ON THE PARALLEL STRUCTURING OF RESOURCE MANAGEMENT

Robert W. O'Dell

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-85-27 November 1985

Chapter 1

Introduction

The execution of parallel programs on parallel computer architectures will require scheduling, coordination and resource management. These activities will themselves consume computer resources. Centralized resource management will become the performance bottleneck when large numbers of processes must be scheduled and managed.

This thesis studies parallel structuring of resource management. The environment of the study will be the Run-time System of the Computation Structures Language (CSL) parallel programming system. The approach has been to design and evaluate a parallel version of the resource management functions of the CSL Run-time System.

The parallel structuring of resource management is essential in order to alleviate the well-known serialization bottleneck [Madnick 74]. This bottleneck occurs whenever a large number of parallel processes must access a shared resource through a serial monitor. In such cases the addition of more processes to the system will cause no increase in overall performance and may even cause it to be worsened.

The CSL Run-time System forms a serialization bottleneck when large numbers of processes must be managed. The design of the Parallel Run-Time System (PARTS) consists of a method of parallelizing the resource management functions of the original CSL Run-time System. The parallel structuring of PARTS consists of a structuring of the system tables such that they support access from multiple copies of the Run-time System, along with specifications for these access procedures.

The evaluation procedure used in this thesis is to define and code simulation models for a single RTS and for differing amounts of parallelism within PARTS. The data used to parameterize the models is taken from a working RTS on a real system where appropriate. The evaluation is performed to show how much parallelism is appropriate for PARTS under what conditions. The results of the evaluation are then used to formulate a detailed scheduler.

The simulation system, PAWS, was used to generate the model of the system. The program was written in PAWS and executed on the VAX 11/750 system operated by the Texas Reconfigurable Array Computer development group at The University of Texas at Austin. The data to parameterize the model came from measurements of the performance of the CSL RTS on the Dual Cyber 170/750 systems operating under the UT-2D operating system.

The thesis is organized as follows. Chapter 2 discusses the CSL RTS and how its parallelism can be represented using a Computation Graph. Chapter 3 gives a more detailed look at how the system tables are partitioned and accessed. Chapter 4 discusses previous work. Chapter 5 discusses the basic scheduling algorithm and analyzes some of its properties. Chapters 6 and 7 review the results of the performance evaluation, formulate a more detailed scheduler, and draw conclusions based on these results.

Chapter 2

A Computation Graph for the Run-time System

The design and analysis of PARTS is prerequisited by the recognizing of parallelism in the original serial version of the CSL Run-time System (RTS). This chapter details how the parallelism was found and exploited.

There exists no formal methodology for finding parallelism in algorithms. Some work has been done to find parallelism directly (and sometimes automatically) from ordinary programming languages such as Fortran [Kuck 77]. This usually entails an analysis of dependency relationships within program loops. Although speed-ups on the order of six or seven are attainable, a basic limiting factor exists in this method -- the algorithm was already mapped to a serial language representation before the parallelism was extracted.

Many applications algorithms, however, do display parallelism on a larger scale. For instance, image processing algorithms can be made to support parallelism by partitioning the data between units of computation. Other algorithms exploit parallelism by partitioning the functions that the computation performs. This lends itself especially to pipelined forms of parallelism. Parallel structuring for many algorithms, however, can be elusive at best.

We concern ourselves here with parallel structuring of operating systems type algorithms -- in particular the resource management algorithms of the CSL Run-time System. The algorithms explicitly control the progress of the parallel computation defined in CSL and are discussed in Section 2.3.

Since the function of the CSL RTS does not easily map into parallel structuring, a more

general approach was taken. In the design of PARTS, parallelism was extracted by analyzing the CSL RTS in the context of a computation graph. A general look at computation graphs follows.

2.1 Computation Graphs

A definition of computation graphs is given in [Browne 85]. A computation graph is a directed graph where the nodes represent binding of operator objects to data objects and arcs represent dependency relationships between nodes. A sample computation graph is shown in Figure 2-1. In this example, node C may begin as soon as nodes A and B have completed.

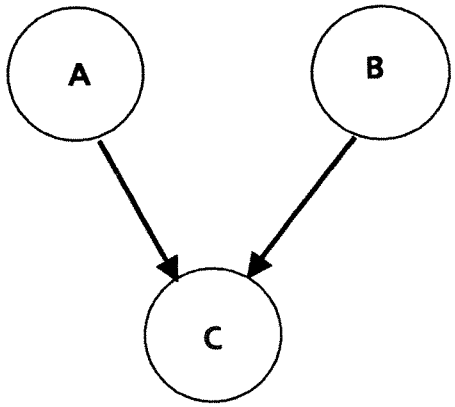

Figure 2-1: Simple Computation Graph

A dependency relationship is either a producer/consumer relationship or a synchronization relationship. There are two types of synchronization relationships -- those that specify a precedence between nodes, and those that specify only that concurrent execution is not valid.

Dataflow graphs [Arvind 77] are subsets of computation graphs. Unlike dataflow graphs, computation graphs do not specify what type of objects must traverse the arcs. Traversing objects may be operator objects, control signals, or data objects. Also, computation graphs need not be static. The possible result of a computation at a node may be the creation of a new node or an entire new subgraph.

An algorithm may be represented by many different computation graphs. For instance, every algorithm may be represented trivially as a single node with inputs and outputs. Conversely, an algorithm may be represented as a multitude of instruction level operator/data object bindings with a very complex dependency structure.

The execution of an algorithm formed in terms of a computation graph is simply a traversal of that graph. If serial execution is used, then a single thread of computation proceeds through the graph in any order that preserves the dependency relationships. If parallel execution is used, any two nodes not dependent on each other may be computed in parallel.

In order to traverse a computation graph on a real machine, a mapping must be made between the nodes of the graph, and schedulable computation structures on the machine. This mapping determines the performance of the algorithm on the machine. Therefore, the mapping which provides the best performance may not exploit all the parallelism available in the computation graph. In fact, the overhead associated with managing parallel computation on a real machine may be so great as to provide much worse performance than a serial execution! Consider for example the execution of single instructions, each residing in its own UNIX process. The process management overhead here far outweighs any benefit of parallelism.

In summary, the development of computation graphs for an algorithm may be used as a tool to uncover inherent parallelism within that algorithm. It is, however, the mapping between computation graph and machine that determines how quickly the algorithm will run on that machine.

In the following section we look at the CSL language and show how it specifies a computation graph and its traversal.

2.2 Overview of CSL

Computation Structures Language (CSL) is a language for expressing the synchronization and communication requirements of a set of tasks that form the nodes of a computation graph [Browne 82]. CSL specifies a traversal of a computation graph. It does not, however, perform any computation associated with the algorithm.

The CSL program follows directly from the organization of the tasks. The user of CSL first organizes his algorithm in terms of separate, independently compilable pieces of ordinary serial code (such as Fortran). These tasks know nothing of the existance of any other tasks in the algorithm. Any synchronization necessary between the execution of these tasks, and any communication requirements between tasks are specified in the CSL program. Synchfonization may be specified by conditions defined within CSL or the serialization of operations within CSL. Communication may be specified through the declaration of channels and shared variables.

A sample CSL program is shown below.

```
JOB Sample-CSL;

CONSTRUCT
    TASKS
          T[i]: File1;  RANGE i=1 to 2;
          T3  : File2;

    CHANNELS
          Ch[i] : DATACHANNEL from T[i] to T3;
                        RANGE i=1 to 2;

END; (* Construct *)

BEGIN (* Executable Code *)

  COBEGIN
       // EXECUTE T[1];
          SEND X to Ch[1];
       // EXECUTE T[2];
          SEND Y to Ch[2];
       // RECEIVE X from Ch[1];
       // RECEIVE Y from Ch[2];
  COEND;
  EXECUTE T3;
```

```
END; (* CSL Program *)
```

The preceeding program traverses the computation graph shown earlier in Figure 2-1 where nodes A and B are equated with T[1] and T[2], and node C is T3.

The declarative portion of the CSL program consists primarily of the Construct statement. The Construct statment defines the tasks, communication channels, shared variables, and task conditions. Local CSL variables and conditions used for sequencing may also be declared. File2 is the compiled code associated with T3. The use of the Range statement allows simple specification of the two channels, corresponding to the arcs in Figure 2-1.

The executable portion of the CSL program consists of a specification of the traversal of the computation graph. The Cobegin statement in this example starts off four parallel streams of control. Although it is assured that statements within a stream will be executed sequentially, it is not known what streams will finish first or what other statements will be in execution during the execution of a given statement, unless some form of synchronization is used. For instance, it is determined that each Send will occur after each Execute. It is not determined which Send will start first. Due to the synchronization effects of channel communication, however, it is determined that neither of the Receive statements may complete before its corresponding Send starts.

Additional semantics of the Cobegin are that it does not complete until the last stream contained within it completes. This is a form of implicit synchronization within the language.

CSL explicitly controls the traversal of the computation. The meaning of "Execute T3;" for instance, is: Send a message to task T3 telling it to commence execution. The statement "Execute T3;" does not complete until word is received that task T3 has completed. This is contrasted with dataflow [Jayaraman 80]. In a dataflow model T3 may commence execution as soon as it sees that both of its inputs are available. CSL, on the other hand, must explicitly start T3 after having received word that both the Receive statements were completed. Although explicit control can waste time and resources through synchronization delay and additional

message overhead, it has the advantages of more easily allowing dynamic computation structures (i.e. dynamic computation graphs), and providing better information for making an efficient mapping between algorithm and machine.

The previous example gives the basic flavor of CSL. Some of the CSL features not shown include, shared variables, task conditions, synchronization conditions, conditional branching (If, While, For-to), guarded commands, and dynamic control. Shared variables are discussed below. The others are discussed in detail in [Browne 81].

Shared variables are used in a With Do Execute statement. For instance, the statements "With A:B Do Execute T1;" allows T1 to begin execution as soon as the shared variables A and B can be reserved for it. All variables left of the colon are reserved for exclusive read-write access during the execution of the task. The variables to the right of colon are reserved for read only access and therefore may support multiple readers simultaneously. The Construct statement is used to declare which shared variables are obtainable by which tasks. Since CSL supports both shared variable and channel means of communication, the programmer may select the method or the combination of methods that would seem to fit the algorithm most easily.

The next section looks at the CSL System and shows the functions of the Run-time System (RTS) within it.

2.3 Overview of the CSL System and the RTS

The CSL System consists of five parts:

1. The CSL Translator. The CSL Translator takes a program written in CSL syntax and converts it to an intermediate form which includes Pascal data structures and code, and calls to the Run-time System.

2. The CSL Run-time System (RTS). The CSL Runtime System is the set of routines that interprets and controls the execution of a CSL program.

3. The Processor Resident Monitor. The Processor Resident Monitor is a local operating system which creates an execution environment for the task or tasks which are under its supervision. Its duties are simply to perform the requests sent to it by the RTS. Requests include the sending or receiving of task variables to or from channels and the transfer of control to the task in order for it to begin its execution.

4. The Job Resource Analyzer. The Job Resource Analyzer scans the CSL program and defines the resources necessary to implement a computation graph.

5. The System Loader. The System Loader executes the binding that establishes communication paths and communication structures between the CSL System and the local operating systems for each task environment.

The CSL Run-time System is at the hub of the wheel in that it must interact with each of the other parts. It can be said to define a local operating system for a given parallel program [Browne 84]. The Run-time System therefore maintains the state of the computation as part of its function.

The RTS consists of a set of tables and Pascal procedures. The tables maintain the state of each task and define the communication and synchronization information used in the traversal of the computation graph. The procedures execute on these tables to provide the interpretation of CSL statements, and the recording of the state information associated with their execution. About 80 percent of the procedures are machine independent. Twenty percent of the procedures form the low-level interface between the RTS and its tasks. They contain machine dependent system calls and can vary considerably from implementation to implementation.

The RTS uses six major tables:

1. The Task Table. This table contains information about each task defined in the construct statement. The key to this table is the task name together with its index. It contains the state of the task, and the names of all files associated with it. Files include the compiled file name, task input and output files, and monitor files that are used in debugging.

2. The Job Variable Table. This table contains the values of all variables local to the CSL program that are used in synchronization. They include integers, booleans, conditions, and any constants used in the program.

3. The Channel Table. This is a static table that defines the channels declared in the Construct statement. Information about channels includes the number of buffers allowed, the type of channel as declared by the programmer (either message or data), and a pointer to the Topology Table.

4. The Topology Table. This table is also a static table that shows the source and destination tasks for a given channel. (Two channels may have the same topology.)

5. The Shared Variable Table. This table lists the shared variables, which tasks are

permitted to access which variables (as defined by the programmer in the Construct statement), and the task currently attached to it.

6. The Statement Table. This table is really a linked list structure that represents the executable portion of the CSL program. For the most part, every record in the structure corresponds to a statement in the language. Each record is filled in by the CSL Translator in order to provide the RTS with enough information to perform the statement's function when the time comes for it to be executed.

A Statement Table for the example CSL program in the previous section is shown in Figure 2-2. The horizontal boxes represent the statements of the program with the vertical boxes marking the beginning of parallel streams. Statements are connected to each other with pointers that are followed as the program execution proceeds. The Control Block shown in Figure 2-2a has a pointer pointing to the current statement in execution. In Figure 2-2b, after the Cobegin is encountered, new Control blocks are created to point to various statements under execution. As each stream is finished, the Control Block associated with it is deleted by the RTS. When all Control Blocks are deleted, execution halts.

Since with each Cobegin new streams are started, the RTS must be able to divide its time fairly between the interpretation of each of the streams. This is solved by having control pass from one Control Block to another following the circular linked-list around the loop. As soon as the RTS realizes it has done a sufficient amount of work in a particular Control Block, it passes control to the next one on the linked-list.

The work done to manage the execution of the basic statements in CSL (like Execute, Send, Receive, With Do Execute) consists of three parts. First there is an "initialization" phase where the tables are accessed and a message is sent to the task in question. Second, there is the "polling" phase, where the only work done in regard to the statement is to check whether a reply has been received from the task concerning the accomplishment of that order. Third, there is a "finishing" phase where the reply is noted as received and the status of the tables are changed. This initialization, polling, and finishing sequence is used as the basic model of the RTS and the Parallel RTS (PARTS) in succeeding chapters.

Now that the basic functions of the RTS have been presented, we next evaluate the RTS in the context of a computation graph to uncover inherent parallelism.

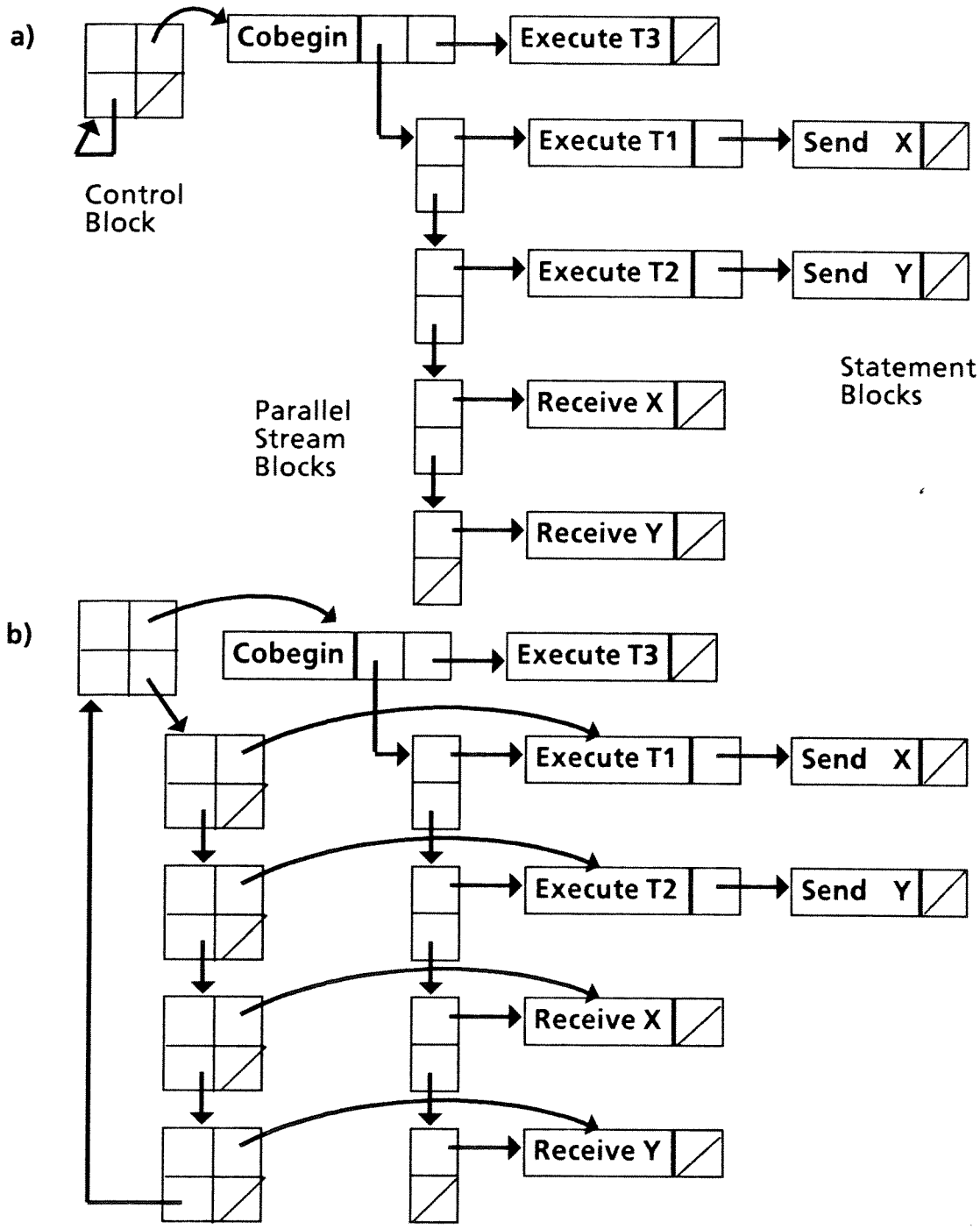

Figure 2-2: Statement Table for the Sample CSL Program

2.4 Development of the Computation Graph

At the highest level, the computation graph of the Run-time System is a single node. This is shown in Figure 2-3a. The inputs to this node are the Pascal tables and data structures created by the CSL Translator. The computation done at the node is an execution against these data structures that performs the management functions of the RTS. The output of the node can be thought of as nil here since the management functions of the RTS have nothing to do with the output of the program that it manages.

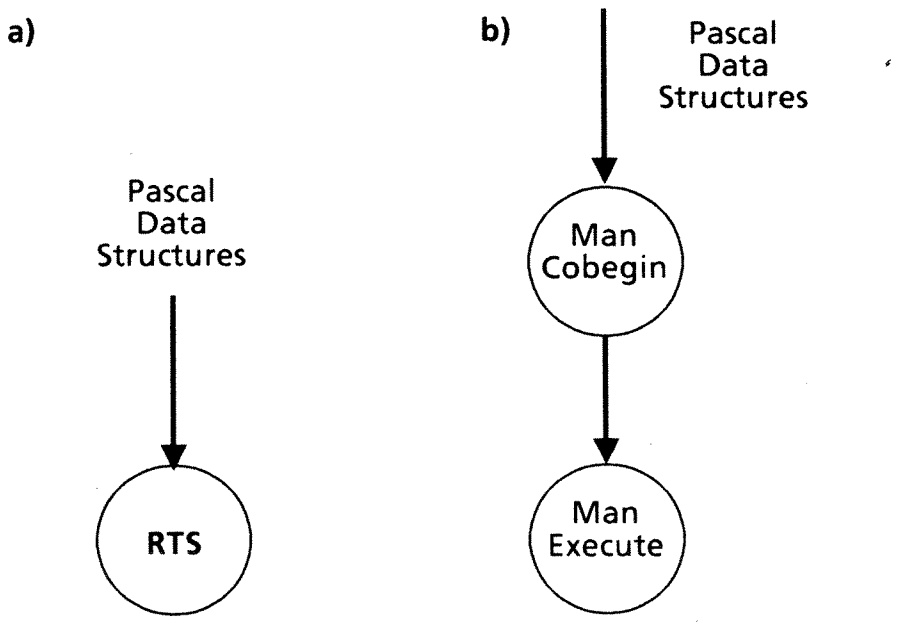

Figure 2-3: Two Computation Graphs

Figure 2-3b shows a simple computation graph for the management of the sample CSL program shown earlier. In this graph the unit of computation is the management done for each of the two logical statements Cobegin and Execute. Man Cobegin stands for the management of the Cobegin and the statements within it. Man Execute stands for the management of the Execute T3 statement. CSL requires that the execution of the Cobegin and the Execute T3 statements occur sequentially. It is therefore not surprising to see the management of these

statements performed sequentially as well. All the Pascal data structures can be thought of as being passed down the arc. This, in effect, gives the state of the system to Man Execute.

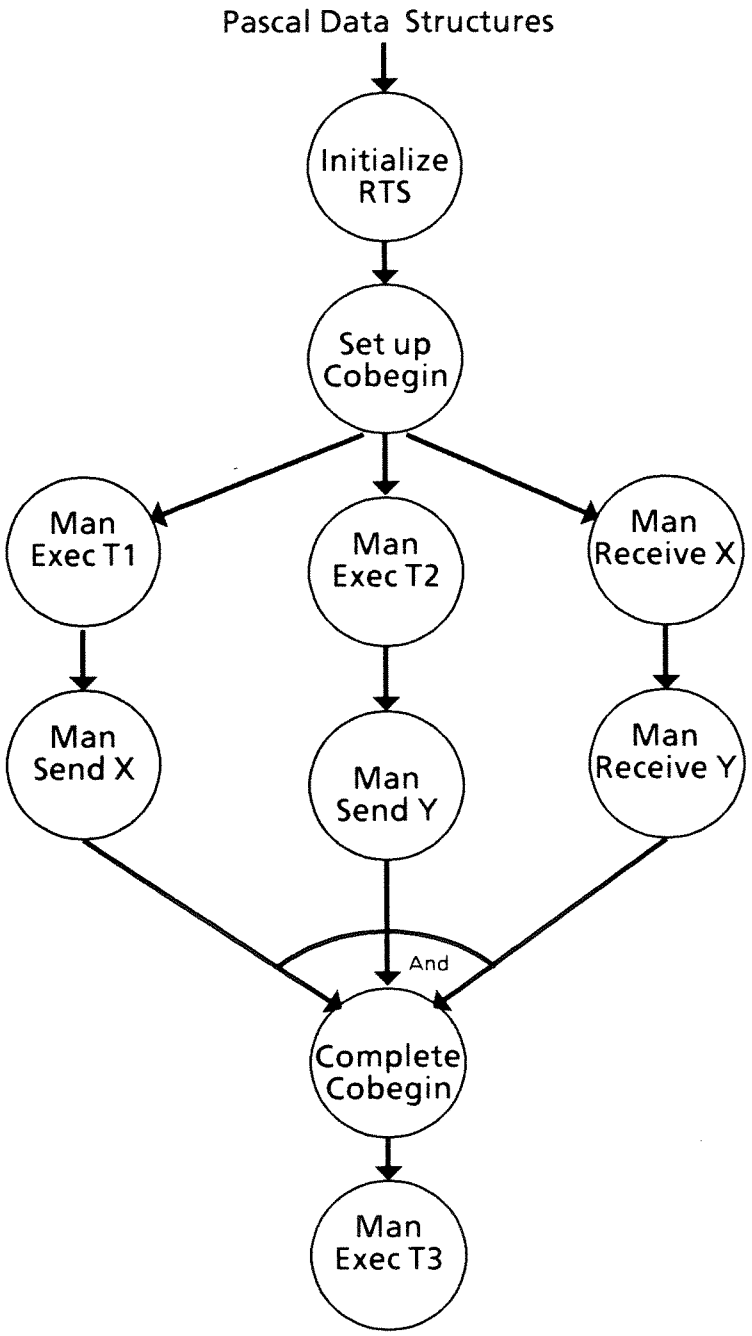

Figure 2-4: A More Detailed Computation Graph

A more detailed version is shown in Figure 2-4. The first job of the RTS in any program is to initialize data objects that it uses. Setup Cobegin does the job of creating new Control Blocks for the Cobegin. The two Man Executes and Man Sends within the Cobegin have no dependencies between each other. There is also no dependency between Man Execute T[1] and Man Send Y. Because of the serial execution of the Executes and the Sends, however, their managers are serialized. The Man Receive nodes are serialized, but the dependency here means that either may execute first as long as both don't execute together. This is due to the fact that T3 cannot be receiving from two channels at the same time. The Complete Cobegin node does some cleanup work after the streams are all completed. The inputs here must all be present before the node may begin.

What are the inputs to Complete Cobegin? There is no clear cut answer here. One could consider that the state of the system is partitioned and sent along appropriate arcs as needed and that the union of the three arcs is the system state. It might be easier, however, to consider the state of the system as residing at every node where an update at one node causes a modification of the same data in all the other nodes. (Dependencies would prevent any inconsistencies due to different versions of state information.) Dependencies would then be simply control signals.

Figure 2-4 still does not begin to show all the parallelism in the management of that CSL program. In Figure 2-4 the Cobegin node was exploded to form a number of new nodes. Any of the nodes in Figure 2-4 is now a candidate for a new explosion. The operations within each ManExecute statement, for instance, can be broken down into a new computation graph with smaller units of computation and a more complex dependency structure.

There are ways to find parallelism other that exploding nodes. The two nodes ManExecute T[1] and ManSend X are a case in point. It is helpful here to ask the question: exactly what causes the dependency between these two nodes. An analysis of ManSend X may show that the cause for dependency between it and its predecessor occurs after some initial computation done within ManSend X. This initial computation, having no dependency with the ManExecute node, may begin immediately. This is in fact the structure of the management

functions in the RTS. Part of the node ManSend X could begin immediately, even though the actual data transfer could not.

The fact that this parallelism exists, however, is not a sufficient condition for exploiting it. After analyzing computation graphs for the RTS down to the next level of detail, it was found that the units of computation without dependencies tend to be very small in size. Their exploitation seems to require unrealistic communication speeds in order to keep the communication/computation ratio down to a reasonable level. It would also require a great deal of effort for only modest improvements. It was therefore decided to work with exploiting the larger grained parallelism using computation graphs of the level of detail of Figure 2-4. This result is not surprising, since if CSL programs are targeted for architectures that support a large grain of parallelism, it would follow that the resource management algorithms running on the same architecture would be subject to the same constraints.

In order to exploit parallelism in the management of more than just this single example of a CSL program, generalizations need to be made about the structure of the computation graphs that result from other CSL programs. The problem with developing a generalized computation graph for the RTS is that the functions of the RTS are data dependent. The CSL program (which is data to the RTS) is the driving force behind the structure of the RTS's computation graph. CSL though, is a very new language that has had only a small amount of real use. It cannot be stated dogmatically what structure CSL programs will take and what features will be most commonly used.

In trying to gain some insight into this problem the author has studied the structure of a number of different CSL programs (including many that have not actually executed) and the computation graphs of the RTS that result from these programs. The following observations have been made:

1. The structure of the graphs can be misleading when conditional branches are used. Using an IF statement, for instance, whole sections of the graph may not be executed due to the evaluation of a conditional expression.

2. The structure of the graphs may be impossible to completely write before run-time since CSL has dynamic features. This includes executing a variable number of tasks

(depending on the value of a variable at run-time) or the execution of a task a variable number of times (such as a Repeat-Until conditional.)

3. The only large scale parallelism within the management functions seems to be in the Cobegin statements. (This is not discouraging since all CSL programs use Cobegin extensively.) This parallelism has the further advantage that it is explicit and definite.

4. The most often used CSL executable statements are Execute, With Execute (which allows shared memory), Cobegin, Send, and Receive.

The conclusion of these observations points to using the Cobegin as the basis for exploiting parallelism. The efficiency or desirability of such a notion, however, has yet to be proven. For though we have shown inherent parallelism exists within the RTS, we have not shown how much benefit can be derived from it, or how well it can be achieved.

The next chapter, then, shows the desirability of the Parallel RTS (PARTS) and details its design.

Chapter 3

Parallel Structuring of PARTS

3.1 The Reason for PARTS

It has been shown that parallelism exists in the functions of the CSL Run-time System. The development of a Parallel Run-time System (PARTS) is also dependent, however, on whether or not a reasonable speed-up is attainable. Parallel structuring of the Run-time System adds complexity and may use additional resources. It should be attempted only if the speed-up gained outweighs the extra resources and complexity, or if a characteristic other than speed-up is desired such as flexibility or reliability. Our main concern here, however, is speed-up.

An example illustrates the benefits that PARTS could provide. Consider a single RTS managing the execution of 10 parallel streams within a Cobegin, each containing the execution of a single task. Consider also that each Execute must be "initialized, polled, and finished." Assume that each task executes for 100 units and that the polling and finishing work takes zero time.

```
Cobegin
   // Execute T1;
   // Execute T2;
        .
        .
        .
   // Execute T10;
Coend;
```

If the initialization of each stream takes zero time, the Cobegin can be completed in 100 units since each stream starts off effectively at the same time. The experience of a running version of the RTS on the CDC Dual Cyber 170/750 has shown, however, that the initialization

time comprises 16 percent of task execution time for average length tasks. In this example, with 16 units of initialization time for each of the 100 unit length tasks, the last stream (and therefore the entire Cobegin) does not finish until 260 time units have elapsed.

If two RTS's could be used together somehow to manage the streams, giving five streams to each, the effective time could be reduced to 180 time units. This is a speed-up of 1.44 over the 260 time units. Of course, an extra processor was used, bringing the total to from 11 to 12. This increase of 1.09 still seems well worth the price paid. Although this example is simplistic, it suffices to show that parallelism in the RTS has potential for fairly good speed-ups of total Cobegin completion time. Chapter 6 takes a closer look at performance issues.

3.2 Overview of PARTS

PARTS is a parallel structuring of the CSL Run-time System that allows a number of tasks to simultaneously manage non-overlapping groups of statements within Cobegins. Each new RTS within PARTS is responsible for a subset of the total number of streams started by a Cobegin. Tables are replicated between RTS's where possible and shared between them when necessary. A synchronization mechanism is introduced to allow exclusive access/updates of the shared tables. The basic format of the RTS's remains unchanged, although minor modifications have been made as they relate to the access of system tables.

Using computation graphs, the last chapter indicated that parallelism could be exploited in Cobegin statements. PARTS exploits it in the following way. Since the management of statements within a stream tends to take a sequential form, it seems logical to make a single stream the smallest unit of work that a task may manage. A task then, may manage a single stream or manage any subset of streams within a Cobegin.

The set of tasks used to collectively manage all the streams within a Cobegin are called RTS's within PARTS. These new RTS's are very similar to the original RTS. Both types can manage any CSL statement. Both types go about managing in the "initialization, polling, finishing" format. The only difference is in the way the new RTS's access their tables. In

PARTS, different RTS's may require the same global computation state information. Thus, some system tables are shared across all of the RTS's. The design of this table partitioning is shown in the next section.

3.3 Partitioning of System Tables

The management of any CSL statement requires access to one or more of the system tables listed in Chapter 2. The management of an Execute statement requires access to the Task Table and often the Job Variable Table as well. The management of a Send requires access to the Channel Table, Topology Table, Task Table, and often the Job Variable Table. These accesses are usually very short, requiring just the update of a single field. Any update changes the state of the of the computation. The management functions in turn, are synchronized through system state information. This is explained in an example.

In the Sample CSL Program in Chapter 2 the Receive statements are located in different streams. They could therefore be managed by different RTS's. In Figure 2-4 the Man Receive nodes are able to execute in any order except simultaneously. The point here is that it is an update of a system table that causes synchronization between the two nodes. As soon as one RTS has "marked" the task T3 as under its control, the other RTS must wait until the task has been released. The system tables, then, can form a dependency relation for synchronization within the RTS's.

In PARTS, it has been decided to make synchronization information in system tables available through shared memory rather than channel communication methods. The reasons are the following. First, the nature of table accesses within the RTS's tend to be very frequent, but short. Second, one of the features that CSL supports is shared memory, therefore it would be likely that such a device would be available for use. Third, the dependencies caused by system tables would not have to be explicitly mapped out between the new RTS's beforehand. Fourth, the nondeterministic features within CSL could be easily preserved. Fifth, the RTS's would not have to be aware of each other's existence, simplifying the design. (This advantage is also cited by Gottlieb in the design of the shared memory based Ultracomputer [Gottlieb 83].)

The design of PARTS, then, assumes that those processors that are used for the execution of RTS's have access to a common shared memory. It is not a requirement, however, that shared memory be used for communication between tasks in the rest of the CSL system. An example of a circumstance where this can be realized is in the TRAC machine [Sejnowski 80].

In PARTS, those tables that contain state information are made available to the RTS's using shared memory. Any other tables are replicated and made available in the local data space of each RTS. The following gives an explanation of the layout and location of each of the six system tables.

1. The Task Table. This is the most actively used table since it gives information about whether a task is ready to have a new operation performed to/from/by it. It is included completely within shared memory. It is the same as the old task table except that it has a new field called "IN-USE" that tells whether or not this record has been "marked" for use by another task.

2. The Job Variable Table. This table is partitioned into two tables: the Local JVTable and the Shared JVTable.

   The Local JVTable is static. It is in the local data space of each RTS and is very similar to the old Job Variable Table. It contains the actual values of any range variables and constants since they may be replicated for each RTS. The values of other variables are not included. In their place, however, is the appropriate index of that variable in the Shared JVTable. Two fields are added: one called "NONLOCAL" which tells whether the variable is nonlocal, and "INDEX" which gives the index into the Shared JVTable if NONLOCAL is true. The Local JVTable is identical for each RTS.

   The Shared JVTable contains the volatile portion of the old Job Variable Table and has the very same format. There is only one copy of it, and it contains integers, booleans and condition variables.

3. The Channel Table. This table contains static information about the channels and may therefore be replicated for each RTS.

4. The Topology Table. This table contains static information about the routing of data on channels. It is replicated once for each RTS.

5. The Shared Variable Table. This table reserves shared variables for all tasks. It is therefore located in the shared memory area.

6. The Statement Table. This table defines the executable code of the CSL program. It is partitioned into sections that reside in various RTS's. The RTS's then execute on their pieces of the statement table in parallel.

Another issue related to the partitioning of tables is the granularity of the unit of locking within the shared tables. In PARTS, for the time being, the granularity unit is the table itself. This will let more than one RTS access different tables at the same time, but mutually exclude on the same table. There are two reasons for this decision. First, it reduces in complexity of the initial design. Second, table accesses tend to be quick; when they are not quick, it is because the whole table must be searched. This means that each unit of locking in the table would have to be acquired during the search, increasing the search time somewhat.

Although the table being the unit of locking granularity makes the design simpler, the extra complexity would be worth the effort if it were shown to substantially increase table access throughput. This in fact would begin to take place as the number of active RTS's reached about ten (depending on the characteristics of the workload). The simulation results of Chapter 6 confirm table access time as being a sensitive area to the performance of PARTS.

3.4 Synchronization in System Tables

Another facet of the design of PARTS is the synchronization mechanism that allows multiple RTS's to mutually exclude their access to the shared system tables.

As explained in the last section, all static system table information has been replicated for access by each of the individual RTS's. This includes the Channel Table, Topology Table, Statement Table, and the Local JVTable. The Task Table, Shared Variable Table, and Shared JVTable, however, are located in shared memory. Access to each of these tables must be mutually exclusive for each of the RTS's.

This can be accomplished using a very straightforward mechanism. PARTS assumes a Test-and-Set instruction to implement the Barbershop Algorithm [Reed 79] to provide a monitor each table. The effect of the algorithm is the same as a group of persons entering a barber shop, taking a number off the rack, and waiting until their name is called. This algorithm allows the RTS's to manage their own table access using two semaphores. The semaphores correspond to the Ticket (or number) and the Server (or barber). The Acquire and Release routines are shown below.

```
(* T controls access to the Ticket *)
(* S controls access to the Server *)

Initially:

T,S := False;  Ticket := 0;  Server := 1;

Procedure Acquire_TT; (* Acquire Task Table *)
  Var
    Temp : Integer;

  Begin
    While Test-and-Set(T) Do (* nothing *);
    Temp := Ticket;
    Ticket := Ticket + 1;
    T := False;
    While Temp <> Server Do ;
  End;

Procedure Release_TT;

  Begin
    While Test-and-Set(S) Do ;
    Server := Server + 1;
    S := False;
  End;
```

Since PARTS allows access to each of three system tables, a total of six semaphores and six integers are needed in shared memory. The granting of the Ticket is probabilistic, but once the Ticket is obtained access is guaranteed. Each RTS waits in a logical queue for the RTS ahead of it to signal that its turn has come.

3.5 RTS Algorithm Modifications in PARTS

Except for the access requred to the shared tables, the algorithms could be left virtually untouched. Each access to a shared table would take the form:

```
Acquire_Tx;
(* Reference of Shared Table Tx *)
Release_Tx;
```

In order to prevent deadlock the following rule is enforced: each table must be released

before another is acquired. This however, does not prevent the same table from being accessed twice inside the critical section. This is allowed in two types of situations. One is when the Task Table or Shared Variable Table must be searched for a given record. Then the entire loop is placed inside the Acquire and Release. The other case is when several sequential accesses to the Shared JVTable are made in succession. This occurs in several of the routines.

The routines EXPRCOND and EXPRARITH which evaluate expressions use the Local JVTable and Shared JVTable extensively. They must be modified in much the same way. Since, when these routines are called, it is not known whether variables or constants are being accessed, the following algorithm is used in reference to the tables:

```
Check the type of the variable in the Local JVTable.
If type is other than 'operator'
   then
      If NONLOCAL = true
        then
          Acquire_SJV;
          Access the Shared JVTable using the
             Index stored in the Local JVTable.
          Release_SJV;
        else
          Access the value using the Local JVTable.
    else
      Access the operator using the Local JVTable.
```

This completes the modifications needed to the original RTS in order for it to be able to run in the new environment. The author, however, has modified the RTS further from the standpoint of efficiency.

The routines that manage the execution of the statements Send, Receive, and Execute all have something in common: they use the Task Table extensively during the 'polling' phase of the management. Since these statements are used extensively in CSL, expecially in Cobegins, a serious throughput problem could result from the constant acquiring and releasing of the Task Table by multiple RTS's as they manage these statements. To make matters worse, the Task Table often must be searched to find the particular task in question.

For this reason the RTS's within PARTS have been modified to access the Task Table only during the 'initialization' and 'finishing' phases of management. Each of these three routines now include a special initialization and finishing section. The Execute routine, for instance, was expanded to include an Init_Execute and a Finish_Execute routine which handle any needed accesses of the Task Table.

The basic idea of Init_Execute is to locate the task record and store it in a local area of the RTS for use during the polling phase. This is a safe operation since no change to the task record may occur while the task is execution. When the task has completed, Finish_Execute returns the record to the Task Table. This in effect gives two accesses of the Task Table, when there would have possibly been hundreds. The algorithms for the Execute routines are shown below. Two new fields are added to the Execute statement block record. They are Location and LM_Index. Location is either TT (Task Table) or LM (Local Memory). The LM_Index is the index of the Task Table record entry if Location = LM.

Execute

```
If the task index <> -2 then check the Location entry.
If Location = LM find the record using LM_Index.
If Location = TT then call Init_Execute.

If the task index = -2 search all the local records
   for the task record.  If not found Init_Execute.

Perform the standard cases in the old Execute version.
( When case of LF:  call Finish_Execute. )
```

Init_Execute

```
Acquire_TT;
Search the table for the desired task.
If IN-USE = true then Release the Task Table
   and exit.
If IN-USE = false copy the record into the
   first available space in the local area.
Set IN-USE to true for the Task Table copy.
Release_TT;
Set Location to LM.
Set LM_Index to the new record position.
```

Finish_Execute

```
Acquire_TT;
Search for the desired task.
Copy the record from the local area into the
  position found.
Release_TT;
Set Location to TT.
```

The algorithms above work because each RTS is able to communicate with any task directly in order to determine if it is finished. If instead, a completed task modified the Task Table directly, then every RTS would have to access it in the polling phase and bottleneck would result. If interrupts were used, the completed task should modify the local record space, not the Task Table.

The Send and Receive management routines work similarly. First the channel is found in the Channel Table. Then the sending (or receiving) task is found using the Topology Table. What follows, however, is the same sequence of actions used in the Execute routine. The task is 'initialized, polled, and finished' in the same way as in the Execute manager. The only difference here is that the *message* sent to the task tells it to do a send or receive rather than an execute. Thus the same local record storage area and Location and LM-Index fields can be used for all three routines.

Chapter 4

Previous Work

The design of PARTS can be thought of as a solution to parallel structuring of resource management in a specific domain. The CSL language and the functions performed by its run-time system have been well defined. Therefore, the solution given is one based as much as possible on knowledge about the original specifications.

For instance, an accurate determination of how often the RTS's within PARTS would need to communicate could be made because of knowledge about how a single RTS accesses its tables. This led to the use of a shared memory model with each system table being the basic unit of granularity. Similarly, the manner in which the parallel RTS's would be spawned and the mechanism for their control need not be extremely flexible since their functional characteristics are known in advance. The communication and control features, then, are not providing general solutions to parallel control, but rather specific solutions based on this system's requirements.

Although PARTS is a specific solution, some of its features are based on other solutions to very general and sometimes very specific problems. Rather than trying to define all relevant work, we restrict ourselves to work which is basic to shared memory implementations and work which deals with parallel resource management for parallel systems. The following areas of concern are included.

4.1 Synchronization

Basic mechanisms for achieving mutual exclusion within multiprogramming operating systems were introduced by Dijkstra [Dijkstra 65]. Later Dijkstra [Dijkstra 74] and Lamport [Lamport 74] studied similar issues for parallel processing.

The synchronization primitives in this thesis are used to implement an Acquire and Release of shared memory by the RTS's. These procedures then form the basis of a monitor [Hoare 74] over the table resources. The basic primitive Test-and-set is used to implement the Barbershop Algorithm presented in [Reed 79]. This algorithm solves the problem of granting exclusive access to a shared memory in a multiprocessing environment. Requests are always handled in a FIFO fashion. The use of Test-and-set in this thesis as a basic primitive makes the solution viable in a parallel processing environment.

An extension to the Test-and-set instruction called the Replace-add is discussed in [Gottlieb 81]. The Replace-add is a generalized form of the Test-and-set instruction. The Replace-add instruction can be implemented in such a way that multiple accesses to the same memory location by many independent processors can take place as quickly as a single access by a single processor. For instance, the readers-writers problem can be implemented without recourse to any critical section code. Although the cost of a basic memory access would be expensive in this system, it could well be worthwhile as a synchronization tool a future version of PARTS.

4.2 Granularity

The granularity of a parallel computation refers to the size of the elements that make up the computation. Granularity may be broken down into two basic types -- entity and event granularity [Mohan 85]. Entity granularity refers to the size of structural features within a computation such as the number of tasks or the size of the smallest lockable data element. Event granularity refers to the time between certain necessary operations such as communication or synchronization.

In PARTS, the issue of granularity is most clearly seen as entity granularity. The

decision to provide separate, lockable access to each of the system tables, but not to individually lock entries within those tables comes as a result of studying the length and types of accesses that the RTS's would need to be making. What is garnered from the literature in this area is that the choice of granularity can make very large differences in system performance and that the "right" granularity for a given situation may not be as obvious as it seems.

4.3 Control Issues

Process creation plays an important role in PARTS. The ability to spawn off new control processes quickly is critical to the design of this system. In [Barak 82] a suggestion for a type of process creation is presented. Barak proposes that processes could be dynamically created by the activation of a dynamic data type called a prof. A prof is a frame of a sequential process. In PARTS, this idea is used, but knowledge about the system allows this to be done in a static fashion. This is discussed further in the next chapter.

In a message based system, the need to consider communication times in the assigning of processes to processors is discussed in [Williams 83]. As communication times increase, the advantage of reassigning processes decreases dramatically. This is of major concern in PARTS and is the main reason that the different copies of the RTS's are assigned statically to their processors.

The form of control within the RTS's in PARTS is very similar to that in [Tilborg 80]. In that system, like in PARTS, each control process is responsible for a set of tasks (although that set may vary with time.) In [Tilborg 80] and in PARTS the control processes have equal weight in the acquiring of resources, however, in PARTS there exists a master-slave relationship between the RTS members. A single RTS is responsible for starting some set of slave RTS's (which may in turn start others.) The master RTS is then responsible for recognizing the completion of its slave RTS's as well as managing any other work assigned to it.

The load sharing policy in PARTS can be recognized as belonging to the server initiative class as shown in [Wang 85]. This paper taxonomizes load sharing systems and provides

performance results about them. In PARTS, the servers (processors) are partitioned into groups and each group serves one source. Although this system tends to waste computing resources, this is not seen as prohibitive given current trends in the cost of processors and memory. In [Shen 85] an optimal load sharing assignment algorithm is presented. This paper shares a common purpose with PARTS in that the purpose of load sharing should be to minimize completion time.

Chapter 5

Scheduling Issues in PARTS

5.1 The Need for Scheduling

Chapter 3 detailed the design of the individual RTS's within PARTS. It did not however, discuss how or when these RTS's should be generated. This is the subject in this Chapter.

Scheduling in any system can be thought of as the interrelation of two issues -- policy and mechanism. Scheduling policy is the goal desired from the system. Scheduling mechanism is the means used to try to achieve those goals. For instance in a traditional multiprogramming batch system, the policy might be to achieve minimum average turnaround time in all the jobs. The mechanism chosen to achieve that goal might therefore be to implement a shortest job first ordering on the job set.

A mechanism is chosen because it operates on parameters that are controllable and useful in achieving the policy. The "shortest length object code" parameter as an approximation to the shortest execution time of a job may not give the fastest turnaround time, but in the absense of complete information, it may still be the most useful parameter available. In fact, it is often hard to extablish the connection between mechanism and policy. But it would be expected, that the more information available, the better the parameters will be and the better the mechanism will be in implementing policy.

In most systems, the policy involves the minimization and the maximization of more than a single parameter. It may be desired, for instance, to minimize job turnaround, maximize throughput, and minimize costs. Since there probably will be no point in the system's operation-

space that simultaneously achieves these goals, an analysis of the relative importance of each must be weighed carefully.

In the PARTS system, the policy of scheduling consists primarily of a single goal -- minimizing the execution time of a CSL program. The mechanism for achieving this goal is the creation of new RTS's. The policy is subject to the constraints of the current configurations of the Run-time System. The mechanism is established from a set of derived parameters. The parameters are in many ways rules-of-thumb.

This is perhaps a broader view of scheduling than some might perceive. The scheduling mechanism in most systems usually assumes a fixed number of jobs that must be executed. The mechanism then maps jobs to resources in some fashion. With PARTS, the number of jobs may vary. Scheduling therefore becomes the creation and mapping of RTS's on the machine in such a way as to minimize the total execution time of the CSL program.

It is not claimed that the scheduling decisions made are the best decisions in all cases. The lack of completely specified information about the system assures this! What is hoped, however, is that the scheduler uses to full advantage whatever information might be available. This information takes two forms: knowledge about the system, and knowledge about the load. The system here is the environment that PARTS is executing in. System factors are not affected by the CSL program. The load is the structure and characteristics of the CSL program that the user has designed. The load would be expected to vary from program to program. The next section discusses these two types of factors in more detail.

5.2 Factors of Influence

Before discussing the basic attributes of scheduling method in PARTS, it should be helpful to categorize in an intuitive fashion the factors that will help and hurt performance. (Performance is defined as the speed-up of the total computation due to the parallel structuring of PARTS.) In other words, given a certain number of RTS's working in parallel on a CSL program, what factors cause better speed-up to be obtained, and what cause worse speed-up. Many have already been hinted at in various places, but are listed again for completeness.

The factors can be separated in terms of system factors and load factors as explained above. They are listed in the manner that causes better speed-up in PARTS. Obviously their contra-positives will cause worse performance.

System factors:

- Small RTS creation time. It should be clear that the smaller the time it takes to create the new RTS's, the better the performance. This includes the time it takes to partition the system tables for each RTS, and decide what streams each RTS will manage.

- Small RTS invocation time. Once the RTS has been created, this is the time it takes to tell the RTS to begin execution.

- Low frequency of communication between RTS's. This causes less synchronization delay in PARTS from access to shared memory.

- Small volume of communication. In PARTS, this means that critical sections within the acquiring and releasing of tables are executed quickly.

- Large task initialization time as compared to task execution time (I/E). Although it seems counter-productive to desire the RTS's to take a long time initializing a task, if this time has been reduced as much as possible, and if it is large relative to the execution time of the task, then more potential speed-up exists. It is easy to see in the CSL example in Chapter 2 that if this ratio is small, then the speed-up of the initialization relative to the task execution is insignificant.

- Large task finishing time as compared to task execution time (F/E). This works the same way as initialization time. Both initialization and finishing are system factors since they do not vary from task to task.

Load factors:

- Large number of streams. The more streams present, the more work to be done, and the greater the number of RTS's that can be created to parallelize that work.

- Small task execution time. If the tasks are small, then more of the total work becomes the initializing and finishing of the task. This results in greater speed-up potential. This is really a restatement of the preceeding system factor "large task initialization time."

- Small average task-to-task communication time. This works the same way as a small task execution time. Since the statements Send and Receive also involve the initializing and finishing of the tasks associated with the communication, a small communication time results in greater speed-up potential. It is different from small task execution time, however, in that small communication time is always desired, while small execution time is usually not, due to the overhead of starting a task.

- Large frequency of communication. Communication between streams causes streams to synchronize at their respective Send and Receive statements. This in turn causes the RTS's to experience sudden bursts of management work. It should be easy to see that more potential parallelism exists here than if the RTS was given small pieces of work in a scattered fashion.

- Uniform task execution times. This is best understood by a counter-example. Consider 10 tasks executing in parallel. Nine have execution times of 100 MS. One has an execution time of 5000 MS. (Assume the task initialization and finishing times are 10 MS each). The 5000 MS task so dominates the total amount of time to complete the Cobegin, that it does not matter how many RTS's are used. The RTS containing the 5000 MS task will complete last (approximately 5020 MS after it started).

- Little use of dynamic features in CSL. CSL allows features such as the execution of a range of tasks, where the range is not defined until run-time. Other instances of the dynamic range statement in CSL can involve channel communication and shared memory usage. In the best case, this causes extra work for the scheduler. In the worst case it can severly affect performance.

The scheduling strategy used in PARTS should, then, take note of the preceeding factors wherever possible. The following section outlines the basic strategy of the scheduling method.

5.3 Basic Strategy -- How to Create RTS's

The first decision to be made is whether to create RTS's statically or dynamically. Three basic strategies are discussed -- two dynamic and one static. In the analysis below, the author has used real system data on the CDC Dual Cybers wherever possible. This gives credence to the claims made about the chosen scheduling method on the Cyber, but means also that the decisions made here may not be the best on other systems.

The first idea is for RTS's to be dynamically created at the start of each Cobegin.

One advantage of dynamic scheduling in PARTS would be the delay of the creation of new RTS's until it is relatively certain they are needed. For instance, a Cobegin could be stripped apart at run-time, with streams being given to new RTS's. This would make it easier to assure that there are, in fact, enough streams needing management to justify the expenditure of resources to manage them. Also, the dynamic features of CSL such as the execution of a range of

tasks, could be handled with relative ease since the range would be known at the time the RTS's are created. It could also be possible for one RTS that has finished its work to come to the "rescue" of an RTS that is currently swamped with a large work-load.

The CDC implementation of CSL shows process creation statistics that, unfortunately, make the dynamic creation method too expensive. Aside from any compiling that might take place, the loading of a process and its preparation for communication with another RTS takes 400 MS. This is longer than the average time it taskes a task to execute! Since RTS creation time cuts directly into the "profits" derived from parallelism, this cost is absolutely prohibitive.

The second idea is to create RTS's that dynamically receive streams as their input data.

This method retains most of the advantages of the basic dynamic creation method. The difference is that RTS's would not be destroyed after each use, but would rather remain in a waiting mode. This of course, means greater resource overhead. Some sort of master task would scan down the streams of a Cobegin and distribute them to the slave RTS's. The most straightforward way to do this would be to include the Statement Block Table in shared memory. The pointers to the heads of the streams could be passed to each RTS in turn. Since these statement blocks are the most frequently referenced pieces of data in the CSL System, it would be necessary to have each RTS copy all its streams into its own local area.

In this approach, the dynamic ranges could be handled easily since their values would be known at RTS invocation time. Also the start-up overhead could be drastically reduced as compared to the first method.

Although this idea could probably be the method of choice in a number of systems, it is not chosen here. First, it is not clear that the dynamic features in CSL will be used often. Even so, it does not retain a large advantage over the way the static method deals with this situation. Second, this method performs two extra phases of work that can be eliminated in the static case. One is the run-time analysis of streams to determine properties that might influence what RTS it is sent to. (For instance, how many statements are included, what type are they, how large are the tasks to be executed). The static method performs this once-for-all at compile time. The

other phase of work is the copying of streams to the local areas of each RTS. Even if shared memory contention could be eliminated, this would still directly deteriorate performance. Third, this method incurs a great deal of complexity that can be eliminated in the static method.

The third idea is a static scheduling method. The RTS's are created at compile time with the statements they will execute. They are loaded with the rest of the tasks and are ready to begin as soon as a "Go" signal is received.

The main advantage here is that the RTS can be started as easily as a task can. The high cost of communication on the Cyber is reduced to the bare minimum, the set of system calls comprising a single RTS-to-task communication -- 4 MS. With this lower communication overhead, PARTS could afford to have more RTS's working in parallel, reducing Cobegin completion time over that of the other two cases (except, perhaps, when dynamic ranges are used.)

Another significant advantage with this method is that the RTS's could be initiated exactly as if they were ordinary tasks to be executed. There would be no need for a special process to analyze the Cobegin's streams and fork off RTS's. The RTS which runs the sequential portion of the CSL program could simply "Execute" the other RTS's at the beginning of its own Cobegin.

There are two main disadvantages of this approach. One is that the analysis of the CSL program at compile-time may not generate as many useful scheduling parameters as may be generated at run-time. The other is that the static approach will waste resources if dynamic features of CSL are used. An example concerning the use of dynamic features follows.

5.4 Details of the Static Method

The first issue needing attention is how the static method handles the dynamic features in the CSL Language. An example of dynamic ranges used for indexing follows.

```
Job  Dynamic_Range_Example;

Var A,B : Integer;

Construct
    Tasks   T[i]: File1;  Range I := 1 to 10;

Begin
  (* Calculations involving A and B *)
  Cobegin
    // Execute T[i]  Range I := A to B;
  Coend;
End.
```

If A and B are only determinable at run-time, the static scheduler assumes that all 10 tasks could be executed. If, say, two RTS's were decided upon, their executable code would look like the following:

```
(* RTS1 *)

Begin
  (* Calculations involving A and B *)
  Cobegin
     // Execute RTS2;
     // Execute T[i];  DRange I := A to B [1..5];
  Coend;
End.

(* RTS2 *)

Begin
  Cobegin
     // Execute T[i];  DRange i := A to B [6..10];
  Coend;
End.
```

A new type of Range statement is added called DRange or Dynamic Range. Each RTS checks the values of A and B and then executes those tasks that lay in the intersection of the indices A to B and those of [x..y] that are assigned to it. (The general form of [x..y] includes a step value as well).

The disadvantage with this method is that a RTS may have been created needlessly if the range turns out to be from only 3 to 5. Ameliorating this disadvantage are two factors. First, RTS2 may not, in fact, have been created needlessly, since it may be invoked in the next Cobegin. (The static scheduler can allow RTS's to be reused). Second, since the start-up overhead of RTS2 is very small, it should not noticeably affect performance. Here, RTS2 is the equivalent of a task that does almost nothing and will complete before any of T[3], T[4], or T[5]. The only time lost is the 4 MS RTS2 initialization time.

The implementation of this DRange feature is very simple. Three new fields are added to the Range Data Blocks that are pointed to by the Parallel Stream Blocks. They are the Dstart, Dstop, and Dstep values of the [x..y by z] range assigned to it by the scheduler. When an RTS manages its Cobegin, it checks to see if the Dstart value is -1. If so it continues as normal. If the Dstart value is not -1, it compares the range A to B with [x..y by z] and only creates the streams that are at the intersection of the two ranges.

The next issues concerns RTS creation. As was mentioned before, each RTS is created at the beginning of the run. It is assumed that it has top priority on the processor to which it is assigned, and remains resident over the course of the run. Since it would be wasteful for each RTS to be used only once, provision has been made to re-use the RTS's whenever possible. This is accomplished through multiple entry points.

Each RTS has one entry point for each possible invocation. The code at the entry point sets a pointer to the beginning of the statements to be executed for this particular Cobegin. It then branches the RTS into its main loop. The master RTS that "Executes" this slave RTS must specify the appropriate entry point. Example:

```
(* RTS1 *)

Cobegin
   // Execute RTS2.Entry2;
   // Execute RTS3.Entry1;
   // (* other statements *)
Coend;
```

The entry points signify the second use of RTS2, and the first use of RTS3. The use of

entry points in RTS's does, unfortunately, make the RTS's slightly different from ordinary tasks. However, all tasks have implicit entry points anyway, so this addition will not require any additional system features, even though it tarnishes complete uniformity between task and RTS invocation methods. This addition seems a small price to pay for the additional flexibility offered in the system.

The next issue of concern is the placement of parallel streams that "execute" other RTS's within the Cobegin. Although each stream is evaluated nondeterministically, the serial RTS has the custom of starting them off in the order they are written. This design feature can be used in PARTS to great advantage. The PARTS Scheduler will assign any forking of RTS's to the beginning of each Cobegin. This gives the minimum delay to the critical path.

Another issue to be dealt with in the basic mechanics of scheduling concerns nested Cobegins. In the example above, it is entirely possible that a statement managed by RTS2 could be another Cobegin! This is handled, again, very easily in the static scheduling method. Each Cobegin is analyzed separately at compile time to determine whether it warrants parallelism in its management. RTS's may then invoke and manage each other.

The set of RTS's executing in a CSL program with nested Cobegins then becomes a tree-shaped structure of master-slave relationships. It is a method of adaptably adding parallelism to the system that invokes RTS's only when necessary. The tree will have a height less than or equal to the level of nesting of Cobegins in the program (depending on whether or not all the Cobegins warrant parallelism in their management). Each parent RTS is responsible for the invocation and management of its child RTS as well as any other normal CSL statements assigned to it. For instance, RTS2 in the above example could be structured as:

```
(* RTS2 *)

   Cobegin
      // (* statements *)
      // (* statements *)
      // Cobegin
            // Execute RTS4.Entry1;
            // (* statements *)
            // (* statements *)
         Coend;
```

```
  // (* statements *)
Coend;
```

In this case RTS2 cannot finish until it has received word from RTS4. The act of having each RTS responsible for its own set of slave RTS's has several advantages. First, the management of the RTS's may be spread more evenly over the system. Second, the Cobegin associated with RTS4 may be executed based on a condition. If the condition proves false, then RTS4 is not started needlessly.

5.5 Basic Strategy -- When to Create RTS's

The last section outlined the approach for creation, invocation, arrangement of streams, and management of dynamic features within the RTS's after it was already decided how many RTS's should be created and what they should contain. This section focuses on these latter issues.

The remaining questions to be answered are: What are the criteria for the creation of new RTS's in a given CSL program, and what should these RTS's contain. Actually, we may narrow down the scope of the questions further. Since our focus for scheduling rests on parallelism within Cobegins, and the smallest unit of scheduling is a stream, we may restate our questions as: What are the criteria for deciding how many RTS's to create in a given Cobegin, and how should we distributed the streams of the Cobegin to the new RTS's.

To recapitulate, the policy of the scheduler is to minimize the CSL program execution time. Of some concern also is the efficient use of resources. (Efficiency is defined as the speed-up divided by the number of processors used). One of the guidelines used in this regard is that if an error in the schedule must be made, it would be better to err on the side of too few RTS's than too many. At least then, fewer resources would be used. The use of resources, however, is not of so great a concern that the *efficiency* of a schedule is maximized rather than the *speed-up*. Efficiency comes into play only when discussing the point where an "N" processor Cobegin should become an "N+1" processor Cobegin.

In this section we first discuss the question of how many RTS's to create. We refer below to the "weight" of a stream. The weight of a stream is a scalar that helps determine the propensity of its Cobegin being parallelized. The sum of the weights of the streams are used in the final calculation.

In analyzing the streams in a Cobegin, two distinctions can be made. One is whether the stream contains an Execute, Send, or Receive. The other is whether the stream contains more than one statement.

Although it is not customary to write parallel streams that use none of: Execute, Send, or Receive; if such a case occurs, the weight of such a stream should be zero. (It is noted again that these statements completely dominate the time required to complete a Cobegin). The execution times of statements without them are negligible. On the Cyber, this ratio is greater than 100 to 1.

An analysis of real programs written in CSL show that at least half of the parallel streams written in CSL contain a single statement. Those with more than one statement typically have a series of Sends, or a series of Receives. Often, an Execute is used before or after such a series. The contention is that in these cases no special weighting should be given a stream with more than one statement. The use of Sends and Receives almost invariably implies that these streams are proceeding in lock-step fashion with each other, or may start after an execute has completed in another stream. This in turn means that the RTS's are operating with bursts of management work. Therefore, a multiple statement stream is no more important than a single statement stream. It is also contended that any stream (single statement or not) containing a Send, Receive or Execute should be rated equally for the very same reasons. The Send or Receive and the Execute are invariably dependent on each other in some form, and create bursts of work for the RTS's.

The use of the Cobegin statement however, has not been mentioned in this context. The issue of scheduling Cobegins within Cobegins brings up some non-trivial problems. Should the scheduling algorithm weight a stream differently if it contains a Cobegin? The ramifications of this question are evidenced in an example.

Consider two RTS's that each manage 5 streams. One of the RTS's contains a Cobegin in each of its streams. The other has 5 parallel execute statements. If the nested Cobegins are large, they may be parallelized independently, however, if all 5 nested Cobegins are small, no extra parallelism will be provided. This means that one of the RTS's could have a large load to manage alone, even though the sum of all the streams it manages would warrant the creation of a new RTS.

The only way to optimally handle this situation is to monitor the work being done by each RTS at run-time, and make the necessary decisions then. It would take a very intelligent dynamic scheduler to do this. For one thing the execution of each stream could be probabilistic based on the value of some condition. Furthermore, even with no probabilistic conditions, it is not certain that all 5 nested cobegins could even be in execution at the same time, unless a detailed analysis of the dependencies between the Cobegins is made. Finally it is considered that this case is a second-order factor in the design of the scheduler, and that the number of unavailable quantities during an execution in the first-order factors mentioned earlier suggests that the additional scheduler complexity would not buy much in terms of additional performance.

So far, then, we have any stream containing a Send, Receive, Cobegin, or Execute as having an equal weight of one. One aspect not discussed is the possible weights of Executes associated with varying task execution times. This first version of the scheduler does calculate, if possible, the average execution time of all the streams containing tasks for use in the schedule. If a stream has more than one Execute written in a serial fashion, then the execution time for that stream is the sum of its tasks' execution times. Since the initialization time (I) for each task is known, the execution time may be used to determine the I/E ratio which in turn indicates the desirability of creating RTS's. If I/E is low, then more streams must be present before a new RTS is created.

Putting together our heuristics for this section, we now make an attempt at answering the first question -- that of how many RTS's to create. The algorithm consists of:

1) Calculate W, the sum of the weights of the streams
   for a given Cobegin.
2) Calculate E, the average execution time of the tasks

```
   to be executed.  Assume standard values if not
   known or ask for guestimate values.
3) Using the result of 2, generate a value N.  N = f(E).
4) Calculate (W/N).  This is the number of new RTS's to
   create.
```

The value N represents the point where a single RTS has all the streams it can handle efficiently on average. The assumptions here are that N is a function of the average execution times of the tasks. Also, the scheduling function implies that the number of RTS's should be linear with increasing N. (In the next chapter on performance, some of these notions are challenged). The basic purpose of the performance evaluation, then, is to model the system and derive N = f(E) using various loads. Actually, a great deal more was gained.

The second major question of this section is: once a number of RTS's for a given Cobegin is decided upon, what should they contain? This can be answered quite simply -- a balanced load. Actually, the implementation of this is fairly straightforward.

Firstly, since the weight of each stream is the same, the scheduler may simply partition them on an equal-number basis. Of course the fact that the Master-RTS must "Execute" the other RTS's needs to be taken into consideration. Any remaining streams with weight one are distributed evenly starting with the first RTS to be invoked by the Master-RTS. Any weight zero streams may be added to the Master-RTS.

Secondly, no performance advantage exists for having streams with implicit dependencies in the same RTS (for instance, a corresponding Send and Receive being located in the same RTS). A new shared table access must be made whether or not the same RTS is used. Therefore no attention need be made to any dependency relations.

Thirdly, if the execution times of the tasks are known, and if some are much longer than others, it would be best to try to distribute their Execute's as evenly as possible over the RTS's. A further improvement can be realized if these Execute's are put at the beginning of the list of streams so that they may be started sooner.

In summary, the static scheduling method was chosen because of its simplicity and its ability to execute most CSL programs more quickly than the best dynamic method on the Cyber Plus system. The system calls on the Cyber necessary to perform task initiation and communication are extremely expensive. Therefore, the PARTS Scheduler is actually more effective in a static form.

Chapter 6

Model and Performance Evaluation

The previous chapters have described the structure of PARTS and the basic scheduling method. The purpose of this chapter is to form a model of PARTS and evaluate its performance.

The goals here take several forms. First, we show analytical results for the completion time of a Cobegin in a simple system model. Next we develop a more realistic model and use simulation methods to demonstrate how the factors of influence affect performance. Finally, we use the results of the model to develop a better scheduling algorithm for PARTS. In all cases performance is defined as the speed-up of the completion time of a Cobegin due to the parallel structuring of PARTS.

6.1 An Analytic Evaluation

A simple model of PARTS can be evaluated with a queueing network. In this section the purpose is to calculate the time it takes to complete a Cobegin on this basic model.

Figure 6-1 shows the queueing model for a single RTS. A burst of N tokens leaves the gate and enters queue I, the Initialization node. Each token upon leaving queue I enters the Process node. The Process node has N servers, so the queue wait time is zero. The tokens then enter the Finishing node and are finally collected at the gate.

Let I equal the initialization time. This value is constant for all tokens. Let P1, P2, ... ,PN be the amounts of time each stream spends in the execution state. Let F equal the Finishing time. This queueing model then represents the basic features of what transpires during the course of a Cobegin. There are, however, some assumptions made in this model:

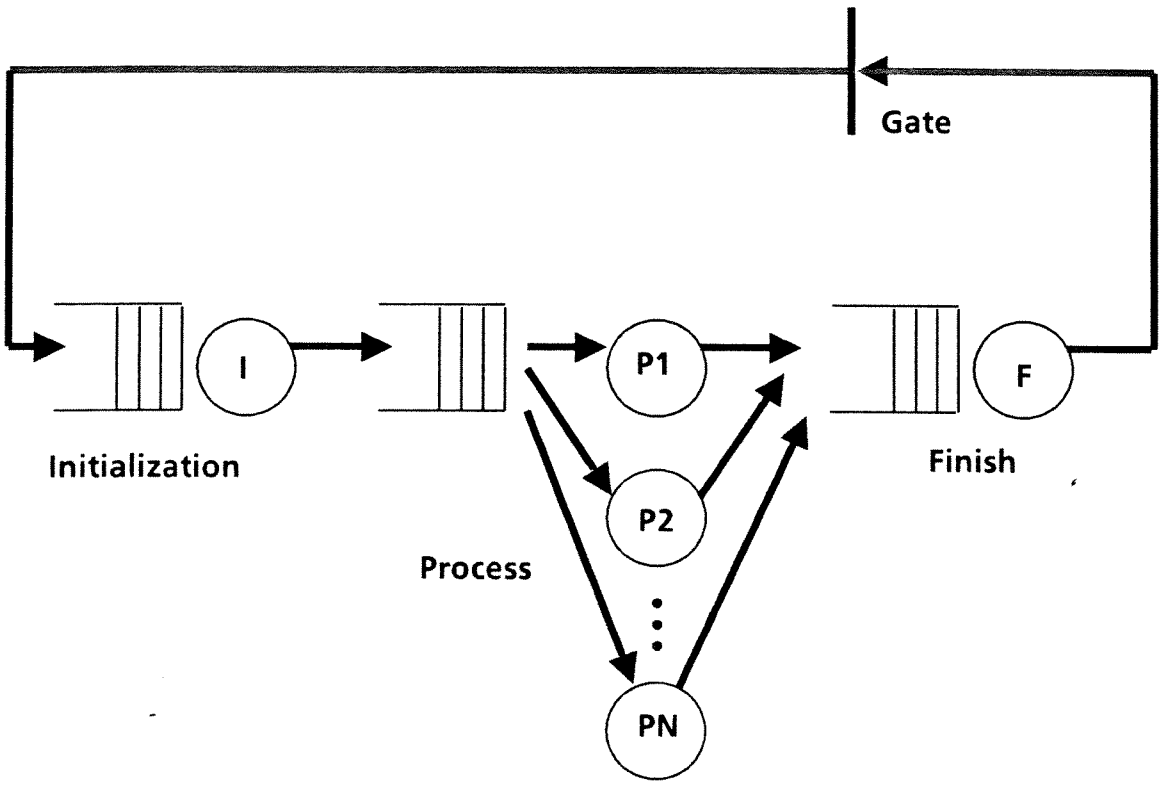

Figure 6-1: Queueing System for a RTS

- No Data Dependencies. Each of the N streams are independent with no communication requirements. This is a basic assumption of all queueing networks since token service times can only depend upon the load at that service center, not upon a global system state. (The effect of communication on this model will be discussed at the end of this section).

- Each stream contains one statement. The model can approximate streams with more than one statement if the execution time is viewed as the sum of all operations in the stream.

- Constant Service Time. This is a valid assumption since I and F are constants anyway, and P1,...,PN are assumed to be known for the CSL program.

- F <= I. This simplifies the calculations and is a valid assumption in the RTS since F is approximately equal to I.

Given the preceeding assumptions, the completion times of a Cobegin may be calculated. For the cases where more than one RTS is employed, it is assumed that no RTS start-up overhead exists, and the RTS's shared memory access time is zero.

	CONDITIONS:	RESULTS:
1)	1 RTS 1 Stream	I + P1 + F
2)	1 RTS N Streams P = P1 =...=PN P >= NI	NI + P + F
3)	1 RTS N Streams P = P1 =...=PN P < NI	NI + P + kF 1 <= k <= N
4)	1 RTS N Streams P's are variable	Max [I+P1,...,NI+PN] + kF 1 <= k <= N
5)	R RTS's N streams P = P1 =...=PN P >= NI	(N/R)I + P + F
6)	R RTS's N Streams P = P1 =...=PN P < NI	(N/R)I + P + kF 1 <= k <= N/R
7)	R RTS's N Streams P's are variable	Max[I+P[1],...,(N/R)I+P[N/R], I+P[N/R+1],...,(N/R)I+P[2N/R], ...,...,....., I+P[(R-1)N/R+1],...,(N/R)I+PN] 1 <= k <= N/R

Case 1 considers a single stream to be processed by the RTS. Case 2 allows N streams that all perform the same amount of work and where no stream finishes until all have been initialized. Case 3 relaxes the latter of the two assumptions. In case 3, as P varies from NI to 0, k approaches N as a linear function. Case 4 relaxes the assumption that the streams have to do the same amount of work. Cases 5-7 repeat the work of 2-4 except that multiple RTS's evenly divide the management work. It is assumed that N/R is an integer, that I and F do not vary from RTS to RTS, and that the RTS's split the streams equally among each other.

The results of the different cases reveal several major points. First, the value of P is

very important in determining when the Cobegin will complete and how much parallelism can be taken advantage of in the RTS's. A comparison of examples 2 and 5 shows that the only time savings with parallel RTS's occurs in the Initialization node. The factor of importance in determining speed-up then becomes the I/P ratio. If the P's are not all the same, then examples 3 and 6 show that the Finishing node plays a part. Here the advantage lays in the fact that in example 6, the value "k" is limited to a much smaller range.

The value "k" is an indicator of the manner in which the streams complete. It can be thought of as the number of tokens in the Finishing node at the time the last stream finishes execution. If k=1, then the last stream to finish executing is immediately serviced by the RTS. This is good since it implies that the RTS had not built up too big of a workload and was ready to service the stream. A large k on the other hand can mean one of two things. Either all the streams completed at about the same time, or some streams completed before others were even started. Both of these conditions causes the use of parallel RTS's to be more effective.

If the P's vary somewhat it can be seen that there would be an advantage in starting the streams with the largest P's first. This is partly offset by the fact that this tends to make the streams finish at closer intervals to each other. It can be shown, though, that as long as F <= I, the total completion time can never be better than scheduling the largest P's first.

The preceeding model has a basic shortcoming in not providing for the communication between streams allowed by CSL. If communication takes place during the execution of N streams in the above situation, we may set the lowerbound of completion time at: I + Max[all P's] + F. The upperbound is: NI + NF + NC + sum[all P's], where C is the average communication time between streams assuming communication does not need to be managed. (The upperbound is the case where all operations occur serially). In the case where all streams are symmetrical (i.e. execute the same code and send/receive the same amount of data), the completion time is exactly: NI + P + C + NF, where C is the amount of time one process spends in communication transfers. Here again, it is assumed that communication does not have to be managed.

The basic point about communication is this. Although it is very difficult or impossible to derive general expressions for completion time in Cobegins when communication is involved, the effect of communication is to create bursts of work rather than spreading it out evenly. In the symmetrical stream case above, which would seem to be an advantageous situation since communication takes place in parallel, the value of "k" equals N, implying that all the streams complete at the same time. These bursts of work actually allow parallel RTS's to provide better speed-up potential in PARTS. If we relax the assumption that communication does not have to be managed, we find even greater workloads for PARTS and therefore more speed-up potential.

Therefore, the fact that the PARTS Scheduler considers streams as executing independently does not hamper its ability to cause good speed-ups in CSL program execution. If anything, communication will cause speed-ups to increase over what is originally estimated. This paradox is exploited throughout the design of PARTS.

Using the model in Figure 6-1, we may see that the speed-up obtainable in all cases is linear with respect to N. It is also valuable, however, to determine how the speed-up varies with respect to changes in the process time P. This is shown in Figure 6-2.

The assumptions here again are that the RTS startup overhead is zero with no shared memory contention between RTS's. Also the calculations assume that P = P1 = ... = PN, and that N, I and F are constants. R is the number of RTS's employed in the system.

T1 and T2 are the most interesting data points in the system. For process times less than T1, neither a single RTS nor R RTS's experience any idle wait time (waiting for a stream to finish so that it can be managed.) For process times between T1 and T2, the R RTS's experience idle wait time, but the single RTS does not. This means that as process times increase, the speed-up ratio of the R RTS's falls off rapidly. For process times greater than T2, both forms of RTS's experience idle wait time; and the function decays to a speed-up of 1, but at a slower rate.

In order to operate at the desirable point (T1,R), we may solve for the number of RTS's to be generated. For instance, with 10 streams each lasting 80 MS, and an initialization time of 20 MS, we have R = (10/(80/20+1) = 2 RTS's. Since the RTS's in this model are assumed to

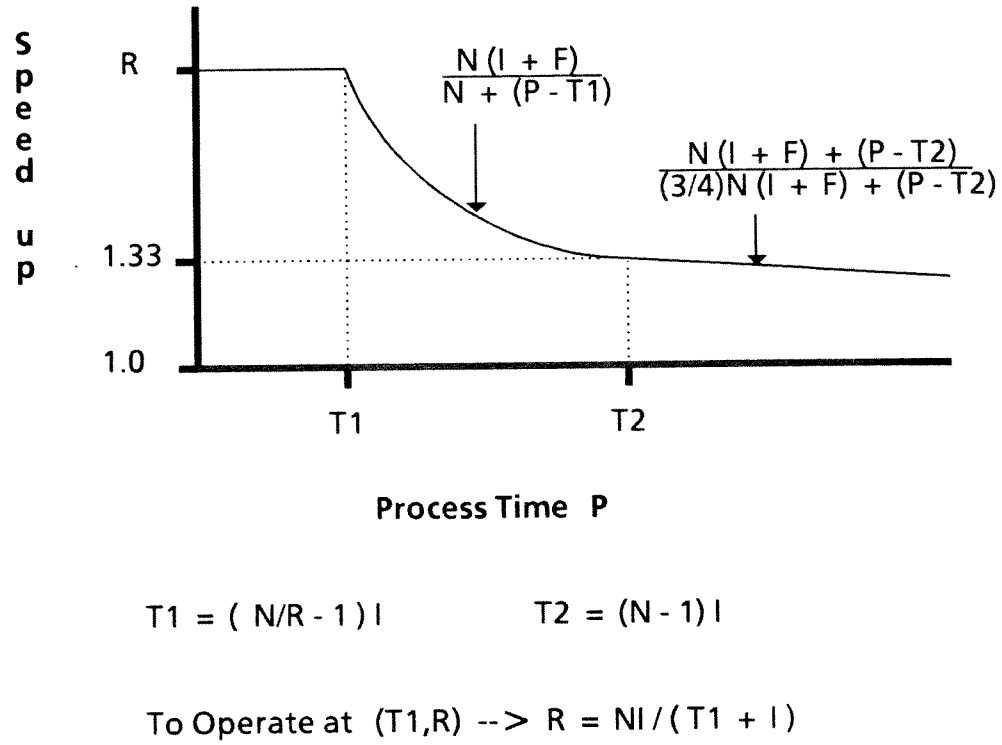

Figure 6-2: Speed-up as a Function of Process Time

have ideal properties, it would not be prudent to use this result directly in scheduling. What the result does constitute, however, is an upper bound for scheduling decisions. In the following sections a model is introduced that allows a much better analysis to take place.

6.2 Simulation Using PAWS

In order to evaluate a more detailed model of PARTS, a simulation language called PAWS was used.

The Performance Analyst's Workbench System (PAWS) is an event driven simulation language especially designed for the modelling of computer and information systems. PAWS uses a pictorial construct for modelling called an Information Processing Graph (IPG). The IPG can be thought of as a set of nodes at which information is processed. Upon completion of processing at a node, information may flow from one node to another for additional processing.

Information Processing Graphs are used to describe, pictorially, the information flow as well as the specification of the processing done at specific nodes. The model is first represented as a collection of nodes and edges. Nodes are those elements in a model at which some activity occurs. Edges specify the direction of information flow as well as the probability of such an occurence if more than one path is provided. The transactions that pass though the IPG are classified by category and phase. The category of a transaction is fixed for the life of the transaction, while the phase may be caused to change. Categories are used to allow the gathering of statistics on different classes of transactions, even though they may flow through the same model. Phases are often used for routing within the model.

The activities at the nodes may be classified two ways, those that advance the simulation clock and those that do not. For the purposes of this thesis, those that advance the clock are queued servers with a distribution. Other nodes used that do not advance the clock are those that create transactions, destroy transactions, perform calculations on phases for routing, and allocate and release shared memory. The edges developed for the model of PARTS all have a probability of 1.0 for each transaction.

6.3 Parameters of the Model

The parameters of the simulation model incorporate all the essential factors of influence in the CSL RTS. They may be grouped in terms of system parameters and load parameters. Where possible, the system parameters are fixed using real data obtained from analyzing a single RTS executing on the Dual Cyber 170/750.

The system parameters are:

- The number of RTS's executing in parallel.
- The task initialization time. This value was determined to be 4 +- 1.25 MS on the Cyber system and is used as such.
- The task finishing time. This value was also determined to be 4 +- 1.25 MS.
- The RTS invocation time. Since the PARTS Scheduler allows RTS's to be executed as if they were tasks, this value is also 4 +- 1.25 MS.

- The shared memory access time. This is the length of time of an average access to the shared memory tables after the table has been acquired. This value is not precisely known although it is expected to be less than 0.5 MS (one eighth the task initialization time).

The load parameters are:

- The number of streams.
- The execution time of the tasks in the streams.
- The variance of the execution times.

There are then, five variable parameters left to be explored. They are the number of RTS's, the shared memory access time, the number of streams, and the execution time and variance of the tasks. In the next section the simulation model is presented.

6.4 A Model for PARTS

In this section the models are presented for a single RTS, and a multiple RTS system that simulate the management of a Cobegin. The Information Processing Graph (IPG) format is used. Figure 6-3 shows the IPG for a single RTS managing 3 streams.

A transaction is generated at Source. At Fork, 3 new transactions are generated with phases 1,2 and 3. The transactions flow into the node IANDF which performs the initialization and finishing functions of the RTS. The transactions are then routed to the execution nodes E1, E2, and E3. The compute node adds 3 to each phase so that when the streams are finished, they may be routed to the node Join. The semantics of the Join node coincides perfectly with that of the Cobegin, since the transaction does not leave the Join node until all three forked transactions have arrived. PAWS allows the servers to differentiate the service times based on the phase of the transaction. In the case of IANDF, however, this is not needed due to the fact that both initialization and finishing times have the same distribution. The number of streams may be varied by adding execution nodes and forking more transactions at the fork node. The execution times of the tasks may be varied by independently altering the service times of each execute node. The variance of the execution times may be controlled by altering the variance of the service distribution. This completes the representation of a single RTS.

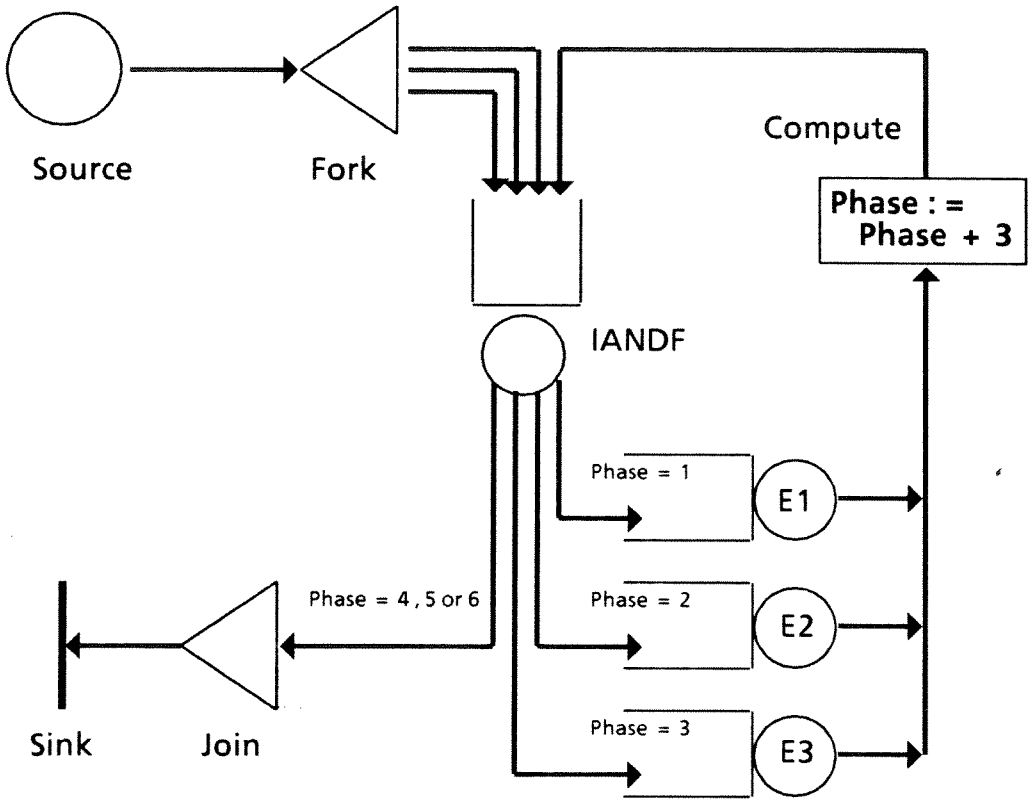

Figure 6-3: IPG for a Single RTS with 3 Streams

The model of a multiple RTS system is shown in Figure 6-4. The Fork node forks 3 transactions that correspond to the 3 RTS's to be executing in parallel. The node IRTS delays RTS2 and RTS3 corresponding to the RTS invocation time. RTS1 does not have to be initialized, but is delayed 1 clock tick in node Delay1 corresponding to the fact that RTS1 must initialize RTS2 and RTS3 before it may begin managing the rest of its streams. (Each RTS then enters its appropriate network. Each RTS network corresponds to the IPG shown in Figure 6-2 without the source and sink nodes). On completion of their networks, RTS2 and RTS3 must re-enter the IANDF node associated with RTS1 corresponding to the fact that RTS1 is responsible for their management.

This completes all aspects of the model except that of acquiring shared memory. Figure

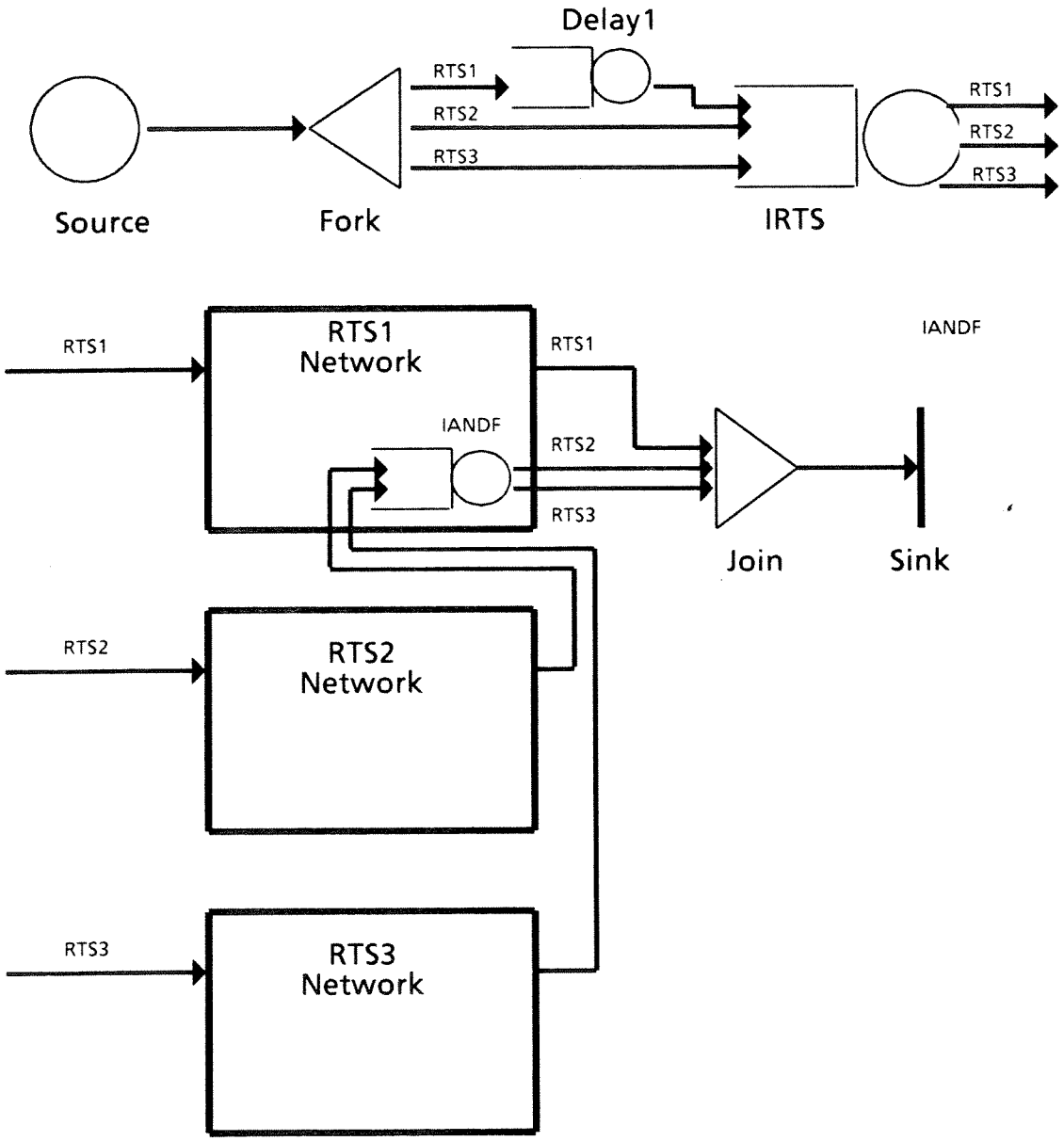

Figure 6-4: IPG for a 3 RTS System

6-5 shows this portion of the model. The nodes Acquire Table and Release Table work together to provide mutually exclusive access to the node Table Delay. A queue is formed in front of this access. The distribution of this delay corresponds to the amount of time each RTS spends accessing the shared memory.

In the model, each RTS acquires the table every time a transaction enters its IANDF

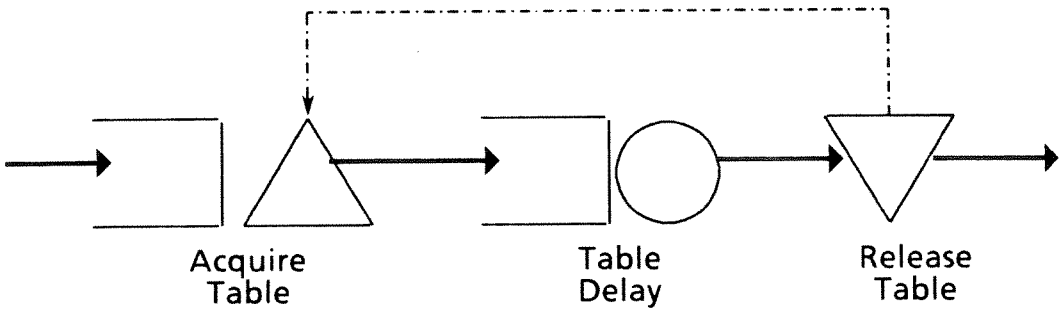

Figure 6-5: Shared Memory Acquisition

node. (In other words, each stream is assumed to need a shared table access before and after its execution). Only one copy of these nodes exists in the model.

The model of shared memory access differs from PARTS in the following ways. In PARTS, three shared tables could be accessed: the Task Table, the Shared JVTable, and the Shared Variable Table. In the model these three are consolidated into a single table. This increases contention between the RTS's. In PARTS, several accesses of the Shared JVTable may take place before and after each stream's execution. In the model all accesses are modelled as the sum of all their access times. This will increase the average wait times for memory access, but should not noticeably affect throughput. In PARTS, accesses to the shared table may need to be made more than twice per stream if the stream has more than one statement. In the model, streams are considered as if they were single statements, so this affect is not considered. This will reduce shared table contention somewhat, however, the main source of contention at the beginning and end of the streams is still modelled.

6.5 Simulation Results

Four important experiments are conducted. First, it is illustrated what speed-ups are obtainable with constant stream execution time. Second it is illustrated how variance about the mean stream execution time affects performance. Third, variance is explored regarding sets of streams with widely varying mean stream execution times. Fourth, it is illustrated how the increase in shared memory access time affects performance.

Unless otherwise stated, the shared memory access time is assumed to be the worst-case expected value of 0.5 MS with a variance of 0.25 MS. All distributions are erlangian in order to model the low variance of the service times.

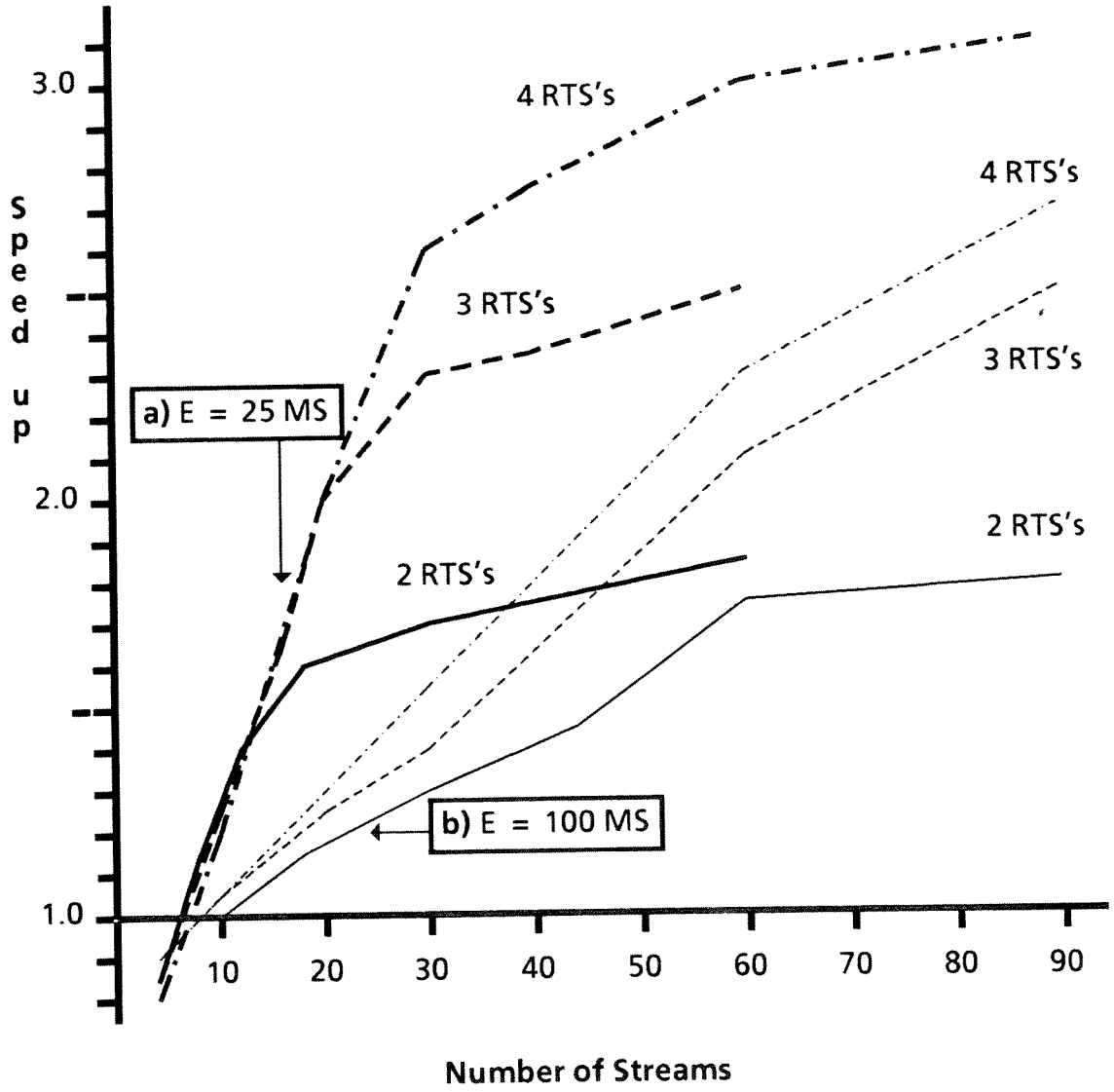

Figure 6-6: Speed-ups for Constant Stream Execution Time

Figure 6-6 shows the speed-ups obtainable by 2, 3 and 4 RTS's executing in parallel over a single RTS. In Figure 6-6a the stream size is 25 MS, that of an average CSL application. In

Figure 6-6b the stream size is 100 MS. In this case the same speed-ups may be obtained, but only if a sufficient number of streams are present.

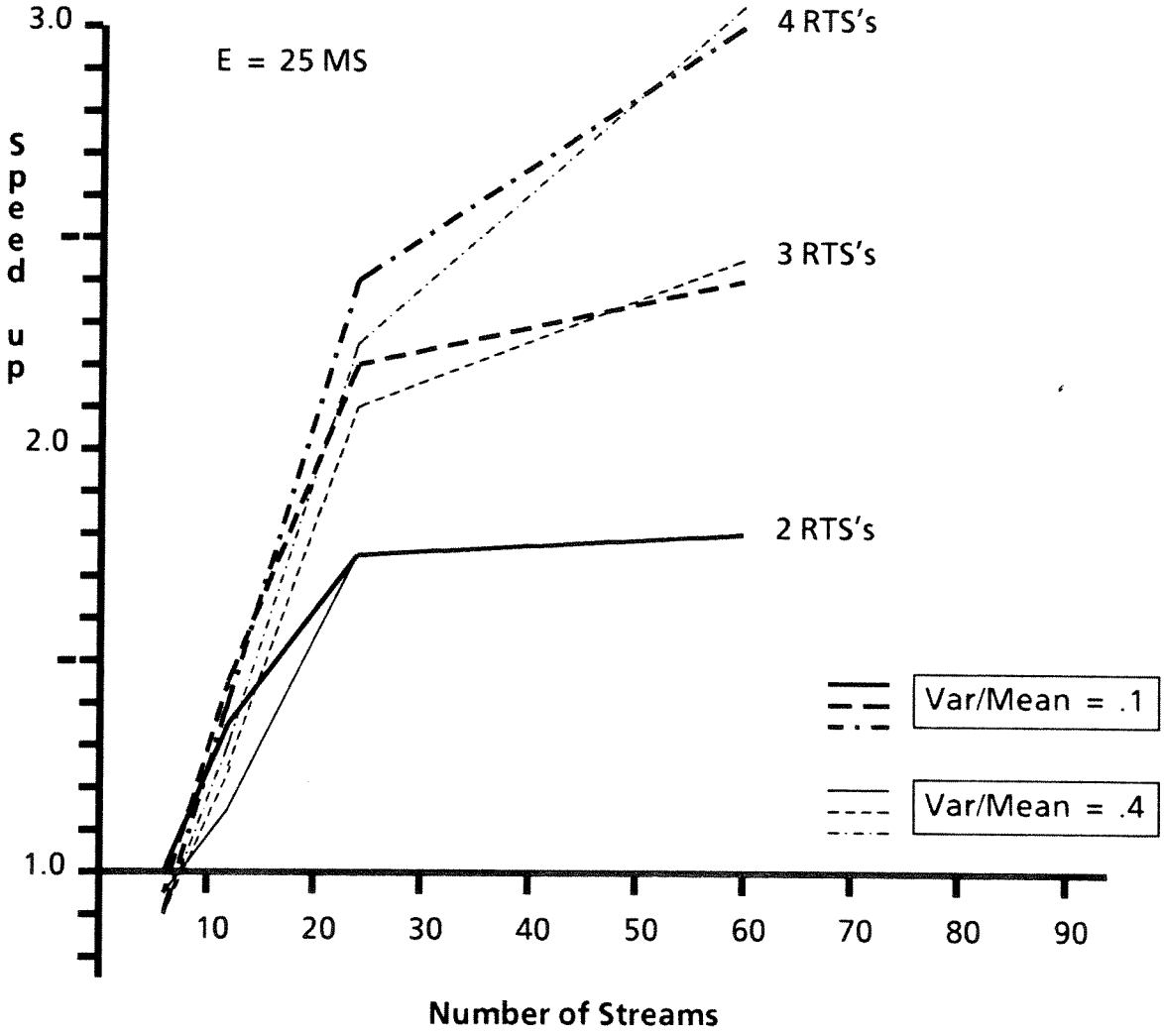

Figure 6-7: Speed-ups for Streams with Variance

Figures 6-7 and 6-8 show how an increase in the variance of stream execution times hurts performance. For Figure 6-7 where the variance/mean ration is .1, the performance decrease is almost negligible. If the variance/mean ratio is increased to .4, then major performance decreases are noticed until the number of streams is equal to about 25. The fact that these curves start off sluggish but yet make up their lost ground is an interesting

phenomena. It is explained in the following way. The initial performance decrease is due to the fact that the increased variance causes the streams to naturally finish at different times, causing there to be a more even workload on a single RTS's rather than a burst of work. When more streams are added, however, the sheer number of streams causes the RTS to overload, even though their finishing times may be spread out. Finally at 60 streams or more, it is noticed that slightly more speed-up is obtained. This is because the larger variance causes some streams to finish much faster than average, allowing the RTS to start work on them sooner!

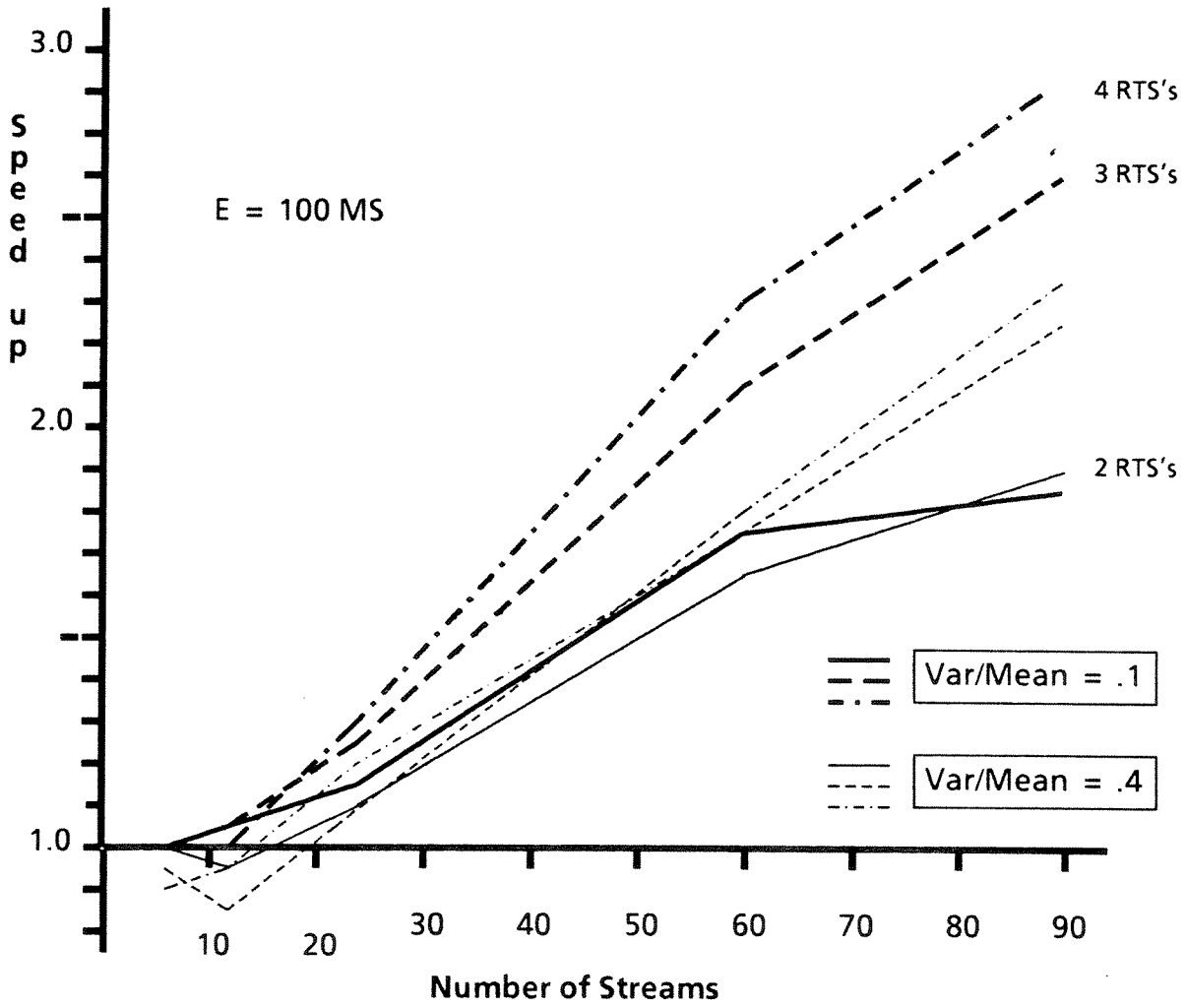

Figure 6-8: Speed-ups for Streams with Variance

In Figure 6-8 for the variance/mean ratio equaling .1, some performance degradation is

noticed already. This is because the value of the variance is much larger now, even though the percentage is not. When the variance/mean ration equals .4, the first dips in performance start to become very pronounced.

In the third experiment, the variance is not a variance about a mean, but rather comprised of two groups of streams. The first group of streams comprise 2/3 of the total number of streams and have an execution time of 25 MS. The other 1/3 of the streams execute for 100 MS. The longer executing streams are spread evenly over the multiple RTS's. In Figure 6-9 it is shown that the speed-up is about the same as all the streams being 100 MS! The speed-up curve where all streams are 25 MS is shown for reference.

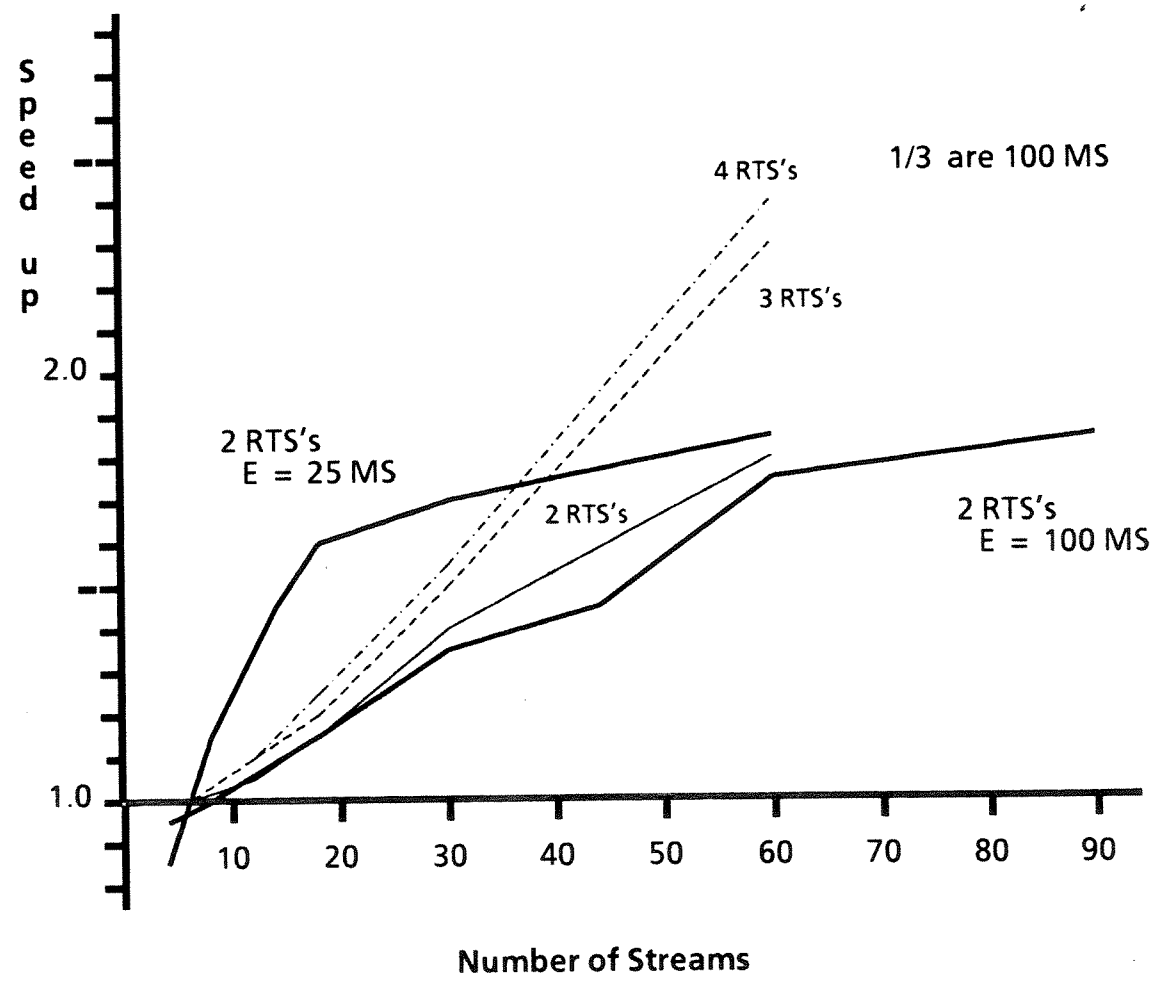

Figure 6-9: Mixed Groups of Stream Execution Times

In another experiment a single 100 MS stream was executed within a group of 25 MS streams. The simulation compared a 2 RTS system to a single RTS system. The speed-up here was actually worse than the case where all the streams are 100 MS, and was less than 1.0 up to 18 streams. The basic point is that a single long stream will remove any opportunity for speed-ups until the enough of the smaller streams exist that their final completion time becomes longer than that of the longest stream.

The fourth experiment increases the time spent in an average shared memory access to 2 MS, four times larger than before. Figure 6-10 shows the dramatic reduction in performance associated with the increase in shared memory access time. The new curves have the same shape as the old, but the point of diminishing returns is reached after a two or three RTS system is configured. This is a rough analogy to the work balance equation that states that the speed-up of an N processor shared memory system is limited by the probability of each processor executing in a given critical section, where Max[N] should be no greater than 1/p [Cezzar 83]. In the example here the shared memory access time is one-half that of stream initialization time, thus p = 1/3. This cannot be used as a strict rule since the RTS's also encounter periods of idle wait time, however it does form a good heuristic for an upperbound schedule.

It is demonstrated by the graphs that the most important parameters in determining the speed-up obtainable by PARTS are the shared memory access time and the length of the longest stream in the Cobegin. The next section uses these guidelines to improve the scheduling algorithm.

6.6 A Better Scheduler

To determine how many RTS's to schedule during a Cobegin, some criteria must be used. If the policy is strictly to maximize speed-up, then the number of RTS's corresponding to the speed-up curve that maximizes speed-up for a given number of streams should be chosen. However, this may result in very poor efficiency characteristics. In order to allow efficiency to have some bearing on the scheduling decisions, an arbitrary parameter is chosen such that the number of RTS's will not be increased unless the speed-up associated with the increase is at least some value "x". For the PARTS Scheduler a good value to choose is 0.2.

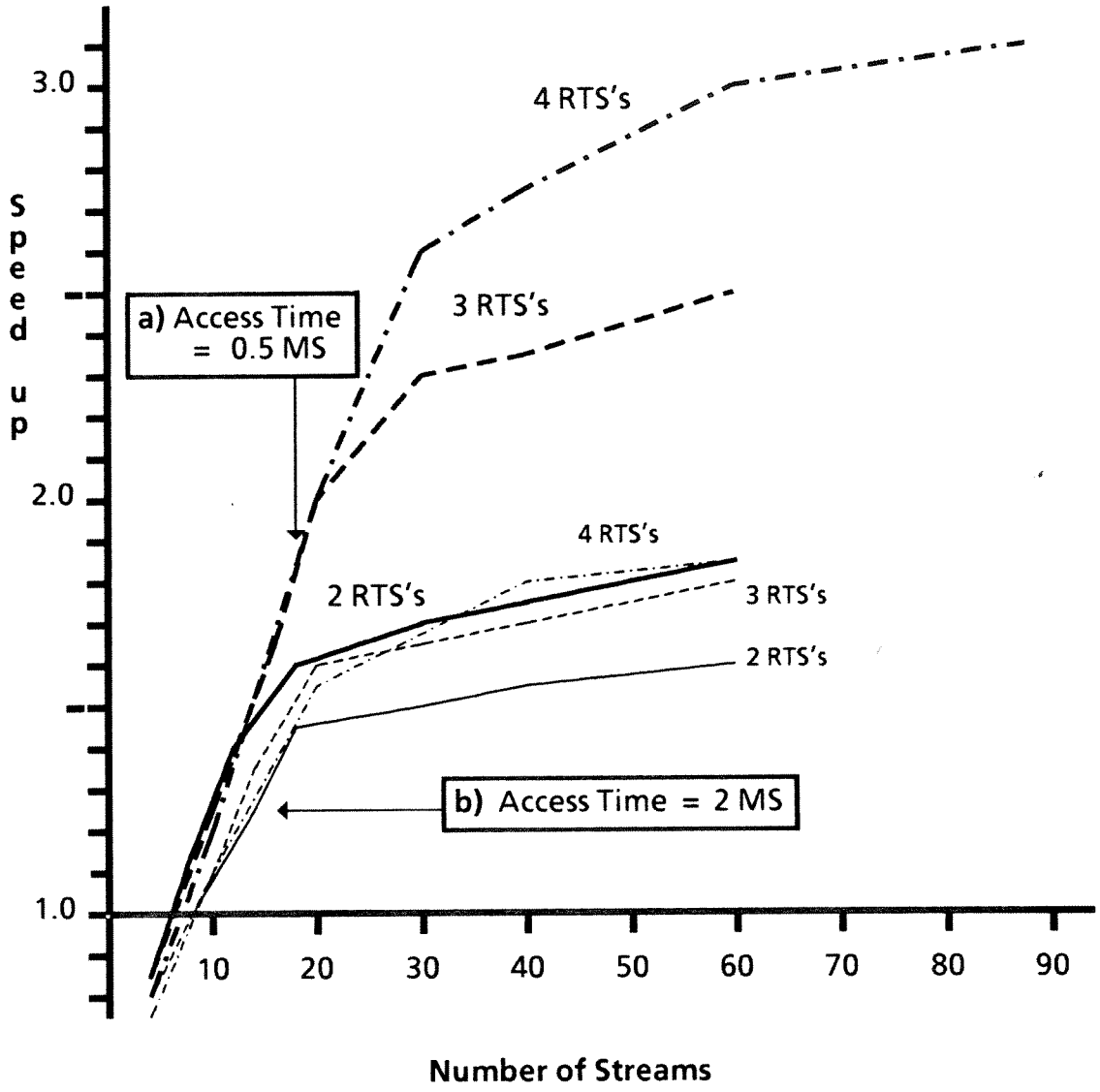

Figure 6-10: Shared Memory Access Time Effects

Using the value 0.2 as a basis for scheduling, Figure 6-6 yeilds the following general scheduling pattern.

E = 25 MS (E/I = 6.25) E = 100 MS (E/I = 25)

Streams	RTS's	Interval	Streams	RTS's	Interval
0 - 10	1	--	0 - 20	1	--

10 - 17	2	7	20 - 35	2	15
17 - 30	3	14	35 - 65	3	30
30 - 58	4	28	65 -125	4	60
58 -114	5	56			

An analysis of the this data shows a pattern that develops in the performance curves. Using data obtained from Figure 6-6 and other simulation runs for E/I ratios of 1.6 and 100, it was determined that the number of streams X, at the point where speed-up equals 0.2 is fit very well by the function X = 4*SQT(E/I). The function that describes the size C, of the interval for the operation of 2 RTS's is closely fit by C = 7*SQT(E/I/6.25). C increases by a factor of 2 for each new RTS scheduled. The constant 6.25 was chosen to calibrate the function for the case where E = 100 (100 units = 25 MS), an expected common actual operating point of the RTS.

The manner in which the variance in Figures 6-7 and 6-8 deviate from the curves associated without variance is very difficult to quantify. The speed-ups were just barely affected when the variance/mean was 0.1, however, were very noticeably affected when speed-ups were 0.4. This suggests a nonlinear relationship between speed-up degradation and increasing variance. The reader can see that the point where 1 RTS should become 2 (the number of streams at which the speed-up equals 0.2) has moved to the right. This increase in the number of streams seems to be almost negligible for a variance/mean of 0.1, but equal to E/I streams for a variance of 0.4. If a square relationship between performance degradation and variance/mean is assumed, then the number of streams by which the curves initially move to the right can be fit by (Var/0.4)**2*(E/I), where Var is the variance/mean.

Taking variance into account, then, the point where one RTS should become two is X = 4*SQT(E/I) + (Var/0.4)**2*(E/I). Variance is not used in the calculation of the interval size C because of the complexity of the relationship. This absence, however, should not be of much significance. It should cause the schedule to be just slightly conservative in those cases where a large number of streams (more than 30) exist having a large variance (greater than 0.3).

Once it has been determined that more than 1 RTS is needed, the decision for scheduling more RTS's is based on the interval increase parameter C. If the number of streams N being greater than X is within C units of X then only 1 more RTS is needed. If the number of streams

is between C and 3C units from X, three are scheduled. If between 3C and 7C, four are scheduled, etc. In each case the interval doubles in size.

Putting the results together, we have a general scheduling guideline for determining R, the number of RTS's from N identical streams:

$$R = \log_2 ((N-X)/C + 1) + 2 \quad \text{for } N > X$$

$$= N/X \quad \text{for } N <= X$$

Where:

$$X = 4*SQT(E/I) - (Var/0.4)^2 * (E/I)$$

$$C = 7 * SQT(E/I/ 6.25)$$

Assumes:
- Shared memory access time <= 1/8 I.
- E/I >= 1.
- I <> 0.
- I = F.
- N > 0.
- All streams have the same mean execution time.

Figure 6-11 shows the graphical representation of the scheduling equation. R is the number of RTS's, N is the number of streams, Var is the variance/mean, and E/I is the stream execution to initialization ratio. Since the equation calculates the number of *additional* RTS's to schedule during the Cobegin, the value 2 is added. The equation will most likely lose some of its effectiveness if any one of the following holds: $E/I < 1.5$, $N > 100$, $Var > 0.5$, or $R > 7$, however, none of the CSL programs written so far would satisfy any of these conditions.

Since the preceeding scheduling equation assumes that all streams have the same execution times, the schedule is still fairly limited in scope. One naive solution to this problem would be to simply schedule the longest streams without regard to any others. This will work as long as the number of smaller streams is still small enough that the RTS can finish them before the longer streams begin to complete.

The approach taken here is to estimate beforehand, for each group of streams with the

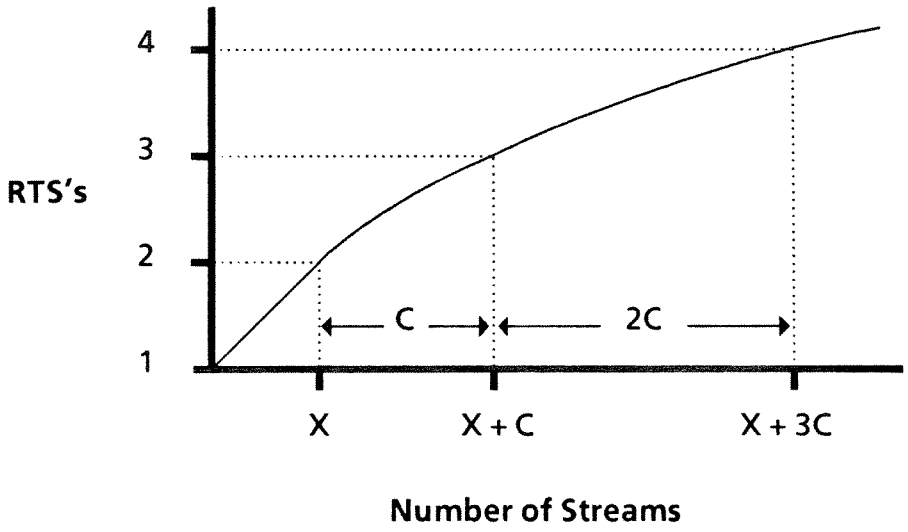

Figure 6-11: The PARTS Scheduling Equation

same mean execution time, whether the group will complete before some larger group. If so, this group is disregarded in the schedule. The parameter that is used to estimate the completion time of the last stream is called Q, where Q = 2 * I * Num + E. Num is the number of streams with mean execution time E. The parameter Q is an exact estimate of completion time if the variance of the stream execution time is zero, and a very reasonable estimate if either N is large, or the variance is small.

The scheduling algorithm for those streams containing tasks is now written as:

```
Schedule Groups

1)  Calculate the schedule R, using the
    equation above independently for
    each group of streams independently
    using E, I, and N.

2)  Calculate Q for each group.

3)  Create A, an array of triples (Q,E,R).

4)  Find the integer J such that A[J].Q is
    the Max(Q) in A.  Insert J into SET.
    (SET is a set of integers).
```

```
5)  Call Create_Set( J, SET).

5)  The Schedule is assigned to the sum of all
    independent schedules R in A such that the
    index in A is a member of SET.
```

The algorithm Create_Set is shown below:

```
Procedure Create_Set( J; var SET);

    If A[J].E < any other Q whose index J is not
       already in SET

       Then
         Insert those J's into SET.
         Find M such that A[M].Q is the Min(Q)
             for all J's in SET.
         Call Create_Set( M, SET).
       Else
         Skip.
```

The approach is illustrated in an example. Consider a Cobegin with streams that last 1000 MS, 800 MS, 500 MS, and 100 MS. The first step is to estimate when all of these groups of streams will complete using the parameter Q (which is dependent on how many streams of each group are presend). Suppose the Q's are 1200, 1100, 900, and 300 respectively. The algorithm determines the following facts: The group of 1000 MS streams cannot be scheduled alone since while an RTS is completing them (during the period of 1000 to 1200) the group of 800 MS streams will still be completing (1100 being greater than 1000). Thus the 1000 MS and 800 MS need to be scheduled as a group. Now consider the 800 MS group. The group of 500 MS streams will still be completing when the 800 MS group starts completing. Therefore, they also must be considered in the group with 1000 MS and 800 MS. The 100 MS streams however, do not cause any overload of an RTS to occur, being finished by the time the 500 MS streams start completing, therefore the schedule for the 100 MS streams can be disregarded. The total schedule, then, is the sum of the schedules for the 3 remaining groups.

An area of impreciseness still exists in this algorithm along the boundary of the decision for whether to include the group of streams in the overall schedule or not. The algorithm could

be refined to scale the amount of each individual schedule that is calculated for the final result. For instance, if the 800 MS group had completed at 1020 rather than 1100, there would be some overlap, and therfore some need for additional parallelism, but not as much as if there was a complete overlap of the intervals in which the streams complete.

Finally, putting together all the results and insights we have obtained, we may revise the scheduling algorithm in Chapter 5 to determine a better criteria for deciding: How many RTS's to create.

Scheduling Algorithm

1) Determine the weights of the streams in the Cobegin.

2) Disregard any streams with weights of 0.

3) Collect the streams into two groups: those containing executes, those containing only Sends and Receives.

4) Find or estimate the execution times of all the tasks.

5) Find or estimate the variance/mean for all the tasks. If not known assume 0.1.

6) For those streams only containing Sends and Receives group them together with those streams containing the tasks that the Sends and Receives refer to. Otherwise distribute them evenly over such streams. In both cases, consider the these streams as having execution times equal to those of the streams they are associated with.

7) Calculate the execution time E for each remaining stream. E is equal to the sum of the execution times of the tasks in a given stream.

8) Group the streams by execution time and calculate the schedule using the Schedule Groups algorithm and equation above. Round the result down to the nearest whole integer. If the result is less than one, assume one.

Chapter 7

Conclusions and Summary

From the simulation model in the last chapter it can be seen that the PARTS Scheduler can be used to increase the speed-ups of CSL programs logarithmically with respect to the degree of parallelization in those programs. This assumes, though, that the unit of work for each parallel stream remains the same while the degree of parallelization increases.

In many parallel algorithms, however, this is simply not the case. As the parallelism increases, the amount of work done between required communications decreases (often linearly), thus the task execution times decrease. The simulation and analytical results of Chapter 6 show, however, that the performance of PARTS increases with decreasing task size (see Figure 6-2). This yeilds a very encouraging conclusion. The performance decrease associated with decreasing communication granularity can be significantly offset by the increase in management parallelism!

This thesis has studied the parallel structuring of resource management in the context of the Run-time System for the CSL parallel programming system. It is one of the first case studies for parallel structuring of a real resource management system.

A method of parallel structuring the CSL Run-time System was presented that uncovered and exploited the large-grained parallelism in its management functions.

The mechanisms for scheduling the Parallel Run-time Systems were presented that allow the Run-time Systems to be executed as if they were ordinary tasks, reducing the complexity. Parallelism within the scheduling itself was exploited by allowing Run-time Systems to invoke and manage each other in a multilevel master-slave tree-shaped structure.

A detailed scheduling algorithm was developed based on the evaluation of analytic and simulation models of PARTS. The simulation model was also used to determine the effectiveness of the scheduling algorithm based on the CSL programs already written.

To parameterize the system model, real data was acquired from the operation of a running version of the CSL Run-time System on the CDC Dual Cyber 170/750 computer system. Also, real versions of CSL programs were to used to formulate the basic scheduling strategy.

Parallel structuring of resource management is perceived as essential to alleviate the performance bottleneck that results from large numbers of processors being scheduled and managed. The ever-present goal of increasing the speed of program execution through the use of large numbers of processors assures that this issue will play an important role in the development of parallel computing.

Bibliography

[Andrews 82] Andrews, G. R.
Distributed Allocation with Pools of Servers.
ACMSigact-SigopsConf.onDist.Computing :73-83, 1982.

[Arvind 77] Arvind, K. P. Gostelow, and W. Plouffe.
Indeterminacy, Monitors, and Dataflow.
OSR 11(5):159-169, November, 1977.

[Barak 82] Barak, A. B.
Dynamic Process Control for Distributed Computing.
Proc.3rdIEEEConf.onDist.Systems :36-40, 1982.

[Browne 81] Browne, J. C., A. Tripathi, et. al.
A Language for Definition and Control of Reconfigurable Parallel Computation Structures.
1981.
May be found in 1981 ICPP.

[Browne 82] Browne, J. C., A. Tripathi, et. al.
A Language for Specification and Programming of Reconfigurable Parallel Computation Structures.
ICPP , 1982.

[Browne 83] Browne, J. C.
Basic Software for High Performance Parallel Architectures.
1983.

[Browne 84] Browne, J. C.
Overview and Functional Architecture of the CSL System.
Apr, 1984.

[Browne 85] Browne, J. C.
Representation and Programming of Parallel Computations: A Unified Approach.
ICPP , 1985.

[Cezzar 83] Cezzar, K. and D. Klappholz.
Process Management in a Speedup-Oriented MIMD System.
ICPP :395-403, 1983.

[Dekel 81] Dekel, E. and S. Sahni.
Parallel Scheduling Algorithms.
ICPP :350-351, 1981.

[Dijkstra 65] Dijkstra, E. W.
Solution of a Problem in Concurrent Programming Control.
Communications of the ACM 8:569, 1965.

[Dijkstra 74] Dijkstra, E. W.
Self-Stabilizing Systems in Spite of Distributed Control.
Communications of the ACM 17:643-644, 1974.

[Gottlieb 81] Gottlieb, A., B. D. Lubachevsky, et. al.
Coordinating Large Numbers of Processors.
ICPP :341-349, 1981.

[Gottlieb 83] Gottlieb, A., R. Grishman, et. al.
The NYU Ultracomputer - Disigning an MIMD Shared Memory Parallel Computer.
IEEETOC C-32(2):175-189, Feb, 1983.

[Ho 83] Ho, L. Y. and K. B. Irani.
An Algorithm for Processor Allocation in a Dataflow Multiprocessing Environment.
ICPP :338-340, 1983.

[Hoare 74] Hoare, C. A. R.
Monitors: An Operating System Structuring Concept.
Communications of the ACM 17(10):545-557, October, 1974.

[Jayaraman 80] Jayaraman, B. and R. M. Keller.
Resource Control in a Demand-driven Data-flow Model.
ICPP :118-127, 1980.

[Jayaraman 82] Jayaraman, B. and R. M. Keller.
Resource Expressions for Applicative Languages.
ICPP :160-167, 1982.

[Jayaraman 83] Jayaraman, B.
Constructing a Parallel Implementation from High-Level Specifications: A Case Study Using Resource Expressions.
ICPP :416-420, 1983.

[Kruskal 82] Kruskal, C. P.
Algorithms for Replace-add Based Paracomputers.
ICPP :219-223, 1982.

[Kuck 77] Kuck, D. J.
A Survey of Parallel Machine Organization and Programming.
ComputingSurveys 9(1), Mar, 1977.

[Lamport 74] Lamport, L.
A New Solution of Dijkstra's Concurrent Programming Problem.
Communications of the ACM 17:453-455, 1974.

[Leinbaugh 81] Leinbaugh, D. W.
High Level Specification of Resource Sharing.
ICPP :162-163, 1981.

[Lo 81] Lo, V. and J. W. Liu.
Task Assignment in Distributed Multiprocessor Systems.
ICPP :358-360, 1981.

[Madnick 74] Madnick, S. E. and J. J. Donovan.
Computer Science Series: Operating Systems.
McGraw-Hill, Inc., 1974.

[Mohan 85] Mohan, J., A. Jones, et. al.
Granularity of Parallel Computation.
Proc.18thHawaiiICSS :249-255, 1985.

[Mundell 81] Mundell, K. J., M. W. Linder, et. al.
Processor Allocation in Data Driven Systems - Two Approaches.
ICPP :156-157, 1981.

[Ousterhout 82] Ousterhout, J. K.
Scheduling Techniques for Concurrent Systems.
Proc.3rdIEEEConf.onDist.Systems :22-30, 1982.

[Papazoglou 81] Papazoglou, M. P., P. I. Georgiadis, et. al.
Process Synchronization in the Parallel Simula Machine.
ICPP :297-299, 1981.

[Reed 79] Reed, D. P. and R. K. Kanodia.
Synchronization with Eventcounts and Sequencers.
Communications of the ACM 22:115-123, Feb, 1979.

[Sejnowski 80] Sejnowski, M. S., G. J. Lipovski, et. al.
An Overview of the Texas Reconfigurable Array Computer.
NCC :631-641, 1980.

[Shen 85] Shen, C. C. and W. H. Tsai.
A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion.
IEEETOC C-34(3):197-203, Mar, 1985.

[Sips 84] Sips, H. J.
Task Distribution on Clustered Parallel- or Multiprocessor Systems.
Proc.4thIEEEConf.onDist.Systems :126-130, 1984.

[Tilborg 80] Tilborg, A. M. and L. D. Wittie.
A Concurrent Pascal Operating System for a Network Computer.
COMPSAC :757-763, 1980.

[Tuomenoksa 82] Tuomenoksa, D. L. and H. J. Siegel.
Analysis of Multiple-Queue Task Scheduling Algorithms for Multiple-SIMD Machines.
Proc.3rdIEEEConf.onDist.Systems :114-121, 1982.

[Wang 85] Wang, Y. T. and R. J. Morris.
Load Sharing in Distributed Systems.
IEEETOC C-34(3):204-217, Mar, 1985.

[Wegner 83] Wegner, P. and S. A. Smolka.
Processes, Tasks, and Monitors: A Comparitive Study of Concurrent Programming Primitives.
IEEE Transactions on Software Engineering SE-9(4):446-462, July, 1983.

[Williams 83] Williams, E.
Assigning Processes to Processors in Distributed Systems.
ICPP :404-406, 1983.