TOOLS FOR THE EFFICIENT DEVELOPMENT OF EFFICIENT PARALLEL PROGRAMS

Itzhak Nudler
Larry Rudolph
Institute of Mathematics and Computer Science
Hebrew University
Givat Ram, Jerusalem 91904, Israel
May 1986

ABSTRACT

In this paper we describe a software system that presents the programmer with many *views* of the parallel program. We describe the *objects* of a parallel program that we consider significant. Information concerning these objects are entered into a database. In addition, the program is automatically *instrumented* so that during execution, many events and status information can be collected and placed in the database. In particular, the system identifies all accesses to shared variables that may cause problems and whose values may be sensitive to the various scheduling policies or relative speeds of the processors. All such access are then automatically monitored. The programmer can then interact with the system to see various views of the program execution in order to determine where the bugs and performance bottlenecks occur. This tool is being developed as part of the Makbilan parallel processing project at Hebrew University.

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.

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 "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.

This paper describes one such tool that is being developed as part of the Makbilan parallel processing project at Hebrew University [RS]. This tool uses the approach of performance debugging through *programming for Observability*. Performance debugging is a system-wide issue, involving detailed knowledge of algorithm implementation, operating system, and parallel architecture. Hence provisions for observing status and performance events at all levels are required, together with facilities for combining semantic runtime information and development-time information, into a unique high-level representation. We refer to this feature as Programming for Observability [GS].

There are a few projects that are related to this research. The PIE system [SR] is the closest, although it is centered around a particular parallel programming language, MP. Monitoring [Sn82] and debugging [Ga] of distributed programs address a few of the issues that are of concern to us. A major component of our system is the visual display of information concerning the structure and execution of a parallel program. Related research in this area has be concerned with program animation [Re] and visual programming [RGR]. Finally, the reader is referred to a survey article on concepts in concurrent programming [AS] for many of the basic features of synchronization and parallel programming.

In this paper we describe a software system that presents the programmer with many *views* of the parallel program. Some of the views are only concerned with the static program structure while as others capture the dynamic runtime behavior. We describe the *objects* of a parallel program that we consider significant. Information concerning these objects

are entered into a database. The database is then used to construct the static program views. In addition, the program is automatically *instrumented* so that during execution, many events and status information can be collected and placed in the database. In particular, the system identifies all accesses to shared variables that may cause problems and whose values may be sensitive to the various scheduling policies or relative speeds of the processors. All such access are then automatically monitored. The programmer can then interact with the system to see various views of the program execution in order to determine where the bugs and performance bottlenecks occur.

We first describe the types of parallel program objects that are of interest to our system. The subsequent two sections (sections 3 and 4) enumerate the monitoring primitives used to accumulate information relevant to these objects and how this information is to be presented to the user. Section 5 presents a general scheme to identify the potentially harmful accesses. The scheme requires the identification of all processes (or parts thereof) that may be executed concurrently. Specific details are then presented concerning the currently supported programming language, Occam+, in section 6, and in section 7, an example application is presented.

2. PARALLEL PROGRAM OBJECTS

In this section we describe the objects or aspects of a parallel program that we wish to monitor. We have attempted to enumerate those universal and essential qualities that should be present in most parallel programs, regardless of their implementation language. Note that we assume that the program will be written in some parallel programming language and thereby explicitly exclude systems that automatically attempt to detect the parallelism in a program written in a sequential programming language.

The activity of observing events (the unit of information for observability) is termed *monitoring*. The agent executing the monitoring mechanism is a *monitor*. In this paper the term monitoring will be used to denote the process of collecting static and dynamic information concerning a parallel program.

2.1. Processes Every parallel program has some notion of process (sometimes called task, activity, or thread of control). During the execution of a parallel program, parallel work is performed. This parallel work consists of a set of processes that are executing in parallel. In some languages, the processes are specified directly by the programmer (e.g. [BH]) and in others it is indirectly specified (e.g. Concurrent Prolog [Sh] where some operations can be carried out in parallel, such as unifications or parameter evaluation). In either case, one is concerned with the creation, destruction, halting, and restarting of processes, as this indicates the performance of the machine.

A process may either be a templet process or an instantiated process depending on the context: the program specification or the execution of a program. We assign a static id to each unique process templet or static process. Each dynamic process, i.e. an instantiation of some static process during the program execution, is assigned a unique dynamic process id.

2.2. Synchronization Parallel models of computation can be classified as either SIMD or MIMD [Fl]: Single Instruction stream/ Multiple Data stream or Multiple Instruction stream/ Multiple Data stream. In the former, all the process execute in lock step, synchronizing after each operation. In the latter, synchronization or the gathering together of the execution of processes so that no one proceeds past a certain point in the computation until the execution of other processes have reached a specified point. Some parallel programming languages provide explicit mechanisms for synchronization whereas in others it is implicit at the end of a parallel construction. In general, one can consider closed parallel constructs, such as begin/end or parallel iteration loops, where processes are created at the start of the construct and terminated at the end. Open constructs, in contrast, are much more unwieldy, such as the rendezvous mechanism of Ada [Ada], and their occurrence is of great importance.

2.3. Basic Sequential Blocks Each process consists of a set of *basic sequential blocks*. Each such block is a sequence of statements (code segment) that do not cause the creation of new processes nor contain any rendezvous points. In other words, it is the sequence of code between two synchronization points.

Note that the term is similar to that of a basic block (as used in compilers) and should not be confused the term block used in programming languages. We distinguish between static and dynamic basic sequential blocks depending upon if they are part of a static or a dynamic process.

2.4. Atomic Compound Actions Every parallel processor must support atomic operations. That is, some operation that cannot be externally decomposed into more primitive actions [FF]. At some level of abstraction, the programmer also assumes the existence of atomic operations. Often, individual loads and stores to simple variables are assumed to be atomic, however, more complicated operations may also be supported. In particular, a machine or a language may support particular instances of Read-Modify-Write instructions. The well known test-and-set operation is an example of the latter. Even more complicated operations may be assumed to be atomic, such as an individual statement in the programming language or an operation on a complex, multi-field shared data structure.

2.5. Communication At various points in a program, a process may communicate with other processes. This communication may take the form of an explicit send/receive mechanism or may be more implicit as in the case of access to shared memory. The points in the program where such communication occurs are interesting objects.

2.6. Operating System Calls Various invocations of operating system routines may be significant to the execution of a parallel program. Examples of significant events include access to external devices, access to a terminal, spawning of new processes. These are significant events in a program. In a uniprocessor, the time of their occurrence with respect to the whole computation might be interesting but rarely effects the overall performance of the program. In parallel processing, this is not so. The simultaneous request for a particular operating system function by many processes may be a cause of concern since it may significantly degrade the performance.

2.7. Virtual Processors Some parallel programming languages give the programmer explicit control as to how processes or activities are allocated or mapped to the processors. We call such processors virtual, since they are at the program level. During runtime, the virtual processors are mapped onto real processors. The mapping, if supported by the language, is a significant event [JS], [Occ].

2.8. User Defined Events In order to monitor the performance of a parallel program, the user may want to note the execution or occurrence of an execution of an event. Later in this paper we shall describe this in more detail. However, here it is worthwhile noting that such events are significant objects in a parallel program. In the worse case, the event reporting may effect execution, if it is not supported by the underlying hardware.

2.9. Relations between Objects It is not only important to note the existence of the objects described above, it is also important to understand how these objects relate to each other. For example, which processes are synchronizing, what is being sent between one process and another.

3. MONITORING PRIMITIVES

The monitoring mechanism in our system is composed out of two components: *sensors* and a *data-base*. The sensors are logical entities which are responsible for information collection. The information collected by the sensors is recorded in the data-base of the monitoring mechanism. This data-base provides the tools for organizing and manipulating the collected information.

We record information concerning the significant objects of a program both at compile time and at runtime. At compile time, the static information is recorded. Hooks are installed into the code so that the dynamic information can be collected at runtime. Each point in the program where an object occurs is recorded in a database. The position in the source code is recorded so that the user can be presented with the surounding context, at a later point in time. As much significant information is recorded depending on the type of object.

It is important that the collection of runtime information be as non-intrusive as possible. Since a parallel computation may generate a very large number of events that are to be monitored, one desires substantial hardware support for monitoring: a parallel processor may be needed to monitor the parallel processor.

Integration of monitoring information from compile time and runtime and its subsequent graphical interactive display gives our system substantial power. Various levels of monitoring can be presented to the user, as described in a later section, once the particular runtime actions have been recorded. We plan to extend our system to also provide monitoring information about low level hardware functions, such as cache hit ratios, page faults, and bus loads. A user could then discover, for example, how the bus load changes as a function of the status of some shared data structure.

4. INDETERMINACY CONSIDERED HARMFUL

One of the major sources of errors in parallel programs arises from the fact that the execution of most parallel programs is not deterministic. The relative speeds of the processors, contention when accessing shared resources, and the scheduling policies of the operating system have a major effect on the execution of a parallel program. While it is expected that these factors affect the performance of a parallel program, it is often the case that they also effect the results of the computation. Two executions of the same program with the same input may follow different computation paths because the relative speed of the processors differ during each execution. Such nondeterministic behavior is considered harmful.

Except when required, it is better for a program to be determinate than indeterminate. The non-deterministic behavior required by the program should be explicitly specified through the appropriate constructs. We are concerned with identifying and monitoring parts of a program in that may be a source of non-explicit indeterinacy. We refer to such program parts as being *harmful*. Thus, there are harmful code sections, harmful memory access, harmful variables, etc.

Recall our notion of basic sequential block identifying a piece of sequential code between synchronization points. If we assume that the program is sequentially consistent ([La] and [KRS]), then any interleaving of the instructions in one basic sequential block with the instructions in another concurrent basic sequential block, is legal. The many possible interleavings may give rise to non-explicit indeterminancy.

4.1. Harmful Actions

Shared variables play a crucial role in parallel processing. Communication among activities on a shared memory multiprocessor is often more efficient when memory is manipulated directly by the activities than when information is exchanged solely through interprocess message passing mechanisms. Shared variables also make it easy to share information by many processes over long periods of time, such as in a symbol table. (See [Sn] for additional arguments in favor of shared variables.) On the other hand, in our experience, indiscriminate use of shared memory promotes hard-to-find bugs. The indeterminacy of the execution can make it difficult to determine *a priori* if one process will read a shared variable before or after it has been updated by another process.

A *harmful shared memory access* is one that introduces non-explicit indeterminacy. An access that stores a value into a shared variable that may be accessed concurrently by another process is considered harmful. That is, let A denote a store to a variable and B denote any access to that sane variable by another process. If, for a given input, it is possible for A to occur before B in one execution and for B to occur before A in another execution, then A and B are considered harmful accesses. Similarly, we refer to the shared variable that is the target of a harmful access as a *harmful shared variable*.

Message passing or synchronous communications between processes can also be harmful. The sending of a message can be considered to be like a store to a shared variable and the receiving of a message is analogous to the reading of a shared variable. Thus, if two processes send a message to a third process and they are all executing concurrently, the messages may be received in different orders, introducing indeterminacy.

We would like to protect the programmer from harmful accesses. There are three possibilites:

(i) Disallow the possibility of harmful accesses by syntaxtic mechanism. That is, enforcement at the language level.

(ii) Runtime enforcement.

(iii) Compile-time and Runtime warnings.

The first choice is usually considered too restrictive and the second may be too expensive. In this section, we show how to support the third choice. This allows the programmer the freedom to depend on his detailed understanding of the program logic and control flow to ensure that there are no harmful access, while at the same time, providing a mechanism to tell him when he is wrong. The situation is analogous to runtime array bounds checking present in many sequential programming languages.

4.2. Detecting Harmful Actions

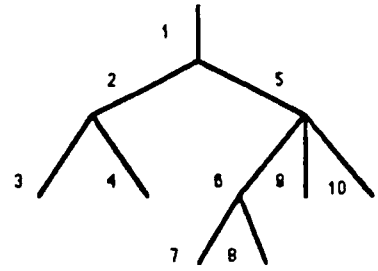
Throughout the rest of this paper, we consider a restricted type of parallel program in which there are only closed parallel constructs. That is, we exclude rendezvous and similar barrier synchronizations, and do not consider synchronous message passing to be an adequate mechanism to enforce synchronization or to guard access to shared variables.

Our overall strategy consists of three phases. The first phase occurs during program compilation. The parallel program is analyzed to determine the actions that may be harmful. Code is then added to the program so that these critical actions can be monitored. The second phase occurs during runtime. The critical actions are checked to see if they are indeed harmful or even if they may be harmful. That is, it may be the case that although during the current execution, the actions are not found to be harmful, however, they could be harmful during different executions. Information concerning the critical actions is recorded in a database. The third phase occurs after program execution and consists of presenting the results of the monitoring to the user.

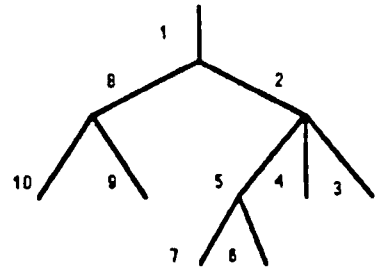
The underlying mechanism in the first two phases is a special method of assigning labels (ids) to basic sequential blocks so that harmful actions can be quickly identified. Static labels are assigned during the first phase and dynamic ones during the second phase. In both cases, the same scheme is followed. We describe the scheme for detecting harmful access to shared variables (and harmful shared variables). Harmful message passing actions are similarly handled.

In order to motivate our mechanism, consider simple parallel programs with no message passing, only closed parallel constructs, and no procedure calls. Then, for each variable, we find all access to it in concurrent basic sequential blocks. If one such access is a write, then the shard variable may be harmful and the concurrent access to it may also be harmful. Thus, the problem reduces to determining which basic sequential blocks may be executed concurrently.

Consider a process invocation graph before any of the parallel constructs have terminated. The resulting structure is a tree, with each edge representing a basic sequential block and each vertex representing a parallel fork. From this tree, it is easy to see that two processes, X and Y, may not execute concurrently, iff one is a direct ancestor of the other.



A Process Execution Graph Before Joins
Figure 4.2.1. Preorder (left to right) Numbering



A Process Execution Graph Before Joins
Figure 4.2.2. Preorder (right to left) Numbering

We label the edges in the tree in preorder fashion ([AHU]), visting the children in a left to right ordering (see Figure 4.2.1). (Note that usually one labels vertices; we assign the same label to the edge that points to a vertex.) It is not hard to see the following:

Claim: Let u and v be two edges in a process tree. E.label(u) > E.label(v), if and only if one of the following is true

(i) v is a direct ancestor of u

(ii) v is a left sibling of some ancestor of u.

In a similar fashion, label the edges in the tree in preorder fashion but visiting the children in a right to left order (see Figure 4.2.2) giving H.labels[1]

Claim: Let u and v be two edges in a process tree. H.label(u) > H.label(v) if and only if one of the following is true

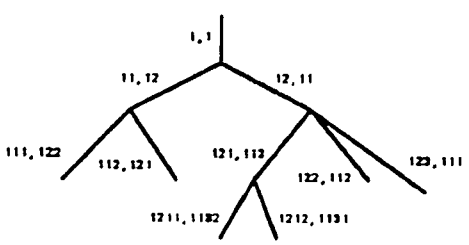
(i) v is a direct ancestor of u

(ii) v is a right sibling of some ancestor of u.

From the two claims, a third claim follows:

Claim: Let u and v be two edges in a process tree. E.label(u) > E.label(v) and H.label(u) > H.label(v) if and only if v is a direct ancestor of u

The E.labels and H.labels can thus be used to determine if two basic sequential blocks are concurrent. To implement this idea, however, we must overcome two problems: the labeling must be performed dynamically, since we do not know how deep it will go, and the closing of parallel constructs must be handled.



A Process Execution Graph Before Joins
Edge labeled as E.label , H.label
Figure 4.2.3. An example of E.labels and H.labels

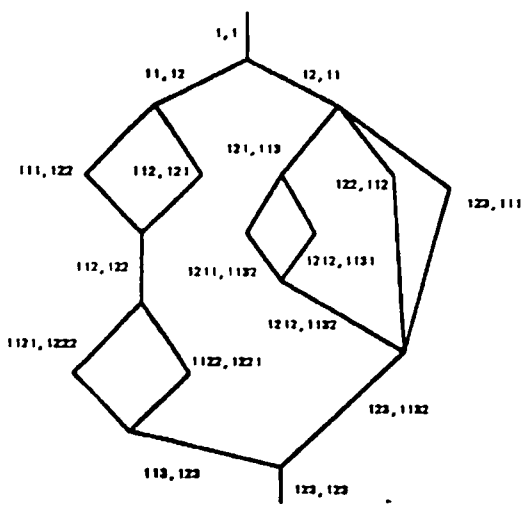
labels in our scheme will be strings of arbitrary length. Let $C_1, C_2, \ldots, C_k$ be k processes in a parallel construct of process p, then we assign labels recursively as follows:

$$\text{E.label}[C_i] = \text{E.label}[p] \,@\, i$$
and
$$\text{H.label}[C_i] = \text{H.label}[p] \,@\, (k-i+1)$$

where @ is the append operator (see Figure 4.2.3).

labels are compared in a lexigraphical ordering: compare labels digit by digit starting from the left, and if these do not match then the shorter one is smaller.

The second problem is how to assign process labels after a parallel construct has terminated. The code up until the parallel construct is one basic sequential block with a particular label. A set of basic sequential blocks are then created and assigned labels as just described. After the parallel construct, there is a new basic sequential block that must be assigned a label. The simplest method is to use the maximum E.label and H.label of all the basic sequential blocks that just terminated. These labels are thus greater than all labels in the processes derived from the previous parallel construct. Unfortunately, the length of the labels begins to grow unreasonably fast; at least by one digit for each parallel construct. However, after the first join, subsequent joins need only increase the last digit of the previous labels (see Figure 4.2.4). These labels



A Process Execution Graph
Figure 4.2.4. An example of E.labels and H.labels after Joins

[1] The letters E and H denoting the type of labels are used because English is written from left to right and Hebrew is written from right to left.

4.3. Compile Time Issues

During compile time, the widths of a parallel construct may not be known, as in the case of a FOR ALL statements with a variable limit. In such a case, we just assume a value of two, since we only want to determine possible concurrency.

Procedures calls introduce an added complexity. Procedures may be invoked from many different parts of the program, variable may be passed by reference causing an aliasing problem, and procedures may be recursive. If a procedure is called by two processes running concurrently, then all the variables it reference may be changed.

Our approach during compilation, is to view procedure calls as processes invocations with the parameters that are passed by reference substituted for the formal parameters. Care is required to avoid infinite expansion for recursive calls. Upon encountering a procedure call, we check all previous occurrences of the procedure on the process invocation graph. If any of them are direct ancestors of this call and with the same variable parameters passed, then the procedure is not expanded; we did all possible static checking. A preprocessor is used to perform this expansion.

4.4. Runtime Issues

During program execution, we wish to detect both harmful and potentially harmful actions. Potentially harmful actions discovered at runtime are ones that may be harmful although there is no evidence during the execution that they are indeed harmful.

Our detection mechanism works as follows. We record, for each write access to a critical variable, the dynamic labels of the basic sequential block executing the write. Although we use the term variable, during runtime, we use the actual address of the variable. This avoids problems with aliasing variables passed by reference and with access to components of larger data structures such as arrays. We only keep the labels of the last write access. Each write or read access causes a runtime check. If the basic sequential block of the last write to the variable is still executing, then the accesses are harmful. This check is performed by searching the list of currently executing processes to see which basic sequential blocks are still executing. In our simulation of a parallel processor, this check is easy. A real parallel will require hardware support for this action, since the offending basic sequential block may terminate during the execution of the check. Potentially harmful actions are detected by just checking to see if the previous write was performed by a direct ancestor or not.

5. VIEWS

The monitoring mechanism data-base provides a tool for associating information from development time and runtime and providing a variety of *views* on the behavior of a parallel program. *A view* is a graphical representation of some information which was collected by the monitoring mechanism.

Currently the system provides four views: static process information, dynamic invocation tree, activity execution times, and harmful shared variables view.

(1) Static Process Information - There are two instantiations of this view. One presents the graph of relations among processes at development time. In this graph, one node is associated to each static process specified in the program. Lines between nodes represent parent/child relations. A node with self-loop indicates a recursive process call.

The second instantiation presents the control flow graph of the program in terms of basic sequential blocks. The graph is a directed one, each edge representing a basic sequential block. Verticies are either fork or join points, representing synchronizations between processes. That is, given a system with only closed parallel constructs, each vertex is either the creation of a set of processes or the termination of a set of processes. With open parallel constructs, the verticies represent rendezvous. Note that this graph presents possible relationships among processes and basic sequential blocks. These relations may not hold at runtime, and in fact, the basic blocks are only templets, as there may be many instantiations of a basic sequential block at runtime.

(2) Dynamic Invocation Tree - This view represents the dynamic invocation sequence of processes (i.e., which processes initiated which other processes). Again, there are two modes indicating whether processes or basic sequential blocks are to be displayed. The user controls the nesting depth of the display. For example, at the highest level, only those processes or basic sequential blocks created by the main program. We use the term *zoom in* or *zoom out* to indicate the changing of the level of nesting displayed.

(3) Activity Execution Times - This view displays the execution times of activities in the form of a bar graph with a separate indicator for each instance of an activity. The total execution time of the experiment is divided into a number of slots, for each slot during which an activity is active, the bar is blackened. The bar graph gives a global view of the parallelism of the application as a function of time.

(4) Potentially Harmful Variables - This view displays the dynamic reference to shared variables that are or may be harmful. The actual implementation of this view is part static and part dynamic.

6. OCCAM and OCCAM+

Occam [Occ] is a programming language designed to support concurrent applications in which many parts of a system operate separately and interact. Its primary application is for programing transputers. It is a very simple language making it easy to describe as well as to analyze. Occam+ is an extension encorporating support for shared variables. Because of its simplicity and power, it was chosen as the parallel programming language to be instrumented. In this section, we describe the basic features of Occam and its extension.

6.1. Basic Occam Features

Occam enables the programmer to express a program in terms of concurrent processes which communicate by sending messages through communication channels. This has two important consequences: it gives the program a clear and simple structure and it allows the program to exploit the performance of many computing components, as each concurrent process may be executed by an individual processor.

Occam can capture the hierarchical structure of a system by allowing an interconnected set of proceses to be regarded from the outside as a single process. At any level of detail, the programmer is only concerned with a small and manageable set of processes. A process can be a single statement, a set of statements, or a set of other processes.

Communication between processes take place through a channel, which is the only mechanism by which processes can transfer information among themselves. The channel is a one-way communication element between two concurrent processes. It may be viewed as a bounded buffer consumer/producer mechanism, with a buffer size of zero. That is, communication is synchronized such that output cannot take place unless the channel is free, and input cannot take place unless the channel is suplying a value. Thus both the input and the output processes must be ready before transfer can take place. There is no implicit buffering associated with channels. The channels operators are input (?) and output (!).

To control the order of execution of the processes, Occam defines three basic constructs : sequential (SEQ) block, where program elements are executed in sequence; parallel (PAR) block, where program elements are executed in parallel and forming the basis of the closed parallel constructs, and alternative (ALT), in which one program element is selected from a set and providing explicit nondeterminism in the language. The conventional 'WHILE' statement achieves repetitive execution of a process, and the conventional 'IF' statement executes the first component process for which some evaluated expression is TRUE. The SEQ, PAR and IF constructs may be replicated, giving, in particular, a closed ForAll parallel construct.

It should be noted that concurrency in Occam can take place at the lowest level of the language. Single statements are processes which can run concurrently. The statements of the language are joined together by constructs.

A name can be given to any process allowing that process to be used by name when it is required. The PROC declaration introduces an identifier to name a process. The named process may have formal parameters. These parameters are replaced by the actual parameters of the substitution, before the named process is executed. Occam has standard, algol-type scoping rules for declaration and referencing of objects.

6.2. Occam+ Occam+ is an extention to the standard Occam language [MS]. Whereas, Occam was designed to program a set of interconnected processes, it has been extended to support programs for shared memory parallel processors. Using the standard scoping rules already part of Occam, Occam+ allows variables to be referenced by any process within their scope. In addition, variables can be passed by reference to declared processes (i.e. procedures) providing yet another mechanism to concurrent processes to shared access to variables. Occam+ also allows declared processes to be recursive.

Despite their similarity, Occam+ programs are expected use shared memory as the primary communication mechanism between processes and use the synchronous message passing facility as a low level synchronization mechanism.

6.3. Development Time Monitoring

The sensors which are responsible for collecting development time information are hooked in the parser of the Occam compiler. These sensors are actions which were added to certain grammer rules. It is important to emphasis that no changes were made in the grammer. This means that these modifications can be done with any other programing language. We simply need to take the parser of the programing language compiler and add to it actions (in the appropriate places) which implement development time sensors.

The information collected during development time concerns virtual processes and shared variables. For each virtual process we record its : parent id, childs list, name, type, start line and end line. Each process is given a unique process id which distinguish it from all other processes. The type of a process may be either: assignment, input, output, seq construct, par construct, etc. If the process is of type process call, the name of the process is the name of the called process, otherwise the process as no name. A hierarchy of processes is created, forming the structure of a graph.

We look at Occam programs as if they are composed from a set of routines corresponding to each of the named processes and to the main process. The creation of the graph of processes is done separatly to each of these routines.

Also the information concerning the control flow graph used to detect harmful access and variables is collected during development time. This information is used to associate information collect at runtime with the program specification.

6.4. Runtime Monitoring

Some of the sensors which are responsible for the collection of runtime information are hooked into the code of the program. These sensors (i.e., these code installations) are generated by development time sensors. The other runtime sensors are inserted into the runtime library. The runtime sensors are responsible for collecting information concerning execution times, instantiations of virtual processes and accesses to shared variables.

7. AN EXAMPLE APPLICATION

In this section, we demonstrate various features of our system by describing sample application. We chose the straightforward paralellization of the quicksort because of its familiarity and simple recursive structure. Quicksort starts

```
Var tab[max]:
Var maxdepth:
Extern Proc chassign(chan c, Value ch, fd):

Proc partition(Var tab[], Value left, right, Var mid)=
        Var     Smaller[max], Bigger[max]:
        Var     SmalerSize, BiggerSize, n, median:
        n := (right - left) / 2
        median := (tab[left] + tab[right]) / 2
        Par i = [0 For right]
                If
                        tab[i] < median
                                Smaller[i] := tab[i]
                        tab[i] >= median
                                Bigger[i] := tab[i]
        pack2right(Smaller, n, tab, left, SmallerSize)
        pack2left(Bigger, n, tab, left+SmallerSize, BiggerSize)
        mid := left + SmallerSize

Proc qsort(Var tab[], Value n, curlevel, Var depth)=
        Var mid:
        Seq
                If
                        n > 1
                                Seq
                                        partition(tab, n, mid)
                                        Par
                                                qsort(tab[0], mid, curlevel+1, depth)
                                                qsort(tab[mid], n-mid, curlevel+1, depth):
                        n = 1
                                If
                                        depth < curlevel
                                                depth := curlevel


Seq                                     -- main process
        ReadInput(tab)                  -- read the numbers to be sorted
                                        -- into the array tab
        maxdepth := 0
        qsort(tab, max-1, 0, maxdepth)
        PrintSortedArray(tab)           -- print the sorted array
```

Figure 7.4. Quicksort Program Fragment

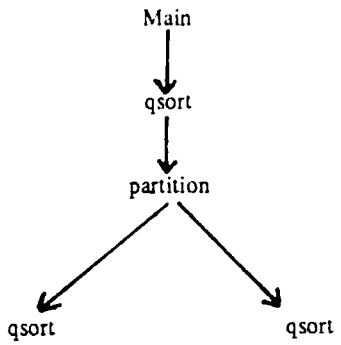


Figure 7.5. Control Flow Graph of Quicksort, Zoom Level = 2

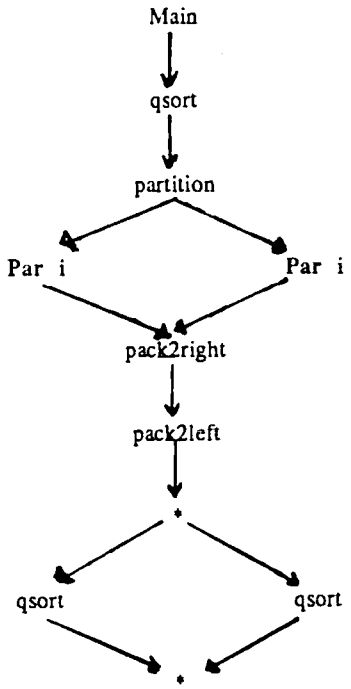


Figure 7.6. Control Flow Graph of Quicksort, Zoom Level = 3

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 in parallel all elements smaller than this value into set Smaller, and those equal to or greater than this value into set Bigger. The parallelization recursively applies quicksort to the subsets Smaller and Bigger in parallel. The shared variable maxdepth is used (erroneously) to record how deep the recursion goes.

Figure 7.4 shows the interesting parts of the Occam+ program for quicksort. Two control flow graphs at different zooming levels are shown in Figure 7.5 and Figure 7.6. The user can point to a part of the graph with the mouse, and the system will show the various information about the process which is represented by this node. This information includes the process' code, its name and type, the file in which it appears, its start line in this file, etc.

There are two shared data structures in the program: the array tab and the variable maxdepth. The first is the array of values to be sorted, and the second is used just to see how deep the recursion goes. At first glance it appears as if the array tab is written and read concurrently by many processes. At runtime, however, we detect that this is not the case and that the array Tab is a harmless array. On the other hand, the accesses to the variable maxdepth are harmful, and will be so detected, since it is written and read concurrently by many different processes.

8. CONCLUSION

We have described parts of a system under development. Currently, the system has been implemented on a VAX/780 machine running the 4.2 Berkely version of the Unix operating system. In the near future the system will be implemented on a Sun 3 workstation (also running the 4.2 Berkeley version of Unix). The window system of the Sun 3 will help to implement the graphics features of the system. The language we choose was Occam+, which is an extension of the standard Occam.

In place of a parallel processor, we are currently using a simulator that executes compiled Occam+ code on a uniprocessor with a primitive scheduler to switch executions between processes. The system has already provided insights as to what are the important actions that should be monitored. In addition, we have a better understanding of the future needs for such a system.

In particular, the database appears to be a significant bottleneck. It is crucial for the runtime information to be assimilated as fast as possible. Moreover, the user display requires a complex access to the wealth of data accumulated from each execution.

We are investigating various methods of performing the monitoring via hardware sensors. We hope to integrate the monitoring within the parallel processor currently being designed at Hebrew University.

In our estimation, the weakest part of the current system is the graphical display of information. Currently the system only provides a set of predefined views. Being a feasibility project, we have focused our attention on the exploration of how all the pieces of the system interact. Future developments of this system may provide the user with high level tools fo programming his own views.

References

[ADA] American National Standard Reference Manual for the Ada Programming Language, ANSI/MIL-STD 1815a-1983, February 1983.

[AHU] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "The Design and Analysis of Computer Algorithms," Addison-Wesley Publishing Company, 1976

[AS] G. R. Andrew, and F. B. Schneider, "Concepts and Notations for Concurrent Programming," ACM computing surveys, vol. 15, no. 1, March 1983, pp 3-43.

[BH] Brinch-Hansen, "The Programming Language Concurrent Pascal," IEEE Trans. Softw. Eng., vol. SE-1 no. 2 (June 1975), pp 199-207

[FF] R. F. Filman and D. P. Friedman, "Coordinated Computing: Tools and Techniques for Distributed Software," McGraw-Hill Book Company, New York, 1984.

[Fl] M. J. Flynn "Very High Speed Computing Systems," IEEE Trans. Computers Vol 54, 1966, pp 1901-1909

[Ga] J. Gait "A debugger for Concurrent Programs," Software Practice and Experience, vol. 15, June 1985, pp 539-554

[JS] A. K. Jones and K. Schwans, "TASK Forces: Distributed Software for Solving Problems of Substantial Size," In Proceedings of the 4th International Conference on Software Engineering. ACM, IEEE, Munich, W. Germany, September, 1979.

[KRS] C. P. Kruskal, L. Rudolph and M. Snir, "Efficient Synchronization on Multiprocessors with Shared Memory," To be published in Proc. 5th ACM Symp. on Principles of Distributed Computing, August 1986.

[La] L. Lamport, "How To Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," IEEE Transactions on Computers, vol. C-28, pp. 690-691, 1979.

[MS] D. Malki and G. Shwed, "Occam+ Reference Manual," manuscript, Institute of Math and Computer Science, Hebrew University, Jerusalem, 1985.

[Occ] *INMOS Limited: Occam Programming Manual,* Prentice Hall International Series in Computer Science, 1984.

[Re] S. P. Reiss "Graphical Program Development with PECAN Program Development Systems," Proc. ACM Symp. Practical Software Development Environments, Pittsburgh, PA, April 1984.

[RGR] S. P. Reiss, E. J. Golin, and R. V. Rubin, "Garden: A system for prototyping Visual Languages," Proc. IEEE Symposium on Visual Languages, June 1986.

[RS] L. Rudolph, and M. Snir, "The Makbilan Project," manuscript, Institute of Math and Computer Science, Hebrew University, Jerusalem, 1985.

[Sh] E. Y. Shapiro, "A Subset of Concurrent Prolog and its Interpretor," Technical report TR-003, ICOT Institute for New Generation Computing, Tokyo, Feb. 1983, also CS83-06, Dept. of Applied Math., Weitzmann Institute of Science, Rehovot, Israel.

[Sn] M. Snir, "Communicating With Shared Variables in Multiprocessors: Hardware-Software Tradeoffs," This proceedings, 1986.

[Sn82] R. Snodgrass, "Monitoring Distributed Systems: A Relational Approach," Technical Report, Carnegie-Mellon University, Pittsburgh, PA, December 1982.

[Th] D. Thalmann, "An Interactive Data Visualization System," Software Practice and Experience, vol. 15, March 1984, pp 277-290