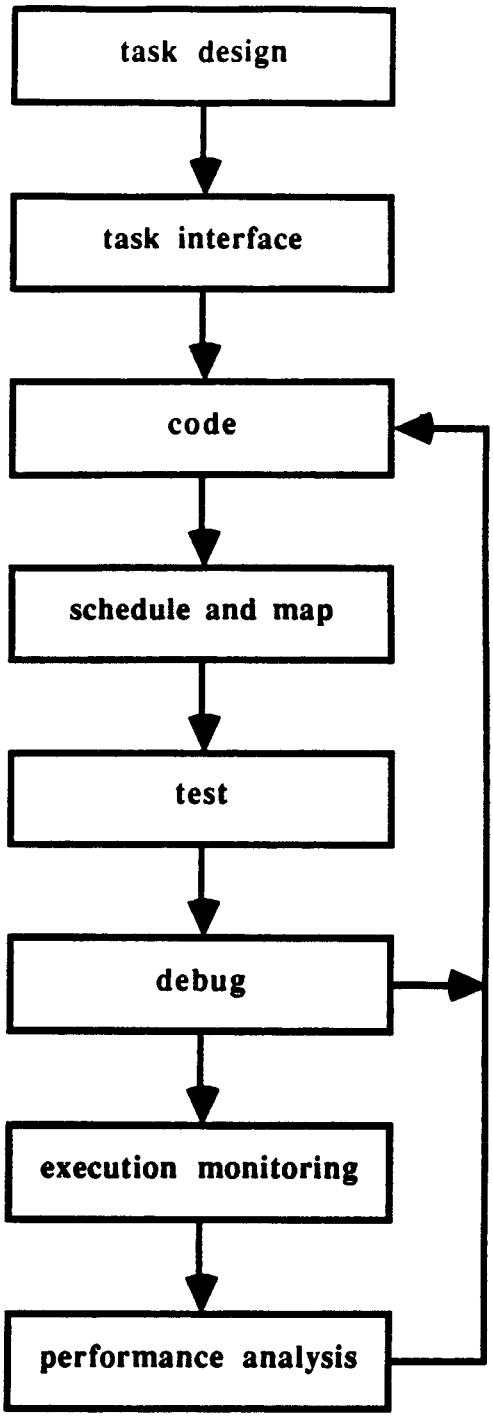

Figure 4–3
Parallel Programming Environment

sequential program development. On the surface, if a parallel computer has N processing elements, then one would think that a parallel program should run N times as fast as its sequential equivalent. In practice, this is extremely rare, as it is difficult to keep all N processing elements busy all the time, and there are various overhead costs (e.g., communication) accrued by dividing up the problem among different processing elements that slow the program. To maximize computer usage and efficiency, the program should be improved until it can run as fast as possible.

4.3 Parallel Programming Tool Sets

Figure 4–3 and Section 4.2 presented outlined the support that a parallel programming environment should provide. Section 3 presented various tools useful for parallel programming. This section will discuss what kinds of tool sets should be provided.

It was previously mentioned that SPMD computations are well suited for SIMD computers, and that MPMD computations are well suited for MIMD computers, and that SPMD computations could be transformed into MPMD-like computations (although they probably would not execute as efficiently), and so could be executed on SIMD or MIMD computers. The implication is that between SIMD and MIMD computers, MIMD computers are more flexible since they can execute either type of computation. However, SPMD computations seem to represent the majority of applications for parallel processing. This suggests that a parallel programming environment on an MIMD computer should contain separate tools for both SPMD and MPMD program development.

However, such tools would differ somewhat in function, but not form. Both SPMD and MPMD program development would benefit from interactive, graphical tools. The primary difference lies in how tasks and task interfaces are described. SPMD programs will be made from a collection of tasks, which will be assembled into a program that will execute on all PEs. Communications between PEs should be relatively simple, in that they should occur at about the same time (since programs executing on each PE will be identical). MPMD programs utilize a task oriented, architecture independent approach. An MPMD

program will contain of subset of tasks from a larger collection. Input and output data for tasks serve as the method of communication. Graphical tools are best used to show communications and task relationships. A graphical tool for SPMD programming should exhibit how PEs communicate with each other. A graphical tool for MPMD programming should exhibit how tasks are grouped into programs, and how data is passed between tasks (and PEs). Graphical methods have been demonstrated to support task and task interface design for MPMD programming (e.g., POKER and MUPPET). It should be straightforward to utilize these methods to support SPMD programming, too. A potential problem with graphics based tools is that it is not clear how well they would handle large programs or architectures, since a screen can only hold so much data before it becomes too cluttered. If this is in fact a problem, a tool to provide some sort of automatic scaling of solutions from a small size to a large size would be helpful in overcoming that problem.

Task and task interface design for SPMD programming should be simpler than for MPMD programming, due to the regular structure of tasks, communications, and architecture. Communications should be simpler, and race conditions much more easily avoided. Thus, there should be little need for data flow analysis tools for SPMD programming. There will be great need for static data flow analysis tools to support MPMD programming. Static data flow analysis can be utilized on the tasks to find more parallelism (as described in [Tri87]), and on task interfaces to find potential race conditions (as described in [TaO80]).

Scheduling of SPMD programs is almost trivial, since each PE executes the same program. Mapping would consist of assigning data to PEs so that communications distance is minimized. For MPMD programming, a good scheduler and mapper are necessary to produce efficient parallel programs. The task (data flow) graph describes how the program would execute if there were as many PEs as necessary for maximal parallelism. The task graph must be scheduled and mapped onto a real architecture, with a limited number of PEs. Efficient scheduling and mapping is a difficult task (optimal scheduling is an NP-complete problem).

Static scheduling is preferable to dynamic scheduling due to the overhead and control needed for dynamic scheduling, however static scheduling requires a method of task performance estimation. A versatile scheduling technique might utilize a dynamic scheduler or a performance estimator (for static scheduling) for the initial program scheduling, but after execution would utilize the profiling information available (from the program execution) to improve the scheduling for the next program execution. Subsequent executions could continue to refine the program scheduling.

Graphical execution monitoring and debugging tools are needed to assist the programmer with visualization of program execution and communication patterns. For SPMD programs, this would provide a useful check against how the programmer thinks these occur. For MPMD programming, a programmer would probably be able to visualize program execution by looking at a task (data flow) graph, but after scheduling it probably would not be clear how the program executed. A tool such as Belvedere, which provided a view of event patterns among PEs, would be useful in this regard. Belvedere provided a view suitable for SPMD programming, but it should be possible to provide a useful view of MPMD programming, too. A graphical execution monitoring and debugging tool should be tightly integrated with the graphical design tools.

For MPMD programming, dynamic data flow analysis tools for uncovering race conditions (as in [AlP87]) would be extremely useful. As with scheduling, it should be possible with dynamic data flow analysis to take information provided from program execution and feed back information into program design.

Simulation (of program execution) is also useful technique for execution monitoring and debugging. A simulator with start, stop, single step, and display capabilities (similar to a source level debugger) can provide the programmer with direct control over program execution and a window into the program's state. However, it is difficult to accurately simulate multiple PEs over time. For example, the SIMON multiprocessor simulator [Fuj83] simulates by executing the assigned tasks of one PE until it becomes blocked by a communication primitive. It then places the PE (and it's state) into a queue, removes another

PE from the queue, checks to see if it is blocked, and if not blocked then simulates it. The state of each PE at a given (simulator) time is not known. Only the state of one PE can be known.

Performance analysis tools are needed to analyze the effects of different partitioning schemes. In [PeG86], comparisons were made of algorithm speedup and the aspect ratio (a measure of the block size) for data partitions. This analysis allowed the programmer to choose a partitioning scheme that minimized program execution time. Choosing a method of partitioning is a very intuitive task, and requires special insight into the problem. It would be very difficult to develop a tool that could perform the necessary analysis automatically, but would be extremely useful since partitioning skill is the most difficult (but necessary) skill that a good parallel programmer must have.

In general, there were few debugging and performance analysis tools found in the literature. This is disappointing, in that good debugging and performance analysis capabilities are necessary for improvement of parallel programs. If fast parallel programs remain extremely difficult to develop, there will probably be little effort in making parallel computing viable (the most effort would be placed on making sequential and vector computers faster).

5. Future Research

Graphical techniques for parallel programming are just beginning to be utilized, and there is much room for growth. Available graphical techniques tend to support MPMD program design (i.e., task oriented, architecture independent). Work needs to be performed to extend this to SPMD programming, and to provide well integrated graphical execution monitoring and debugging.

Good static scheduling algorithms have already been found that are known to provide schedules that are within a constant factor of optimal. The challenge in utilizing static scheduling is to find ways of obtaining task execution times short of actually executing the program. Dynamic scheduling and mapping techniques are not well known, and there is much room for improvement. Most research being performed

utilize heuristics to successively refine the scheduling and mapping.

As previously noted, there are few tools available for parallel program debugging and performance analysis. The most useful techniques so far have been static and dynamic data flow analysis for detection of parallelism and race conditions. Other techniques are needed to help improve performance of parallel programs.

Acknowledgement

This work has been supported, in part, by the National Science Foundation (grant CCR–8700738).

References

[AlP87] T.R. Allen and D.A. Padua. *Debugging Fortran on a Shared Memory Machine*. Proceedings of the 1987 International Conference on Parallel Processing (S.K. Sahni, editor). The Pennsylvaina State University Press, University Park, PA, 1987.

[ArE87] Arvind and K. Ekanadham. *Future Scientific Programming on Parallel Machines*. Proceedings of the 1st International Conference on Supercomputing (E.N. Houstis, T.S. Papatheodorou, and C.D. Polychronopoulas, editors). Lecture Notes in Computer Science 297 (G. Goos and J. Hartmanis, editors). Springer-Verlag, Berlin, 1988.

[BoG87] A.P.W. Böhm and J.R. Gurd. *Tools for Performance Evaluation of Parallel Machines*. Proceedings of the 1st International Conference on Supercomputing (E.N. Houstis, T.S. Papatheodorou, and C.D. Polychronopoulas, editors). Lecture Notes in Computer Science 297 (G. Goos and J. Hartmanis, editors). Springer-Verlag, Berlin, 1988.

[BrS87] T. Brandes and M. Sommer. *A Knowledge-Based Parallelization Tool in a Programming Environment*. Proceedings of the 1987 International Conference on Parallel Processing (S.K. Sahni, editor). The Pennsylvaina State University Press, University Park, PA, 1987.

[CaF87] W.W. Carlson and J.A.B. Fortes. *COSMIC: A Model for Multiprocessor Performance Analysis*. Technical Report TR–EE 87–13, School of Electrical Engineering, Purdue University, December 1987.

[Dar87] F. Darema. *Applications Environment for the IBM Research Parallel Processor Prototype (RP3)*. Proceedings of the 1st International Conference on Supercomputing (E.N. Houstis, T.S. Papatheodorou, and C.D. Polychronopoulas, editors). Lecture Notes in Computer Science 297 (G. Goos and J. Hartmanis, editors). Springer-Verlag, Berlin, 1988.

[DeA87] P.J. Denning, and G.B.Adams. *Research Questions for Performance Analysis of Supercomputers*. Supercomputing (A. Lichnewsky and C. Saguez, editors). Elsevier Science Publishers B.V. (North–Holland), New York, 1987.

[Fly66] M.J. Flynn. *Very high-speed computing systems*. Proceedings of the IEEE, Volume 54, December 1966, pages 1901–1909.

[Fra87] C. Fraboul. *MIMD Parallelism Expression, Exploitation and Evaluation*. Supercomputing (A. Lichnewsky and C. Saguez, editors). Elsevier Science Publishers B.V. (North–Holland), New York, 1987.

[Fuj83] R.M. Fujimoto. *SIMON: A Simulator of Multicomputer Networks*. Report UCB/USD 83/140, University of California Berkeley, 1983.

[GoK87] M.M. Gorlick, and C.F. Kesselman. *Timing Prolog Programs Without Clocks*. Proceedings of the Symposium on Logic Programming. IEEE Computer Society Press, Washington, D.C., 1987.

[Hos86] T. Hoshino. *Invitation to the world of 'Pax'*. Computer, Volume 19, Number 11 (November 1986), pages 8–22.

[HoC87] A.A. Hough and J.E. Cuny. *Belvedere: Prototype of a Pattern-Oriented Debugger for Highly Parallel Computation.* Proceedings of the 1987 International Conference on Parallel Processing (S.K. Sahni, editor). The Pennsylvaina State University Press, University Park, PA, 1987.

[KaB88] A.H. Karp and R.G. Babb, II. *A Comparison of 12 Parallel Fortran Dialects.* IEEE Software, September 1988, pages 52–67.

[MKW84] J.R. McGraw, D.J. Kuck, and M. Wolfe. *A debate: Retire FORTRAN?* Physics Today, Volume 37, Number 5 (May 1984), pages 66–75.

[MKL87] H. Mühlenbein, O. Krämer, F. Limburger, M. Mevenkamp, and S. Streitz. *Design and Rationale for MUPPET: A Programming Environment for Message Based Multiprocessors.* Proceedings of the 1st International Conference on Supercomputing (E.N. Houstis, T.S. Papatheodorou, and C.D. Polychronopoulas, editors). Lecture Notes in Computer Science 297 (G. Goos and J. Hartmanis, editors). Springer-Verlag, Berlin, 1988.

[Nic87] A. Nicolau. *The Cornell Parallel Supercomputing Effort.* Experimental Parallel Computing Architectures (J.J. Dongarra, editor). Elsevier Science Publishers B.V. (North–Holland), New York, 1987.

[PeG86] J.K. Peir and D.D. Gajski. *Toward Computer–Aided Programming for Multiprocessors.* Parallel Algorithms and Architectures (M. Cosnard et al., editors). Elsevier Science Publishers B.V. (North–Holland), New York, 1986.

[PGW87] J.K. Peir, D.D. Gajski, and M.Y. Wu. *Programming Environments for Multiprocessors.* Supercomputing (A. Lichnewsky and C. Saguez, editors). Elsevier Science Publishers B.V. (North–Holland), New York, 1987.

[Pol88] C.D. Polychronopoulos. *Toward Auto-Scheduling Compilers.* Journal of Supercomputing, Volume 2, Number 3, 1988.

[Pra87] T.W. Pratt. *The PISCES 2 Parallel Programming Environment.* Proceedings of the 1987 International Conference on Parallel Processing (S.K. Sahni, editor). The Pennsylvaina State University Press, University Park, PA, 1987.

[PRG87] J. Purtilo, D.A. Reed, and D.C. Grunwald. *Environments for Prototyping Parallel Algorithms.* Proceedings of the 1987 International Conference on Parallel Processing (S.K. Sahni, editor). The Pennsylvaina State University Press, University Park, PA, 1987.

[Ree88] A.P. Reeves. *Programming Environments for Highly Parallel Multiprocessors.* Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications (G. Fox, editor). Association for Computing Machinery, New York, 1988.

[Saa87] Y. Saad. *On the Design of Parallel Numerical Methods in Message Passing and Shared Memory Environments.* Supercomputing (A. Lichnewsky and C. Saguez, editors). Elsevier Science Publishers B.V. (North–Holland), New York, 1987.

[SMS88] T. Schwederski, D.G. Meyer, and H.J. Siegel. *Parallel Processing.* Computer Architecture: Concepts and Systems (V.M. Milutinovic, editor). Elsevier Science Publishers B.V.

(North-Holland), New York, 1988.

[SnS86] L. Snyder and D. Socha. *POKER on the Cosmic Cube: The First Retargetable Parallel Programming Language and Environment.* Proceedings of the 1986 International Conference on Parallel Processing (k. Hwang, S.M. Jacobs, and E.E. Swartzlander, editors). IEEE Computer Society Press, Washington, D.C., 1986.

[SBD87] K. So, A.S. Bolmarcich, F. Darema, and V.A. Norton. *A Speedup Analyzer for Parallel Programs.* Proceedings of the 1987 International Conference on Parallel Processing (S.K. Sahni, editor). The Pennsylvaina State University Press, University Park, PA, 1987.

[Sto87] H.S. Stone. *High Performance Computer Architecture.* Addison-Wesley Publishing Company, Reading, MA, 1987.

[TaO80] R.N. Taylor and L.J. Osterweil. *Anomaly Detection in Concurrent Software by Static Data Flow Analysis.* IEEE Transactions on Software Engineering, Volume 6, Number 3 (May 1980), pages 265–278.

[Tri87] R. Triolet. *Programming Environments for Parallel Machines.* Supercomputing (A. Lichnewsky and C. Saguez, editors). Elsevier Science Publishers B.V. (North–Holland), New York, 1987.

[Wil86] K.G. Wilson. *The Gibbs Project.* Supercomputers: Algorithms, Architectures, and Scientific Computation (F.A. Matsen and T. Tajima, editors). University of Texas Press, Austin, TX, 1986.

[WuG88] M.Y. Wu and D.D. Gajski. *A Programming Aid for Hypercube Architectures.* Journal of Supercomputing, Volume 2, Number 3, 1988.