EVALUATION OF PARALLEL PROGRAMMING LANGUAGES

Mary Kathleen McShea

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-86-22 September 1986

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation and Goals

As research in parallel processing continues, dozens of languages and environments are being proposed as vehicles for expressing parallel algorithms. We need to establish an organizing framework of thought with which to approach the evaluation of any particular parallel programming language.

Some reasons for examining a parallel programming language are:

- to determine its suitability for a particular application
- to determine its adaptability to a particular architecture
- to determine its expressive power and general usefulness

The last reason is, of course, the most compelling, and is the motivation for this research. As parallel machines become more common, more powerful, and less expensive, we can expect the same trend in parallel programming languages as was described in [Ambler 78] for operating systems languages: programs will need to be portable as hardware changes, and software economics will overtake hardware economics as the dominating factor in choosing languages. A high-level language is more portable, and easier to program. [Ambler 78] defines a "high-level" language as one that programs

algorithms, not machines. We will take this algorithm orientation as the starting point for developing a set of desirable qualities for a general-purpose parallel programming language.

The focus for this thesis sprang from experience in developing algorithms to test CSL, the Computation Structures Language developed at the University of Texas at Austin [CSL 85]. Problems in implementing some simple parallel algorithms led to the question of what properties are embodied in these algorithms that must be expressible in a general-purpose parallel programming language. Additionally, it was decided to code these same algorithms in other parallel programming languages for comparison with the CSL versions.

The goal of this thesis is to develop metrics for evaluating parallel programming languages, based on a set of simple parallel algorithms that display different properties of computation graphs. Special emphasis is given to CSL, and a small set of extensions is proposed to increase CSL's generality and usefulness. Other existing parallel languages have been examined, and classified according to the nature of their control constructs for parallelism. Representative languages have been chosen from each subclass, and the same algorithms have been coded in these languages.

1.2 Related Work

There have been several comparative surveys and analytical discussions of parallel programming languages and constructs.

Andrews and Schneider [Andrews 83] survey notations for process creation, synchronization, and communication, and propose a classification

scheme for concurrent languages based on models of process interaction: procedure-oriented (shared memory), message-oriented, and operation-oriented (remote procedure call, combining aspects of the first two classes). The languages they examine include Concurrent Pascal, Modula, CSP, DP, and Ada.

Wegner and Smolka [Wegner 83] do a comparative analysis of the concurrent programming primitives for process creation and synchronization in the languages Ada and CSP, and the synchronization construct monitors. Among the conclusions they draw are: that Ada's one-way naming communication construct is more general than CSP's two-way naming, and that CSP has a more general nondeterminism construct than Ada.

Bloom [Bloom 79] performs an analysis of classes of synchronization mechanisms, including path expressions, monitors, and serializers, using a set of synchronization problems to test their expressive power. Her evaluative technique is similar to the one adopted in this thesis.

[Stotts 82] enumerates a large number of issues in the design of parallel programming languages, among which are synchronization, communication, process creation, process network topology specification, and ease of use. He tabulates these features for 13 languages, including Ada, CSP, Concurrent Pascal, Path Pascal, and Parlance.

The model of parallel algorithms used in this thesis is a simplified version of the computation graph model described in [Browne 85].

1.3 Organization

Chapter Two describes the classification scheme for parallel languages, the set of parallel algorithms used for evaluation, and the metrics of evaluation. Chapters Three through Six discuss the results of the evaluations for the languages Ada, Occam, Sisal, and CSL respectively. Chapter Six also proposes a set of extensions to CSL that extend its generality, while preserving most of its philosophy of separation of parallelism control from computation. Chapter Seven summarizes and discusses the results.

1.4 About the Codes

The Ada codes in this thesis were executed on a Digital VAX/VMS. Occam 2 was not available at the time of writing. The Sisal codes have also not been executed. The unextended CSL codes are legal CSL, but the Pascal task body codes associated with them have been simplified from what is required in the first implementation.

Chapter 2

Parallel Algorithm Characteristics and Language Evaluation Criteria

2.1 Model of Parallel Computation

The model of parallel computation used here to describe both languages and algorithms is a computation graph model based on [Browne 85]. A parallel algorithm is considered to be a directed graph consisting of nodes and arcs. The nodes represent schedulable units of computation, which are sequential code segments, and may be at any level of granularity. The directed edges (arcs) represent dependency relations between the nodes. These dependencies are either synchronization or data dependencies.

Synchronization dependencies specify either an ordering of node executions, or a mutual exclusion constraint where the nodes may not execute simultaneously, but no order is specified. Data dependencies specify an ordered transmission of data between nodes. Two nodes connected by a data dependency arc may be executing concurrently on an ordered stream of data items. Mutual exclusion arcs are a form of data dependency when the connected nodes are accessing a shared data item. In this case, no ordering is specified.

A node executes only when its activation is enabled by its input arcs (in the case of arcs joined by an OR relationship, the enabling of one of a set of

arcs is sufficient). A node may or may not have a state which persists between activations; a node with persistent state is frequently termed a process. After or during its execution, the node enables its outgoing arcs. This can be a result of nothing more than its termination or by actual data transmission. These correspond respectively to synchronization and data dependency arcs.

The set of nodes and arcs describing an algorithm is the computation graph for that algorithm. Any particular legal partial ordering of execution of nodes and enabling of arcs is a traversal (execution) of that computation graph. A coding of a parallel algorithm in a parallel programming language must express both the structure of the computation graph of the algorithm and its traversal (and all legal schedules for that traversal). Examples of computation graphs are given with each sample algorithm below. In the example computation graphs, solid lines with one arrow head represent a total ordering between two nodes, as in data dependency or execution precedence. Dotted lines with two arrow heads represent mutual exclusion constraints.

2.2 A Testbed of Parallel Algorithms

2.2.1 Characteristics of Parallel Algorithms

Three characteristics of parallel algorithms have been used to select a sample set of problems with which to test the representative languages. The characteristics are:

1. model of internode communication
2. creation of the computation graph
3. traversal of the computation graph

Model of internode communication

This thesis uses two basic models of communication between nodes: the shared memory model and the message model (also called the channel model). These two models represent endpoints on a spectrum of models, with shared memory being the most unconstrained method of communication, and channels being the most constrained. By constraint is meant the rules enforced in any use of the model for communication. The shared memory model enforces no rules on the access to data; values need not be written before being read, any node with knowledge of the variable may read or write it, and values exist for the duration of their scope. The basic channel model enforces the rules of read before write, destruction of values after they are read, and one to one directed transmission of values. Other models can be built on these two basic models; two such are the mailbox model and the tuple space of the language Linda [Gelernter 85A,Gelernter 85B] They are intermediate both in the latitude they provide the programmer, and the amount of programmer-specified synchronization required to use them. Mailboxes and Linda tuple space will also be discussed below.

The shared memory model assumes that some subset of the nodes can access the same globally shared variables, which are assigned to and read from equally by all of them. Any synchronization, such as priority, mutual exclusion, or write-before-read constraints must be programmed by the user, with low-level constructs such as semaphores, or high-level ones such as monitors. This model is attractive in some ways because it corresponds to the old operating systems model of concurrent processes on uniprocessors, and is thus familiar, and also because many new parallel architectures actually

use shared memory between processors. There are difficulties with the shared memory model in that some synchronization problems may be hard to understand and code correctly. This problem has been discussed extensively in the literature.

Some algorithms lend themselves particularly well to a formulation using shared memory, however, regardless of the architectural issues. These algorithms can be thought of as a set of nodes that operate on a large shared data structure which has an internal coherency and a relatively permanent existence with respect to the nodes, as for example, a parallel quicksort, which sorts an array of numbers. When an algorithm requires a set of nodes to process repeatedly the same set of data, coding the algorithm using the alternative message model may be cumbersome, and counterintuitive.

The message model is equivalent to a highly constrained and disciplined use of shared variables. A message containing data is sent over a channel (or between "ports"), a one-to-one connection between nodes. The channel is like a shared variable (or set of shared variables) with the constraints that it may not be read until written, and that when read, the value is removed, and is no longer accessible outside the reading process. The data in this case can be considered to be created whenever it is written, and destroyed when it is read. It has no permanent existence outside the execution of the nodes. A generalization of this one-to-one channel model is the broadcast channel, which is one-to-many. In this case, the channel is empty until the source places a value in it, and the value remains there until all of the destinations have received the value. A broadcast channel can also be thought of as an array of one-to-one channels. In either the one-to-one

or broadcast channels, all synchronization is built into the semantics of the constructs of send and receive, and the burden of programming it is removed from the programmer, who must, however, work within this context. This is the basis of the programming language CSP [Hoare 78] and its descendant Occam [Pountain 86]. It should be noted that parameter passing mechanisms correspond to channel communication between a procedure or function and its caller.

Message-based communication is more natural for algorithms in which all communication is one-to-one and there are no permanent data structures involved. A special case is that of a fixed set of nodes through which a stream of data is pumped and transformed, corresponding to a systolic type of architecture (although with no assumptions about lockstep execution of the nodes). Another, more general, example is the parallel simulation given later, where the dynamically created set of entities send messages back and forth to each other; no globally shared data structure is inherent in the problem, and each entity communicates with at most one other at a time.

The mailbox model is a somewhat looser version of the channel model. Data is placed in a shared location called a mailbox. After it has been placed there by a node, other nodes may remove it and use the data. Unlike the channel model, there is no one-to-one connection implied. Any node with knowledge of the name or location of the mailbox may access it, so a mailbox may be considered a many-to-many channel. A mailbox is shared, but unlike shared variables, mailboxes have the same built-in semantics of write before read as channels, so the programmer does not have to specify this synchronization. The mailbox model could be implemented using shared

memory or channels.

The tuple space of Linda's generative communication model is a somewhat more constrained version of shared memory [Gelernter 85B]. A set of processes have a common space for data, like a giant mailbox. The space is called tuple space because data items are a set of components with names and/or values, like a tuple in a relational database. Four primitive operations are available to the processes for reading and writing tuples and these provide the synchronization. The operation out() places a tuple into the space, and the operation in() withdraws a tuple with matching components from the space. These two operations are like channel send and receive. Another operation, read(), allows a process to access a tuple without withdrawing it from the space. This is similar to a shared memory read-only access. The fourth operation, eval(), places a tuple in the space, just like out(), except that the tuple may have components that are not evaluated by the process executing the eval() call. Instead, the components are evaluated when the tuple is added to the space, with an implicit process performing the evaluation.

All of the four models discussed are equivalent since any internode communication can be programmed using them. This thesis will use only the shared memory and channel models to describe the algorithms, because they differ most from each other, and because many parallel programming languages incorporate one or both.

Creation of the computation graph

This may be either static or dynamic, corresponding to compile-time or run-time creation of the graph. A static computation graph has its entire set of node and arc instantiations determined before any traversal of the graph begins. This fixed topology is appropriate when the parallel structure of the algorithm is unaffected by the actual computation occurring in the nodes, as for example, in a parallel matrix multiplication. In this case, the parallel algorithm is unaffected by the contents of the matrices upon which it operates. All parallel programming languages have the ability to specify a completely static computation graph.

A dynamic computation graph is created as it is traversed. New nodes and interconnections are added as the computation progresses, based on the results of the computation up to that point. This corresponds to defining types of nodes, and then creating new instantiations of the node types as needed. A truly general-purpose parallel programming language needs the ability to define node types, instantiate new nodes, and link them to the rest of the computation graph by dependency arcs at runtime. An example of an algorithm with dynamic graph creation is the recursive quicksort given later. Each execution of a node doing the array partitioning may give rise to two more identical nodes that operate on subsets of the array, or it may terminate, when the base of the recursion is reached. A nonrecursive algorithm with dynamic computation graph creation is the parallel simulation given later: a number of "patient" nodes come into existence at random intervals, and are connected to the graph at that time.

Traversal of the computation graph

Traversal of a parallel computation graph includes execution of nodes along with association of data with nodes (either message transmission, or allocation of shared data to a node). Graph traversal may, like graph creation, be either dynamic or static. Traversal of a computation graph may follow a fixed pattern regardless of the data or other conditions at runtime; again, the matrix multiplication example is a case in point. The same steps will always be followed and the same variables accessed in the same order, or the same number of messages sent and received.

However, traversal of the computation graph can frequently depend on intermediate results of the computation; this includes both the execution of nodes and the enabling of arcs. Some examples of this property follow:

- A schedulable unit of computation may execute for a number of repetitions that depends solely upon its input data, and send a varying number of messages after each execution. This is the case in the pipeline example given later, where the central node, SQUASH, transmits some of the data it has received, either changed or unchanged, and suppresses other data.
- For a shared memory model algorithm, the extent of access to a partitioned shared variable may depend on the results of the previous operations upon that data. This is the case in the parallel quicksort, where the size of the partition of the shared array given to each partitioning node is not determined until just before it executes.

- Yet another example is given by the parallel simulation algorithm. A "doctor" node is not aware of what "patient" it is supposed to communicate with next, until so informed by the "nurse" node, whose information may change with each execution. In other words, the choice of the data arc to activate is dependent on the computation.
- Another common type of dynamic graph traversal is unprioritized mutual exclusion on shared data. Any of a set of nodes may access the data in any order, but not simultaneously. One would expect the actual order of access to vary at run-time in the general case. This example appears in the block triangular solver given later.

2.2.2 Example Algorithms

Four algorithms were chosen to form the basis for a set of simple parallel algorithms that would test the capabilities of a parallel programming language. They were chosen to use varying combinations of the three parallel algorithm characteristics given above. The algorithms will be described in their ideal form, with maximum parallelism, minimum constraints, and optimal type of inter-node communication. These descriptions will serve as the canonical form with which to compare the implementations achieved in the various languages, and a list of desirable properties of the implementations is included with the algorithms.

Parallel Block Triangular Solver

Characteristics:

- static computation graph structure
- shared memory communication
- static computation graph traversal except for unprioritized mutual exclusion on vector blocks as described below.

Algorithm: This algorithm is due to Jack Dongarra and Danny Sorensen of Argonne National Laboratory. A system Ax=b is partitioned into square blocks, and solved for x. The array A and vector b are kept in shared memory, and the vector x is solved for in place in vector b (so that the solution vector writes over vector b).

The matrix A is given to be in lower triangular form, so the algorithm corresponds to the back-substitution phase of the solution. The system is partitioned into square blocks of dimension nxn, $1 \leq n \leq N$, where N is the dimension of the original system. Normally, n is an integral divisor of N. The blocks above the diagonal of the matrix are discarded, and the resulting block system is solved as follows:

- Each block on the diagonal, A(i,i), comprises a miniature lower triangular system with the corresponding block b(i), and is solved using the normal sequential method of back substitution. The nodes assigned to the diagonal matrix blocks are called SOLVE.
- Each internal block of A is used to modify the blocks of b by normal sequential matrix vector multiplication: b(i) := b(i)- A(i,j)*b(j). The nodes assigned to the internal matrix blocks are called MATVECT.
- The following execution constraints must be observed between the nodes. All MATVECTs in a row of A must complete before the SOLVE for

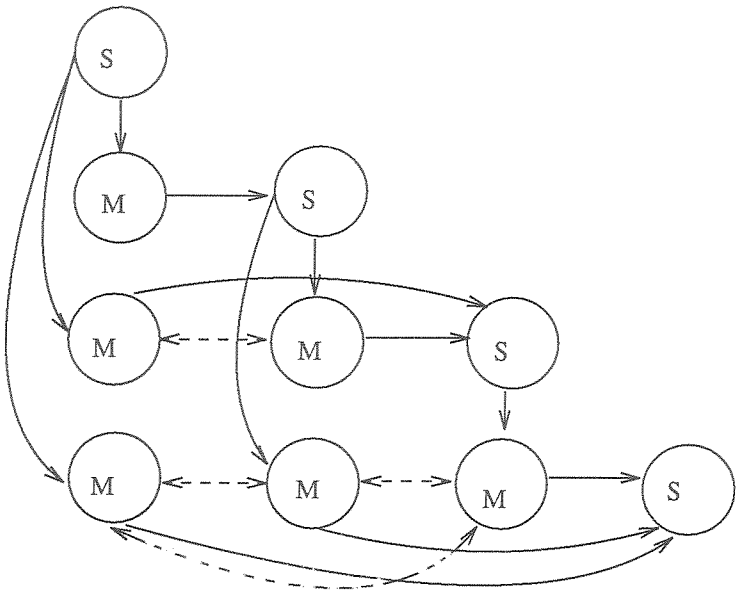

Figure 2.1: Computation graph for triangular solver

that row may begin. A SOLVE must complete before the MATVECTs in the column below it may begin. The first SOLVE, corresponding to matrix block A(1,1), may begin immediately. MATVECTs in a row must be serialized arbitrarily in their execution, since each modifies the same block b(i).

Figure 2.1 gives the structure of the computation graph for the triangular solver algorithm. In this figure, solid arcs denote precedence dependencies, and dotted arcs denote mutual exclusion dependencies.

The requirements for implementing this algorithm are:

1. The nodes should be structured in a method natural to the problem. Each node must correspond to a block in the matrix, and there must be a convenient way of associating the correct data partition of both vector and matrix with the correct node. Additionally, one would like to be able to specify only a lower triangular structure, rather than a matrix, only half of which is useful.

2. The synchronization of the nodes should be specified (or implied) so that maximum parallelism is possible in any execution, regardless of relative speeds of the processors assigned to the nodes. In this problem, the only possible variation of activation of nodes is in the interior MATVECT nodes, where a row of MATVECTs may be arbitrarily serialized in their access to b(i). Although it does not affect the final outcome of the algorithm, it is desirable that no constraint not necessary to the problem be placed in the implementation, so the order of MATVECT executions in a row should not be predetermined.

3. Shared memory should be used to contain the matrix A and vector b. Besides seeming natural to the structure of the computation graph, which corresponds to the structure of the matrix, shared memory is the easiest way to express the read-only access of a column of MATVECTs to a single block of the vector b. Without a shared data structure, the same data must be duplicated and transmitted to as many MATVECTs as require it, although they will not change the block of data.

Parallel Quicksort

Characteristics:

- dynamic computation graph creation
- shared memory communication
- dynamic computation graph traversal: assignment of variable-size array partitions to nodes

Algorithm: This is a simple parallelization of the well-known recursive quicksort algorithm. The nodes are pieces of code that partition an array into two parts, according to the first value in the array: the section of the array containing values less than the partitioning value is again partitioned by another node, and so is the section containing values equal to or greater than the partitioning value. If a schedulable unit of computation is given an array segment of size 2 or less, it sorts it, and the recursion terminates. The size of the array segment associated with any given node instantiation depends on the contents of the entire array at run-time. Figure 2.2 gives the structure of a sample execution of the parallel quicksort algorithm. Numbers in brackets represent array values before node execution. The numbers in parentheses are the associated indices of the shared array.

The requirements for this algorithm are:

1. Dynamic creation of only the necessary nodes and their dependencies should be available. Upper and lower bounds on the number of nodes required can be calculated from the size of the data, but the relationships between them cannot be determined before runtime.
2. Shared memory should be used for the array to be sorted, if possible. No explicit mutual exclusion synchronization is required on the shared data, since it is divided repeatedly into disjoint subsets. The other form

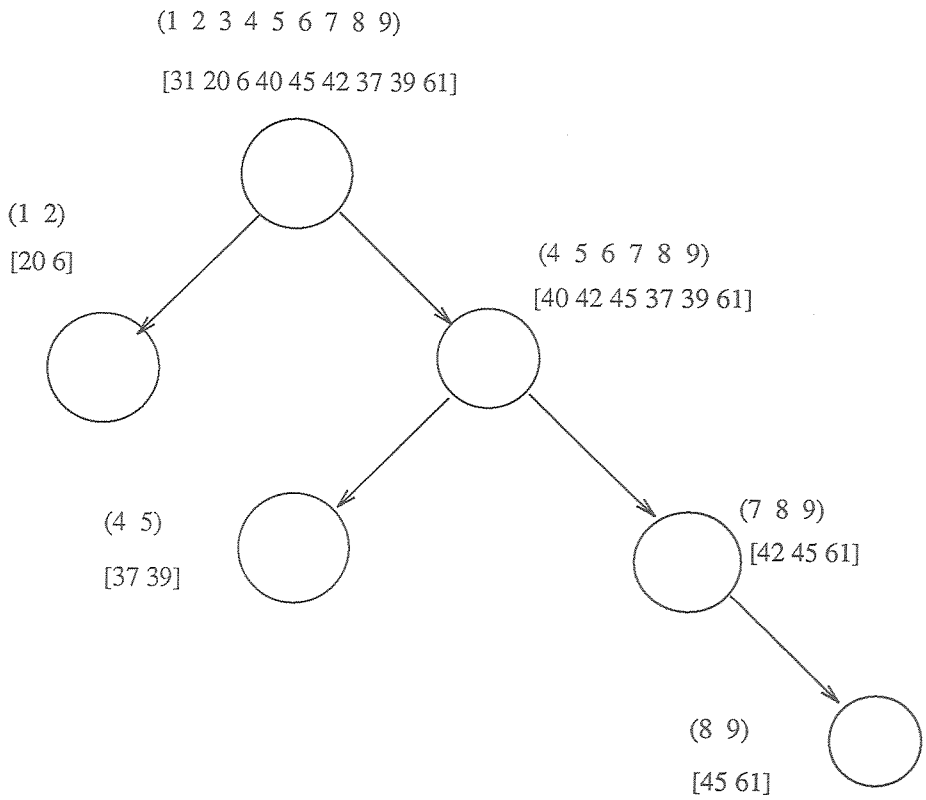

Figure 2.2: Computation graph for an execution of a parallel quicksort

of internode communication in this algorithm is the transmission of the start and length of each array segment to its partitioning node, and message communication, as in the parameters of a recursive call, can be used.

3. Maximum parallelism between nodes should be allowed. All partitioning nodes can be executing at the same time, except for those nodes which are predecessors on the same set of data (those will have terminated after calling their successors in any case).

Text Processing Pipeline (Conway's Problem)

Characteristics:

- static computation graph structure
- message based communication
- dynamic computation graph traversal: data-dependent control of message transmission

Algorithm: This algorithm is known as Conway's problem [Conway 63] which was designed to illustrate the use of coroutines for asynchronous communication between modules. It is patterned on the version in the CSP definition paper [Hoare 78]. Three nodes process a stream of characters in the following fashion. The first node (DISASSEMBLE) receives a sequence of arrays of 80 characters. It sends each character in an array one at a time to the second node (SQUASH), whose function is to examine the stream of characters and transmit it unchanged, except for removing all consecutive

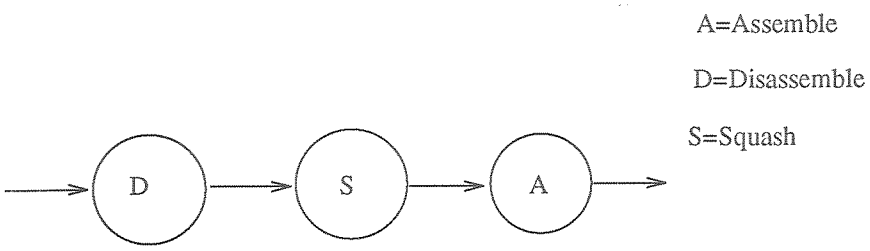

Figure 2.3: Computation graph for text processing pipeline

asterisks '**', and replacing them with a '@' ('@' is used here instead of the ↑ in the Conway and Hoare papers). The third node (ASSEMBLE) receives a stream of characters from SQUASH and packs them into arrays of 125 characters, which it passes to a lineprinter, or other external device. Figure 2.3 gives the structure of the computation graph for the pipeline.

The requirements of this algorithm are:

1. Message passing communication is used, since all communication is ordered and one-to-one, and is most natural to the visualization of the computation graph as a pipeline.
2. The SQUASH node must be able to decide whether or not to send a character between executions. When it has received one asterisk, it will not transmit the character until the next one is received and examined; otherwise, it will immediately transmit the character just received. This is the dynamic graph traversal characteristic described above. In

particular, SQUASH should not be required to send a dummy filler message when it is in decision mode between executions. This is precisely the reason this algorithm was chosen by Conway to demonstrate the usefulness of coroutines.

Parallel Simulation

Characteristics:

- dynamic computation graph creation
- message based communication
- dynamic computation graph traversal: message sources and destinations vary at runtime

Algorithm: The following situation is simulated. Three doctors share an office and a nurse. Any doctor can see any patient when he is free. A pool of 100 potential patients exists at any time - the number and identity of the currently sick patients varies randomly. A sick patient checks in with the nurse who either sends him to the first available doctor or queues him in a single FIFO queue. Doctors are also queued if no patients are currently available. A patient, when assigned a doctor, communicates his symptoms to the doctor, who diagnoses him, and sends back a prescription. The patient then goes on his way for a random amount of time, until he gets sick again. This continues indefinitely, and presumably, data can be collected as to average length of queues, wait time, etc., although this is not germane to evaluation of the parallel implementation. Figure 2.4 gives a possible snapshot of the

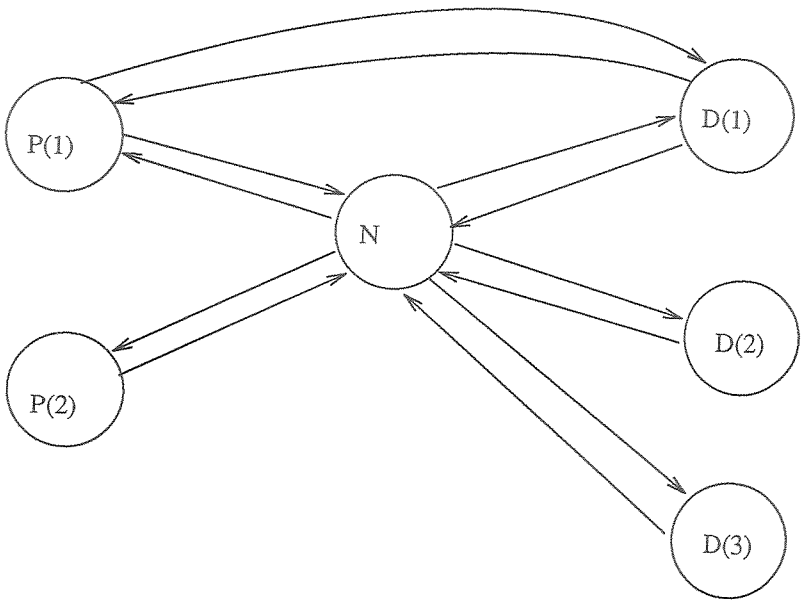

Figure 2.4: Snapshot of an execution of the parallel simulation

computation graph of the parallel simulation algorithm. Only two patients are currently ill. Patient(1) has been assigned to Doctor(1). Patient(2) has not yet received its assignment.

The requirements of the algorithm are:

1. Patient nodes should be created and destroyed dynamically. It is not essential for this formulation of the algorithm that a patient persist between visits to the doctor. This implies that the dependencies between the doctors and patients and between the nurse and the patients are also dynamic. The nurse and doctor nodes and their dependencies do persist throughout the simulation.
2. The fairness of the doctor and patient queues should be preserved. In this formulation of the algorithm, this entails a central bottleneck of

two queues, centered in the nurse node. Two shared queues could have been implemented, with each patient and doctor node following a predetermined protocol. In any case, fairness is essential to the algorithm.

3. Message based communication should be used between the nodes, and message sources and destinations should be dynamically based on the information supplied by the nurse. That is, no doctor or patient ever knows which patient or doctor he will be assigned to, until just before he communicates with the other node.

2.3 Classification of Parallel Languages

2.3.1 Characteristics of Parallel Languages

Here, only the aspects relating to parallelism specification and control are considered, leaving aside conventional sequential language characteristics such as data types and operations. A parallel language needs the following capabilities:

1. define and instantiate a set of nodes (sequential code segments of arbitrary granularity)
2. define their relationships, including data and synchronization dependencies and execution constraints
3. schedule the execution of the nodes according to their dependencies
4. control transfer of data between the name spaces of the various nodes. This may consist of both message transmission and shared data assignment.

A parallel programming language defines a set of virtual parallel machines. A program in a parallel language defines a computation graph constituting a particular virtual machine, and controls its traversal at execution time.

2.3.2 Classification Scheme

A general classification scheme for parallel languages is adopted, based on the criterion of implicit vs. explicit parallelism. The parallelism specification and control is either explicitly under the direction of the programmer, as in most imperative languages, or is implicit in the semantics of a higher-level declarative language, such as the functional dataflow languages. The explicit parallelism group can be subdivided into two classes based on the degree of abstraction of parallelism control from the computation of the algorithm.

This classification scheme is not fundamental, and it does not divide the set of parallel languages into disjoint subsets. It is useful, however, in two ways. It corresponds intuitively to how a programmer would lump languages together as being similar to use. It is also useful for the purpose of this thesis: to analyze the generality and limitations of parallel programming languages when applied to the given testbed of parallel algorithms.

The three resulting classes of parallel languages are as follows.

Parallelism-Extended Languages

The largest class of languages by far has this set of characteristics:

1. Parallelism is explicitly under the control of the programmer.
2. Node definition and computation graph topology are embedded in the sequential computation code.

These are standard sequential imperative languages with extensions for communication, synchronization, and node creation. Nodes are generally processes with persistent state, large-grain instances of procedures, and are defined by procedure-type definitions, and created and terminated by ordinary declarations and/or procedure call mechanisms. Communication may be either message-based (CSP and languages for distributed systems) or allow for shared memory as well (Ada and the class of concurrency control languages designed for operating systems). These languages range from standard sequential languages with low-level operating system calls added, such as concurrent FORTRANs and Cs, to the specially-developed, complex virtual machine defined by Ada.

Abstract Computation Graph Languages

These languages have the following characteristics:

1. Parallelism is explicitly programmed.
2. The topology of the computation graph is expressed separately from the code of its component nodes, in special control code.
3. The traversal of the computation graph may also be expressed separately from the computation of the component nodes.

These languages are special-purpose parallel programming languages, in that their main function is to express parallelism specification and control, and they may lack a wide variety of data types and abstraction mechanisms. The nodes are written in another language, and linked to the computation graph via the abstract parallel program. CSL, developed at the University of Texas [CSL 85], is a language that specifies both topology and traversal of a computation graph. Other languages may express only topology: examples are Parlance [Reynolds 79] and TASK [Schwans 82], two languages which define a computation graph, but leave the traversal of the graph to the execution of the nodes, which are written in parallelism-extended languages of the previous class.

Declarative Languages

These languages have the following characteristics:

1. The topology and traversal of the parallel computation are not explicitly programmed, although the parallelism-conscious programmer must certainly be aware of the underlying philosophy of the language's implementation.
2. Automated extraction of implicit parallelism is aided by the restriction of data types and operations, and sometimes by the addition of special parallel operations on data objects.

These languages include functional languages, such as SASL [Turner 83], and data-flow languages with functional semantics, such as SISAL [McGraw 85]. For the functional languages, the programmer defines the problem

in terms of evaluation of expressions. The definition and traversal of the parallel computation graph is automated by the compiler. The target machine is usually a special-purpose non-Von Neumann architecture. The granularity of the nodes is not under the control of the programmer, and is usually much smaller than that specified by parallel programs written in imperative languages, such as Ada. In functional and dataflow languages, all operations are executable as soon as their operands are available, thus, node definition and scheduling is implicit in the dependency of function compositions and parameter transmission. Communication is strictly through function results. The functions have no persistent state, and there is no global state of the computation graph, hence no shared memory in the form of global variables is allowed to the programmer, although it may be used to implement passing of operands.

Notes on the Classification Scheme

There is some fuzziness even in this very general distinction between languages. For instance, a parallelism-extended language can be programmed in a disciplined, hierarchical manner to emulate an abstract computation graph language, by abstracting all parallelism control to the outer level of the program. As noted, computation graph languages are sometimes associated with nodes written in parallelism-extended languages; in these cases the computation graph programs are reduced to aids in configuration control, loading and linking. All but the purest of higher-level languages have constructs that make the programmer think explicitly of parallelism control, for example, the cross and dot product FOR statement operations in SISAL. However, the

areas that have been delineated give a general idea of the approach that a programmer will take to programming a problem, given a language in that class.

2.3.3 Choice of Languages

A language has been chosen from each subclass above in which to code the test algorithms for comparison. The criterion for choosing each representative language was that it should be well-known and influential, and likely to be in use for some time to come. Each language will be sketched below, and more detailed descriptions of its particular parallelism-related constructs will be given in the appropriate chapters.

Two languages were chosen from the parallelism-extended class. Ada is the U.S. Department of Defense language for real-time process control. It is a very complex language that includes mechanisms for explicit parallelism specification and control, and for both message-based and shared memory communication. Occam [Pountain 86] is a language based on Hoare's CSP. It is a simple language using explicit parallelism specification and control, and a strictly message-based communication scheme. Ada and Occam were chosen for their differing models of communication, and to contrast the tradeoffs of simplicity and ease of use vs. complexity and a rich set of operations.

CSL is the language chosen for the abstract computation graph class. CSL stands for Computation Structure Language, and is based on the philosophy of separation of computation from parallelism control. Schedulable units of computation are coded in an ordinary sequential (not parallelism-

extended) language, such as Pascal. The topology and traversal of the computation graph are specified in the CSL program, and all association of data with nodes and synchronization of node execution is coordinated in CSL code.

SISAL has been chosen from the declarative language group. It is a functional (or applicative) dataflow language. Functions operate on their arguments and return values. Parallelism is extracted by the compiler. In general, arguments to function calls can be evaluated in parallel. Additionally, some explicitly parallel operations on array structures are provided.

2.4 Evaluation Criteria

Given a coding of a parallel algorithm in a particular parallel programming language, the following criteria will be considered to evaluate the successfulness of the translation.

2.4.1 Expressive Power

Any parallel programming language should have constructs that conveniently express implicitly or explicitly the structure of the computation graph, including the definition of nodes and their dependency relationships.

The parallel algorithm should be expressed completely with no unnecessary constraints. The purpose of parallelizing algorithms is to obtain maximum speedup, therefore the maximum parallelism inherent in the algorithm description should be exposed in the parallel program. An example of this is the block triangular solver, which has a fairly complex set of synchronization sequencing rules. A simpler, and computationally equivalent, version of the same algorithm would process the matrix one column at a

time. The result would be the same, but some permissible partial orders of execution of the schedulable units of computation would be excluded, and we wish to preserve the maximum amount of asynchronicity inherent in the algorithm. Therefore, the complex synchronization relationship between the nodes should be expressible in the language, whether it is explicitly specified by the programmer or extracted by the compiler.

A language should allow a model of communication that is most natural to the algorithm, or the algorithm will have to be transformed to meet the limitations of the model of communication that is available. It has been postulated that the shared memory and message models of communication are duals [Lauer 78], and that any problem expressible in one is expressible in the other. However, this may require that other aspects of the algorithm be changed, or that additional complexity be introduced to maintain the algorithm semantics. Again, an example is the triangular solver. When the algorithm is changed from a set of nodes operating on a shared data item to a set of nodes communicating by channels, some of the inherent parallelism may be lost if an arbitrary order of execution of the interior MATVECT rows is enforced before runtime. This is because channels are one to one and deterministic. If the shared vector blocks are encapsulated within monitors which communicate with the MATVECTs by channels, the nondeterminism of order of access is preserved but additional complexity is created, and duplicated transmission of data and control messages is introduced.

Another criterion for expressive power is that superfluous code, in definition of node and activation of arcs, not be necessary in order to maintain the semantics of the algorithm. Some method of dynamic feedback for

the creation and traversal of the computation graph should be available at execution time, because changing a dynamic algorithm to a static algorithm frequently introduces such computationally unimportant overhead. This may cause a reduction in efficiency, since all possible cases must be provided for at runtime, whether needed or not. Examples are:

- creating at compile time the maximum set of nodes that could possibly be needed for a computation graph, when only a subset will be needed at execution time
- executing unnecessary nodes which simply check that they are not needed and then terminate
- transmission of unnecessary data. This can occur either when it is necessary to inform a redundant node that it is not needed in order to force it to terminate, or when control of message transmission is separated from computation so that an intermediate state which would result in no data being transmitted cannot be checked for. This also would add extra code to the receiver, which must check for and throw away a dummy message.

2.4.2 Understandability

A parallel algorithm should be understandable given the piece of parallel code that embodies it. The computation graph topology and possible traversals should be recoverable from the code with as little effort as possible. Many of the comments about expressive power apply to understandability as well, since the original structure of an algorithm that has been transformed to fit the limitations of a language will not be as easily recoverable. The code

will have to be closely examined to see which dependencies are absolutely necessary to the correct execution of the algorithm, and which are artifacts of the translation. Understandability is necessary for maintenance even more for parallel codes than for sequential ones, especially where the codes are modules that will be fitted together into a complex system. Problems that adversely affect understandability include superfluous code and unnecessary constraints on execution order.

Another factor in understandability is the ability to specify a hierarchy of computation graphs. A whole computation graph may be abstracted to a single node in an encompassing graph. This allows a top-down development and analysis of large parallel systems, and is a fundamental property of computation graphs. A parallel language should ideally allow hierarchical composition of parallel algorithms. The algorithms given do not test this property, but we will examine it when discussing the example languages in general.

2.4.3 Ease of Use

This is a highly subjective metric, but the following points will provide a focus for discussion.

A parallel language that provides most of the mechanisms for expression of the inherent characteristics of parallel algorithms will be easier to use in the general case than a more limited language, since not as much energy will be spent on transforming algorithms. In some cases, an algorithm may be impossible to code in a given language, if it is totally lacking in some essential property of the algorithm. However, complex languages have some

drawbacks in ease of use, since it may take quite a while to learn them, and subtle errors may continually be introduced. A parallel language should provide a powerful but simple array of mechanisms for parallelism specification and control.

Another factor affecting ease of use is the level of the language constructs that are available to the programmer. High-level synchronization constructs in an explicit parallelism language can reduce substantially the amount of code necessary to express a problem correctly. Another example is the purpose of declarative languages, to provide the programmer with a high-level way of defining the required results, without the necessity of specifying the steps to obtain those results.

Hierarchical composition of graphs is also a factor in the ease of use of a parallel language, since of course simpler abstractions are easier to code than large, complex, single-level systems. This is similar to the same issue in conventional sequential languages.

2.5 Summary

The graph model of computation has been discussed, and the terms defined. This model is used to describe both algorithms and languages, since all three (model, algorithms, languages) define virtual parallel machines. Three characteristics of parallel algorithms have been enumerated, and four simple but non-trivial algorithms have been created to provide a mix of the characteristics to test the general-purpose qualities of a programming language. A subdivision of types of parallel programming languages has been given, and four languages have been chosen from the subclasses to code the

sample algorithms for purposes of comparison. Finally, several evaluation metrics have been given with which to judge the successfulness of the translation of the test algorithms into the representative languages.

2.6 Organization of the Following Chapters

Each language will be *briefly* described, with emphasis on parallelism specification and control constructs. The language reference manuals should be consulted for other details. In the section on expressive power, the codes will be introduced and evaluated. General discussions of the understandability and ease of use of the language will follow.

Chapter 3

Ada

3.1 Language Description

Ada is the language developed by the U.S. Department of Defense for real-time process control [DoD 83]. It has a very complex and rich set of data types and abstraction mechanisms, plus constructs related to real time programming. Here, we shall briefly survey those programming constructs related strictly to parallelism specification and control. See the Language Reference Manual (LRM) for further information.

The unit of parallelism is the task. A task is defined with similar syntax to the other modules of Ada, the package and procedure. Ordinary tasks "depend" on their declaring procedure or task, and are activated at the same time. The declaring procedure or task does not terminate until all of its dependent tasks have terminated. This corresponds to the Cobegin/Coend construct in other languages. Tasks can be dynamically created after the start of their declaring procedure or task by the use of access (pointer) types. Tasks may not make directly recursive calls.

Tasks may communicate either through shared variables or a message based mechanism called the rendezvous. Shared variables are declared using ordinary scoping rules. Synchronization of access to shared variables is not guaranteed, and must be programmed. A compiler pragma "shared" is

defined in the LRM as guaranteeing indivisible access to the variable between synchronization points by any task. This means basically that all local updates of the variable will be written to the shared copy before another task accesses it. This pragma is not implemented in the VAX Ada implementation, but instead a pragma "volatile" [Vax Ada 85] which guarantees that the variable will be accessed only from shared memory, and not placed in high-speed registers (which might not be equally accessible to all accessing tasks). In other words, except for simple read-only access to shared variables, and those few situations where mutual exclusion of readers and writers is unimportant, the user must program all synchronization details. Ada's rendezvous mechanism allows tasks to communicate through named ports called entries. The communication is synchronized - the sender and receiver "rendezvous" at the entry point. The rendezvous is a caller-server model. The caller must know the entry point of the server model, but the server need not know the identity of the caller. Multiple callers who attempt to access the same entry point of a server are automatically queued by the runtime system, so that their requests are not lost. The rendezvous may pass parameters both ways between the caller and server. The caller performs the rendezvous by naming the server module and the particular entry point desired. The server performs the rendezvous by executing an accept statement on the entry point.

Ada has a mechanism for nondeterminism, the select statement. The select statement allows the task to accept nondeterministically one of a set of rendezvous requests that are pending. The rendezvous alternatives within a select may be guarded by When statements that only allow the rendezvous to be selected if the condition following the When statement is true at the

time. This form of guarded nondeterministic branching is based on Dijkstra's guarded alternatives construct [Dijkstra 75].

Miscellaneous other constructs of Ada related to tasking are the ability to terminate and abort tasks, to propagate exceptions through the task calling structure, and to access certain attributes of tasks, such as whether it has terminated or failed, and its relative priority.

Ada is classified as a parallelism-extended language. All parallelism is explicit, strictly under the control of the programmer. The structure of the computation graph corresponds to the declarations and activations of tasks and the procedures upon which they depend, and to the sequence of rendezvous calls and shared variable accesses. This structure is embedded in the program text as the task declarations, the rendezvous statements, and the shared variable references.

3.2 Expressive Power

3.2.1 Block Triangular Solver

Ada solution

```
with text_io;
procedure trisolver is
--
use text_io;
package fl_io is new float_io(float);
use fl_io;
--
Amatrix : text_io.file_type;
bvector : text_io.file_type;
xvector : text_io.file_type;
--
numblocks : constant := 6; --dimension of block system
mvlimit : constant := 5;   --highest index of matvects
sizeblock : constant := 2; --blocking factor
```

```
--
type matrix is array (1..sizeblock, 1..sizeblock) of float;
type vector is array (1..sizeblock) of float;
type blockmatrix is array (1..numblocks, 1..numblocks) of matrix;
type blockvector is array (1..numblocks) of vector;
type service is (read, write);
--
A : blockmatrix;
b : blockvector;
pragma volatile (b);  -- informs Ada compiler that vector b
                      -- is shared between tasks
--
task type monitor is
   entry start (s : in service);
   entry stop_read;
   entry lock;
   entry release;
   entry finish;
end monitor;
--
monitor_b : array (1..numblocks) of monitor; -- one monitor for
                                             -- each block
--
task type solve is
   entry start (i : in integer);
   entry signal;
end solve;
--
solves : array (1..numblocks) of solve;
--
task type matvect is
   entry start (i, j : in integer);
   entry signal;
end matvect;
--
matvects : array (2..numblocks, 1..mvlimit) of matvect;
--
task body monitor is
   readers : integer := 0;
   writers : integer := 0;
begin
   loop
      select
         when writers = 0 =>
         accept start (s : in service) do
            case s is
               when read => readers := readers + 1;
               when write => writers := 1;
            end case;
         end start;
      or
```

```
         accept stop_read;
         readers := readers - 1;
      or
         when readers = 0 =>
         accept lock;
      or
         accept release;
         writers := 0;
      or
        accept finish;
        exit;
    end select;
end loop;
end monitor;
--
task body solve is
   sigcount, i, j, index : integer;
begin
   accept start (i : in integer) do
     index := i;    -- discover identity
   end start;
   for sigcount in 1..(index - 1) loop
      accept signal;  -- wait for matvects in this
                      -- row to terminate
   end loop;
   for j in 1..sizeblock loop
     b(index)(j) := b(index)(j) / A(index,index)(j,j);
     for i in (j + 1)..sizeblock loop
        b(index)(i) := b(index)(i) -
           (A(index,index)(i,j) * b(index)(j));
     end loop;
   end loop;
   for sigcount in (index + 1)..numblocks loop
      matvects (sigcount, index).signal;  -- signal matvects in
                                          -- this columnto start
   end loop;
   if index = numblocks -- if this is the final solve,
                        -- terminate the monitors
                        -- and print results
   then
   for i in 1..numblocks loop
      monitor_b(i).finish;
   end loop;
   create (xvector, out_file, ''xvector'');
   for i in 1..numblocks loop
      for j in 1..sizeblock loop
         put (xvector, b (i)(j) );
         new_line (xvector);
       end loop;
   end loop;
end if;
```

```
end solve;
--
task body matvect is
   i1, i2, k, l : integer;
begin
  accept start (i, j : in integer) do
     i1 := i;
     i2 := j;
  end start;
  if i2 < i1 then  -- if this is a block below diagonal
    accept signal;
    monitor_b(i1).start(write); --synchronize on vector
    monitor_b(i2).start(read);
    monitor_b(i1).lock;
    for k in 1..sizeblock loop
      for l in 1..sizeblock loop
         b(i1)(k) := b(i1)(k) -
                ( A(i1,i2)(k,l) * b(i2)(k));
      end loop;
    end loop;
   monitor_b(i2).stop_read; --release vector block
   monitor_b(i1).release;
   solves(i1).signal;
 end if;
end matvect;

begin
open (Amatrix, in_file, ''Amatrix'');
open (bvector, in_file, ''bvector'');
for i in 1..numblocks loop
  for k in 1..sizeblock loop
     for j in 1..numblocks loop
         for l in 1..sizeblock loop
             get (Amatrix, A(i,j)(k,l));
         end loop;
       end loop;
     end loop;
end loop;

for i in 1..numblocks loop
   for j in 1..sizeblock loop
     get(bvector, b(i)(j));
   end loop;
end loop;
-- send the tasks their indices
for i in 1..numblocks loop
   solves(i).start(i);
end loop;
for i in 2..numblocks loop
   for j in 1..mvlimit loop
      matvects(i,j).start(i,j);
```

```
   end loop;
end loop;

end trisolver;
```

Discussion of Ada solution

In general, the Ada solution manages to meet all the specifications of the original algorithm. The schedulable units of computation are the arrays of the task types SOLVE and MATVECT. The synchronization of the tasks depends on both the rendezvous mechanism and a user-programmed monitor, which encapsulates the shared vector b. The monitor task is adapted from the readers-writers example in [Barnes 82]. For the rendezvous mechanism to work properly here, each task must know its index in the matrix, therefore the main procedure transmits this information after it starts (it is the only actual computation that the main procedure does). Each task performs this self-identification rendezvous as its first act. Using this information, each SOLVE task knows which set of MATVECT tasks to notify of its completion, corresponding to the column beneath it. Each SOLVE task also knows how many MATVECT tasks must signal it before it can begin computation (it does not need to know their identities, since it is the receiver in the rendezvous, although it could calculate them from its own identity).

The vector is kept in a shared array, and a user-programmed monitor performs simple first-come first-serve queueing synchronization on it. This requires that each task follow the same discipline of rendezvousing with the monitor before accessing the variable, and of releasing its access after it is finished. The SOLVE tasks do not need to synchronize on b, since the

MATVECT tasks that write the same block will not begin actual computation until the SOLVE has finished. No two SOLVE tasks write the same block of b. The matrix is also kept in a partitioned shared variable, but no synchronization is required, since each task accesses a different partition.

Two criticisms of the Ada solution are mentioned here. One is that a lower triangular structure of MATVECT tasks is not declarable, since a standard matrix declaration format is used, and the indices may not refer to each other. The 'upper half' tasks terminate as soon as they receive their indices. The other criticism is that a substantial amount of code is introduced by the necessity of programming the task type MONITOR to control the access to the shared data. This code will be virtually identical in all similar situations. It does not seem unreasonable in such a rich language to expect some more high-level shared memory synchronization constructs.

3.2.2 Parallel Quicksort

Ada solution

```
with text_io;
procedure quick is
use text_io;
package int_io is new integer_io (integer);
use int_io;

type intarray is array (integer range <>) of integer;
sortarray : intarray (1..10); --array of integers to sort
pragma volatile (sortarray); --informs runtime system that
--sortarray is shared between tasks

-- procedure call_son and task quick_sort are mutually recursive,
-- so both specifications are placed before both bodies, as a
-- forward declaration

procedure call_son (lo, hi, mid : in integer);

task type quick_sort is
```

```
     entry Partition (L, H : integer);
end quick_sort;

-- the following procedure reads in the unsorted array
procedure get_array (sortarray : out intarray) is
    in_sortarray : file_type;
    i : integer;
begin
    open (In_sortarray, in_file, ''unsorted'');
    for i in 1..10
        loop
           get (In_sortarray, sortarray(I));
        end loop;
end get_array;

-- the following procedure checks the size of two partitions,
-- calling a new quicksort on each one,
-- if they are of size > 1
procedure call_son (lo, hi, mid : in integer) is
  type sibtype is access quick_sort;  -- pointer to a task
  sib2, sib1 : sibtype;
begin
if lo < (mid - 1)
   then sib1 := new quick_sort;
        sib1.partition (lo, mid);
end if;
if (mid + 1) < hi
   then sib2 := new quick_sort;
        sib2.partition (mid + 1, hi);
end if;
end call_son;

-- the following task type performs the quicksort partitioning
-- algorithm on a slice of the unsorted array
task body quick_sort is
  lo, hi, mid, i, j, temp : integer;
begin
  accept Partition (l, h : integer) do
      lo := l;
      hi := h;
  end Partition;
  if hi > lo then
     i := lo;
     j := hi;
     mid := lo;
     while i <= j loop
       if (sortarray(i) > sortarray(j))   -- do swap
         then
            temp := sortarray(i);
            sortarray(i) := sortarray(j);
            sortarray(j) := temp;
```

```
            if mid = i                          -- move pointer
               then mid := j;
               else mid := i;
            end if;
       end if;
       if i = mid
          then j := j - 1;
          else i := i + 1;
       end if;
     end loop;
     call_son (lo, hi, mid);
  end if;
end quick_sort;

-- the following procedure starts the sorting
procedure do_sort is
  first_pass : quick_sort;
begin
first_pass.partition (sortarray'first, sortarray'last);
end do_sort;

begin            -- main module
  get_array (sortarray);
  do_sort;
  for i in sortarray'first..sortarray'last
     loop
        put(sortarray(i));  -- print sorted array to terminal
     end loop;
end quick;
```

Discussion of Ada solution

This solution manages to express the problem fairly well. Direct recursion of tasks is not allowed in Ada, however, which introduces some redundant coding here: a procedure must call a task which calls the same procedure. A task is not allowed either to call its own type by recursion, or to create new instances of its type with access pointers. The amount of code is increased to get around this peculiarity of the language, which arises from a distinction between procedures and tasks as objects.

The Ada solution does allow the use of shared memory, however, and

in this case no extra monitor task need be coded, since no mutual exclusion is required on the subsets of data due to the semantics of the task creations.

Only those tasks which are needed are created dynamically, due to the mechanism of access pointers. Since the computation and parallelism control are integrated, it is possible to check the results of the partitioning before calling new partitioning tasks, avoiding waste creation of tasks. The binding of tasks to their array segments is effected by the rendezvous mechanism. Each task is sent the size and starting position of its array segment before it begins partitioning.

Although tasks are created dynamically, the tree structure of the calling mechanism means that the intermediate tasks cannot terminate until all of their dependent tasks have finished. In other words, all the tasks remain active until the array is completely sorted. This is wasteful here, since they need do nothing further.

3.2.3 Text Processing Pipeline

Ada solution

```
with text_io;
procedure pipeline is
use text_io;
   cardreader : text_io.file_type;     -- input file
   lineprinter : text_io.file_type; -- output file
   endchar : constant character := '#'; -- flags end of input
   line : string (1..80); -- input line image
   lim : integer;
   i,j : integer;

task assemble is
   entry pass (ch : character);
end assemble;
task body assemble is
   save : character;
   i,j : integer;
```

```
   line : string (1..125); -- output line image
begin
   create (lineprinter, out_file, ''lineprinter'');
   i := 1;
   loop
      accept pass (ch : character) do
        save := ch;
      end pass;
      if save = endchar then -- signals end of input stream
         for j in i..125
         loop
              line(j) := ' '; -- pad last line with blanks
        end loop;
        put_line (lineprinter, line); -- print last line
        exit; -- exit loop
      end if;
   line(i) := save;
   i := i + 1;
   if i > 125 then
     put_line (lineprinter, line); -- print line when full
     i := 1;
   end if;
 end loop;
close (lineprinter);
end assemble;

task squash is
  entry pass (ch : in character);
end squash;
task body squash is
   cleared : boolean; -- flags empty stages
-- (squash has two stages of the pipeline; when a double
-- asterisk is "squashed", two characters are removed,
-- and both stages are empty)
   first, second,  -- two stages of pipeline
   save : character;
begin
   cleared := true; -- stages initially empty
   loop
      accept pass (ch : in character) do
         save := ch;
      end pass;
      if cleared then
          cleared := false;
          first := save;
      else
         second := save;
         if (first /= '*') or (first /= second) then
             assemble.pass(first) -- pass character if no '**'
             first := second;
             if first = endchar
```

```
            then assemble.pass(first); -- pass end signal w/o
            end if;                    -- further processing
         else
            assemble.pass('@')         -- pass '@' if '**'
            cleared := true;           -- stages are empty
         end if;
      end if;
      exit when save = endchar; -- exit loop at end of stream
   end loop;
end squash;


begin
open (cardreader, in_file, ''cardreader'');
  loop
   if end_of_file (cardreader) then
      squash.pass (endchar); -- pass end of stream signal
      exit;
    else
      get_line (cardreader, line, lim);
      for i in 1..lim
        loop
          squash.pass (line(i)); -- unpack card image
        end loop;
      j := lim + 1;
         for i in j..81
           loop
              squash.pass (' '); -- pad image with blanks
           end loop;
    end if;
   end loop;
  close (cardreader);
end transfer;
```

Discussion of Ada Solution

The solution is quite efficient. An interesting note is that parallelism of three units is achieved, with two of them being tasks that are dependent on the third, which is a procedure. This is because of the Cobegin semantics of tasks declared in procedures mentioned above. The rendezvous mechanism is used for message-passing as required. The central task SQUASH is able to delay message transmission while deciding whether to suppress the double

asterisks. No dummy messages need be sent, again because of the close association of the computation and communication code.

3.2.4 Parallel Simulation

Ada Solution

```
procedure simulation is
type message is string(1..100);
i, j : integer;

task type patient is
   entry assign (doctornum : in integer);
   entry diagnose (rx : in message);
end patient;
type patient_ptr is access patient;
patient_roster: array (1..100) of patient_ptr;

task type doctor is
    entry complain (symptoms : in message;
                    patientnum : in integer);
    entry getid (doctornum : in integer);
end doctor;
partners : array(1..3) of doctor;

task body patient is
pn,dn : integer;
symptom, prescription : message;
begin
   accept getid (patientnum : in integer) do
         pn := patientnum;
   end getid;
   nurse.checkin (pn);
   accept assign (doctornum : in integer) do
         dn := doctornum;
   end assign;
   symptom := ''complain'';
   partners(dn).complain (symptom,pn);
   accept diagnose (rx : in message) do
         prescription := rx;
   end diagnose;
end patient;

task nurse is
   entry checkin (patientnum : in integer);
   entry next (doctornum : in integer);
end nurse;
```

```
task body nurse is
i, firstd, currd, firstp, currp : integer;
patientq : array(1..100) of integer;
doctorq : array(1..100) of integer;
begin
--initialize queues
firstd := 1;
firstp := 1;
currp := 100;
currd := 3;
for i in 1..100 loop
  patientq(i) := -1;
end loop;
for i in 1..3 loop
  doctorq(i) := -1;
end loop;
--match up doctors and patients as they enter
loop
select
   accept checkin (patientnum : in integer) do
   exit when patientnum = 101;
   if doctorq(firstd) /= -1 then --doctor is available
       patient_roster(patientnum).assign(firstd);
       doctorq(firstd) := -1; --dequeue doctor
       if firstd = 3 then
         firstd := -1;
       else
         firstd := firstd + 1;
       end if;
   else  --no doctor, queue patient
     if currp = 100 then
        currp := 1;
        patientq(currp) := patientnum;
     else
        currp := currp + 1;
        patientq(currp) := patientnum;
     end if;
   end if;
   end checkin;
or
   accept next (doctornum : in integer) do
     if patientq(firstp) /= -1 then  -- patient waiting
       patient_roster(firstp).assign(doctornum);
       patientq(firstp) := -1; -- dequeue patient
       if firstp = 100 then
          firstp := 1;
       else
          firstp := firstp + 1;
       end if;
     else  -- no patients, queue doctor
       if currd = 3 then
```

```
          currd := 1;
          doctorq(currd) := doctornum;
        else
          currd := currd + 1;
          doctorq(currd) := doctornum;
        end if;
      end if;
    end next;
end select;
end loop;
end nurse;


task body doctor is
dn, pn : integer;
s, rx :  message;
begin
rx := ''twoaspirin'';
accept getid (doctornum : in integer) do
    dn := doctornum;
end getid;
loop
    nurse.next (dn);
    accept complain (symptoms : in message;
                     patientnum : in integer) do
           s := symptoms;
           pn := patientnum;
    end complain;
    exit when pn = 101;
    patient_roster(pn).diagnose(rx);
end loop;
end doctor;

begin
for j in 1..1000
loop
   i := (j mod 100) + 1;
   if (patient_roster(i) = null) or
      (patient_roster(i).all'terminated) then
--only activate if not already activated
   patient_roster(i) := new patient;
   patient_roster(i).getid(i);
end loop;
--terminate nurse and doctors
nurse.checkin(101);
for i in 1..3 loop
partners(i).complain(''', 101);
end loop;
end simulation;
```

Discussion of Ada Solution

This problem is also coded quite neatly in Ada. The rendezvous mechanism provides the message passing required. Patient tasks are created dynamically, using access pointers. Since the patient tasks have no dependent tasks, they are also terminated dynamically, which is exactly what is required. The nurse and doctor tasks persist throughout the simulation, and are only terminated upon receiving a special signal. The dynamic communication between doctors and patients is achieved by having the nurse match them up, and transmit the appropriate indices into the arrays to each one. The patient can use the index to rendezvous with the assigned doctor, and the doctor similarly knows where to return the diagnosis. Each pair of nodes could have been synchronized throughout their interactions by use of the two-way parameter passing mechanism of the rendezvous, but it was decided to emphasize this by using two symmetric rendezvous's instead.

3.2.5 Conclusions

Ada passes the test of expressive power well. It allows both models of communication. The complex synchronization of the triangular solver problem is easily expressible with the rendezvous mechanism used as a signalling device. Pointers to task types prove very useful for dynamic task creation. Ada's main drawbacks are the lack of a useful high-level synchronization device for shared variables, which causes the programmer to write extra low-level synchronizing code in some cases, and the lack of a simple parallel recursion mechanism. A generic monitor package might serve well as the shared variable synchronization mechanism.

3.3 Understandability

The next issue is recovering the algorithm's structure, given the code it is expressed in. Two of the algorithms given have very simple structures: the quicksort and the pipeline. The nodes of the computation graph are easily identifiable from the task declarations. However, the asymmetric rendezvous helps to obscure the pattern of interactions of these nodes. In the two simply structured algorithms, this is easy to overcome. In the block triangular solver, which involves two-dimensional interactions between MATVECTs and SOLVEs, the two task types must be considered together to reconstruct their relationships, since each acts as receiver and sender to the other at different points. In the parallel simulation example, three task types have two-way interactions. For example, in the patient nodes, one accept statement rendezvous's only with the nurse, and the other rendezvous's with one of the doctors. This is not obvious from the code, and the two other task types must be examined to see that each task only calls one entry. In a system with only a few task types, and symmetric interconnections, the structure is not really difficult to understand; however, it is easy to imagine large, arbitrarily connected systems whose structure would become ever more difficult to recapture from the code. To understand Ada task systems, one must mentally abstract the parallelism-related statements from each task body.

The issue of hierarchicality in Ada is solved by its uniform use of abstraction mechanisms and Pascal-like scoping. An entire parallel computation graph can be embedded in a procedure or package, and linked to a larger graph through a procedure call or similar mechanism.

3.4 Ease of Use

Ada is not particularly easy to learn to use, although its dynamic task creation and two models of communication made it easier to program the example problems, than with the other languages. Part of the difficulty of using Ada is learning the multiplicity of rules about the mechanisms available for parallelism control. Although tasks are in many ways syntactically similar to procedures, they are semantically different in the lack of recursion as a form of dynamic task creation. Another factor is that task creation and termination are usually implicit in the structure of the program text. Only when a task is allocated by a pointer, is it dynamically created, and only when it has no dependent tasks of its own, can it be terminated. However, in the long run, the effort in understanding these implicit, and often not very clearly documented, concepts is paid off by the ability to express a wide range of algorithms.

3.5 Summary

Ada meets most of the criteria for general-purpose expressiveness: static and dynamic graph creation and traversal, and two modes of communication. It lacks a sufficiently high-level construct for shared memory synchronization, leaving all the work to the programmer every time. Ada managed to solve each of the example problems as required by their description, with extra code generated only when a monitor was explicitly programmed for the triangular solver, and when indirect recursion was implemented with a procedure and a task.

The nature of Ada as a parallelism extended language, with sequen-

tial computation mixed in with parallelism control allows dynamic graph traversal, but can obscure the parallel pattern, making the code difficult to analyze. The one-way naming of the rendezvous mechanism, although useful in the caller-server model, also helps to obscure interactions between tasks.

Chapter 4

Occam

4.1 Language Description

Occam is a message-based parallel language developed from CSP [Hoare 78]. The algorithms have been coded in Occam 2, to be released in 1986 [Pountain 86]. The language is conceptually very simple. There are three basic types of statements: assignment, send and receive. The send and receive statements are denoted by '!' and '?', respectively. Processes are composed out of these basic statements with a set of constructors. The constructors include: SEQ, for sequential composition; PAR for parallel composition (with cobegin semantics); ALT for nondeterministic selection on channels (Dijkstra's guarded alternative implemented); IF for deterministic branching; and WHILE for looping. Composition and hierarchy of statements are indicated by indentation in the program text. There is a replication construct which allows compact specification of parallel, alternative, sequential, and conditional processes. Processes may be named as procedures, and substituted in other places of the text. Procedure processes may take parameters, including channels. These processes are specified not to allow recursion, however.

Occam allows only message-based communication of a completely synchronous type. Two processes executing in parallel that wish to communicate must synchronize at their respective send or receive statement, and

execute the transmission simultaneously. Message transmission takes place through named channels, which are one-to-one and unidirectional.

Occam 2 has been extended over the original definition of Occam in several ways, including the addition of data typing and multidimensional arrays. It also requires a rather more complex notation for channel declarations. Each channel must have an associated protocol that defines the number and types of values that may be sent over it at one time. A single channel may have several alternative protocols, and a tag differentiates between them. From here on, the name Occam will refer to the current definition of Occam 2.

There are other constructs in Occam more related to real-time programming and configuration control than to abstract parallelism specification and control. These include the data type timer (used in the parallel simulation algorithm below in place of a random number generator), and provisions for placing processes on processors, and associating processes and channels with other hardware devices. Occam is being developed in conjunction with the transputer, which uses Occam as its assembly language. However, Occam has more general potential, which we will examine.

Occam is classified as a parallelism-extended language, although it was developed primarily for research into parallelism, and its constructs for other types of computation are rather limited in comparison to Ada. This classification is because the basic form of Occam is similar to Ada: schedulable units of computation are composed by the programmer, and the communication and synchronization are embedded in the sequential code of the processes. Occam was chosen in contrast to Ada because its only model of

communication is the message-based one.

One note on the Occam syntax in some of the example algorithms is in order. Since no Occam 2 translator was available at the time of this writing, it was not possible to check the acceptability of the program syntax. Although Occam is simple enough to justify confidence in the codes, in one case it is not clear that the syntax as written would have the desired meaning. This is when applying the array slice designator "FROM x FOR x" to a two-dimensional array. When this designator is applied twice to a two-dimensional array, it is meant to apply to the two different indices. In the Occam syntax definition, it appears that this might be applying the slice designator to the same index twice. If this is the case, the proper array slices can be designated by the more space-consuming, but equivalent, nested FOR construction.

4.2 Expressive Power

4.2.1 Block Triangular Solver

Occam solution

```
PROC solve (CHAN input, left, result, down, INT bd)
  [100] [100] INT a:
  [100] INT b:
  INT  i, j, temp:
  SEQ
    input ? [[a FROM 0 FOR bd]  FROM 0 FOR bd]
    left ? [b FROM 0 FOR bd]
    SEQ j = 0 FOR bd
      temp := b[j] / a[j] [j]
      b[j] := temp
      SEQ i = (j+1) FOR (bd - j)
        b[i] := b[i] - (a[i] [j]*temp)
          -- solve for block x[i] (in place in block b[i])
    PAR
      down ! [b FROM 0 FOR bd]  --transmit result to matvect
      result ! [b FROM 0 FOR bd] --transmit result to collector
:
```

```
PROC matvect (CHAN input, up, left, right, down,
              INT rownum,nblocks, bd)
  [100] [100] INT a:
  [100] INT b1, b2:
  INT i,j,temp:
  SEQ
    input ? [[a FROM 0 FOR bd] FROM 0 FOR bd]
--read in matrix block
    up ? [b1 FROM 0 FOR bd]
--read in vector block b[i]
    IF
      rownum < nblocks - 1
        down ! [b1 FROM 0 FOR bd]
--transmit vector block b[i] (unchanged)
      TRUE
        SKIP
    left ? [b2 FROM 0 FOR bd]
--read in vector block b[j]
    SEQ j = 0 FOR bd
      temp := b1[j]
      SEQ i = 0 FOR bd
        b2[i] := b2[i] - (a[i] [j] * temp)
-- compute results in place in second vector block
    right ! [b2 FROM 0 FOR bd]
-- transmit b[j] to next process in row
:
CHAN (INT; INT) input:
INT bd, nblocks,i,j,size:
[100] [100] INT a:
[100] INT b, x:
SEQ
  input ? bd, nblocks
-- read block dimension and blocking factor
  size := bd * nblocks
  CHAN ([size] [size] INT; [size] INT) getdata:
  CHAN ([size] INT) output:
  IF
    size <= 0
      SKIP
    size > 0
      SEQ
        getdata ? [[a FROM 0 FOR size] FROM 0 FOR size] ;
                  [b FROM 0 FOR size]
        [nblocks] [nblocks] CHAN ([bd] [bd] INT) in:
        [nblocks + 1] [nblocks + 1] CHAN ([bd] INT) left, up:
        SEQ
          PAR
            solve(in[1][1], in[1][1], left[1][2], up[2][1], bd)
            PAR i = 1 FOR nblocks
-- initiate synchronized tasks
              solve (in[i] [i], left[i] [i] ,left[i] [i+1],
```

```
                    up[i+1] [i], bd)
            PAR i = 1 FOR (nblocks - 1)
              PAR j = 0 FOR i
                matvect (in[i] [j], up[i] [j], left[i] [j],
                         left[i] [j+1], up[i+1] [j], i, nblocks,
bd)
            PAR i = 0 FOR nblocks
              PAR j = 0 FOR i
                in [i] [j] ! [[a FROM (i*bd) FOR bd] FROM (j*bd)
FOR bd]
            PAR i = 0 FOR nblocks
              left[i] [0] ! [b FROM (i*bd) FOR bd]
--transmit vector blocks to first column
            SEQ i = [0 FOR nblocks]
              left[i] [i+1] ? [x FROM (i*bd) FOR bd]
-- read in result vector blocks in order
          output ! [x FROM 0 FOR size]
-- output entire result vector
```

Discussion of Occam solution

Since Occam has no shared memory, it is necessary to change the algorithm to fit the semantics of channel communication. The same set of parallel processes as for the Ada solution is declared, using the named procedures SOLVE and MATVECT. Each process must read the necessary blocks of data off the appropriate channels. The channels named input and output are assumed to be special channels connected to external devices. The matrix blocks are read from an external input channel. Each SOLVE process receives its block of the vector from its left channel. By parameterizing the channels, SOLVE(1,1) also receives a copy of the vector block from the first input channel; therefore it begins execution immediately. Upon completing execution, a SOLVE transmits the result block of the vector to a collector channel, which is read by the main process. It also transmits the result block to the MATVECT directly below it. The MATVECTs in a column cooperate by transmitting the block to each other down the column before beginning

computation. This ensures that some parallelism will occur, even though the start of execution will be staggered by transmission time. It also eliminates the need for a variable number of channels from each SOLVE to each MATVECT in its column. The MATVECTs in a row are serialized from left to right since each must finish computation, then transmit the result vector block to the right.

The transformation of the algorithm that was chosen corresponds to a systolic array of processes that pump the vector blocks through, both down and across. Another version could have been programmed that encapsulated each block of the vector in a monitor. This solution in Occam however would have been longer, and more difficult to understand. It would have had more potential parallelism at the expense of extra processes and data transmission.

Occam's replication designators make it very easy to specify a lower triangular matrix of processes, and also to specify any chunk of a vector or matrix. The parameterized arrays of channels also made it easy to compose the matrix of processes.

4.2.2 Parallel Quicksort

Occam solution

```
PROC partition (CHAN fatherinput, lsonoutput, rsonoutput
                    fatheroutput, lsoninput, rsoninput)
  INT x, size, size1, size2, mid, i,j, temp, maxlevels,
      valid, level:
  [10] INT VALS:
-- local variables:
-- x : index of array vals;
-- size: length of array segment;
-- size1,size2: lengths of resulting partitions;
-- vals: array of values to be sorted;
-- maxlevels: maximum depth of this sort tree;
-- level: depth of this sort node;
```

```
-- i,j,mid,temp : indexes into array for partitioning
  SEQ
    fatherinput ? maxlevels; level; size; [vals FROM 0 FOR size]
    IF
      (size = 0) AND (level = maxlevels)
        SKIP
      (size = 0) AND (level < maxlevels)
        PAR
          rsonoutput ! maxlevels; (level + 1); 0;
                       [vals FROM 0 FOR 0]
          lsonoutput ! maxlevels; (level + 1); 0 ;
                       [vals FROM 0 FOR 0]
      (size >= 1)
        IF
          size = 1   -- return ``sorted'' element back up tree
            SEQ
              fatheroutput ! size; [vals FROM 0 FOR size]
              IF
                level < maxlevels
                  PAR
                    lsonoutput ! maxlevels; (level + 1); 0;
                                 [vals FROM 0 FOR 0]
                    rsonoutput ! maxlevels; (level + 1); 0;
                                 [vals FROM 0 FOR 0]
          size > 1 -- partition array, send and collect results
            SEQ
              i := 0
              j := size - 1
              mid := 0
              WHILE (i <= j)
                SEQ
                  IF
                    vals[i]  > vals[j]
-- do swapping if necessary
                      SEQ
                        temp := vals[i]
                        vals[i] := vals[j]
                        vals[j] := temp
                        IF
                          mid = 1
-- if swapped, then move dividing point
                            mid := j
                  IF
-- move index not pointing to dividing value
                    mid = i
                      j := j - 1
                    mid = j
                      i := i + 1
              PAR
-- transmit in parallel the results of the partitioning
                SEQ
```

```
                size1 := mid + 1      -- always nonzero
                lsonoutput ! maxlevels; (level + 1); size1;
                             [vals FROM 0 FOR size1]
              SEQ
                size2 := size - mid - 1  -- may be zero
                rsonoutput ! maxlevels; (level + 1); size2;
                             [vals FROM (mid + 1) FOR size2]
            PAR -- receive in parallel the results of the
                -- subsorting
              lsoninput ? size1; [vals FROM 0 FOR size1]
              rsoninput ? size2; [vals FROM (mid+1) FOR size2]
              fatherinput ! [vals FROM 0 FOR size]
:
CHAN ([10] INT) input, output:
[10] INT vals:
INT i,j,size,maxlevels:    -- main process
-- note: vals has index values 0 to 9
SEQ
  input ? [vals FROM 0 FOR 10]
  size := 10
  maxlevels := size - 1
  [1023] CHAN ([INT] INT) up:
  [1023] CHAN (INT; INT; [INT] INT) down:
  PAR i = 1 FOR 511
--maximum complete binary tree
    partition (down[i], down[2*i], down[(2*i)+1],
               up[i], up[2*i], up[(2*i)+1])
-- the following processes feed and collect data
    down[1] ! maxlevels; 1; size; [vals FROM 0 FOR size]
    up[1] ? size; [vals FROM 0 FOR size]
-- array vals is now sorted
  output ! [vals FROM 0 FOR size]
```

Discussion of Occam solution

The lack of shared memory and of recursion, or other dynamic parallel process creation, made this algorithm a candidate for transformation, as with the triangular solver. A quicksort computation graph can be thought of as a partial traversal of a complete binary tree, visiting only some of the nodes for each particular execution. A static process, channel communication version of quicksort can be formulated by creating the complete binary tree, and then passing the array partitions as required by the algorithm. In this

case, the same number of nodes of the tree will do active work, as in the dynamic graph version, and the rest will wait until they are notified that they are not needed. Since there is no shared memory, one way of collecting the sorted array together is to pass it back up the tree to the root node.

Occam's flexible declarations and compositions of processes and channels make this deformation of the algorithm easy to code. The undesirable aspect is the waste creation of processes, which cannot be avoided, since any one of them might be needed for any execution. Another waste aspect is the duplication and transmission of data, doubly so, since results are passed back up the tree. However, the array is successfully sorted in this case, and the maximum parallelism is allowed, as well.

4.2.3 Text Processing Pipeline

Occam solution

```
PROC disassemble (CHAN input, charstream)
  WHILE TRUE
    [80] BYTE line:
    INT i:
    SEQ
      input ? line
      i := 0
      WHILE i <= 79
        charstream ! line[i]
      charstream ! ' '
:
PROC squash (CHAN raw, processed)
  INT first, second:
  SEQ
    raw ? first
    WHILE TRUE
      SEQ
        raw ? second
        IF
          (first <> '*')  OR (first <> second)
          SEQ
            processed ! first
```

```
            first := second
          (first = '*')
            charstream ! '@'
            raw ? first
:
PROC assemble (charstream, output)
  WHILE TRUE
    [125] BYTE line:
    INT i:
    SEQ
      i := 0
      WHILE i <= 124
        SEQ
          charstream ? line[i]
          i := i + 1
          output ! line
:
CHAN input ([80] BYTE), tosquash (BYTE), fromsquash (BYTE),
     output ([125] BYTE):
  PAR  -- main process
    assemble (input, tosquash)
    squash (tosquash, fromsquash)
    disassemble (fromsquash, output)
```

Discussion of Occam solution

This algorithm is trivial to code in Occam, since it already is the sort of static, message-passing algorithm into which it was necessary to translate the first two algorithms. Since Occam communication and computation are embedded, the SQUASH process can change and suppress data at its discretion, and no extra messages are sent.

4.2.4 Parallel Simulation

Occam solution

```
PROC patient (CHAN tonurse, fromnurse,
              [3] CHAN todoctor, fromdoctor,
              INT self)
  INT dnum, rx, timenow:
  TIMER clock:
```

```
  WHILE TRUE
    SEQ
      clock ? timenow
      clock ? AFTER timenow PLUS 1000  -- period of health
      tonurse ! self
      fromnurse ? dnum
      todoctor [dnum] ! self
      fromdoctor ? rx
:
PROC nurse ([100] CHAN frompatient, topatient,
            [3] CHAN todoctor, fromdoctor)
  [100] INT patientq:
  INT nextslot, i, firstp, currp, firstd, currd, dnum, pnum:
  [3] INT doctorq:
  SEQ
    SEQ i = 0 FOR 100  -- initializations of queues
      patientq [i] := -1
    SEQ i = 0 FOR 3
      doctorq [i] := -1
    firstp := 0
    currp := 99
    firstd := 0
    currd := 2
-- Each time through loop, nurse accepts a message from
-- a patient or from a doctor.
-- If a matchup is possible it is dispatched at once,
-- else the patient or the doctor is queued.
    WHILE TRUE
      ALT
        ALT i = 0 FOR 100
          frompatient [i] ? pnum
            SEQ
              IF
               doctorq [firstd] <> -1
-- a doctor is ready and waiting
                  SEQ
                    topatient[i] ! doctorq [firstd]
                    todoctor[doctorq[firstd]] ! pnum
                    doctorq[firstd] := -1
                    IF
                      firstd = 2
                        firstd := 0
                      firstd < 2
                        firstd := firstd + 1
                currp = 99
-- all doctors busy, queue patient
                  SEQ
                    currp := 0
                    patientq[currp] := pnum
                currp < 99
                  SEQ
```

```
                    currp := currp + 1
                    patientq[currp] := pnum
        ALT i = 0 FOR 3
          fromdoctor [i] ? dnum
            SEQ
              IF
                patientq[firstp] <> -1
-- patient ready and waiting
                SEQ
                  todoctor[i] ! patientq[firstp]
                  topatient[patientq[firstp]] ! i
                  patientq[firstp] := -1
                  IF
                    firstp = 99
                      firstp := 0
                    firstp < 99
                      firstp := firstp + 1
                    currd = 2
-- no patients in room, queue doctor
                      SEQ
                        currd := 0
                        doctorq [currd] := dnum
                    currd < 2
                      SEQ
                        currd := currd + 1
                        doctorq [currd] := dnum
:
PROC doctor ([100] CHAN topatient, frompatient,
             CHAN tonurse, fromnurse, self)
  INT i, pnum, symptoms, rx:
  SEQ
    rx := self
    tonurse ! self
    WHILE TRUE
      SEQ
        fromnurse ? pnum
        frompatient[pnum] ? symptoms
        topatient[pnum] ! rx
        tonurse ! self
:
[100] [3] CHAN (INT) ptodoc, doctop:
[100] CHAN (INT) ptonurse, nursetop:
[3] CHAN (INT) nursetodoc, doctonurse:
INT i:
PAR
  PAR i = 0 FOR 100
    patient (tonurse[i], fromnurse[i], ptodoc[i], doctop [i], i)
--ptodoc[i] and doctop[i] are each an array of 3 channels
  nurse (nursetop, ptonurse, nursetodoc, doctonurse)
--nursetop and ptonurse are arrays of 100 channels,
--nursetodoc and doctonurse are arrays of 3 channels
```

```
  PAR i = 0 FOR 3
    doctor (nursetodoc[i], doctonurse[i],
            [ptodoc FROM 0 FOR 100] [i],
            [doctop FROM 0 FOR 100] [i])
--nursetodoc[i] and doctonurse[i] are each a single channel
--ptodoc, etc. and doctop, etc. are vertical slices,
--arrays of 100 channels
```

Discussion of Occam solution

Occam manages to capture the problem rather well, except for the lack of dynamic process and channel creation. Since an upper level of 100 is given on the number of patients, an array of patient processes of that size is created. These patients remain active throughout the simulation, timing themselves through periods of health before reporting to the nurse. The nurse process encapsulates the queues for doctors and patients, so fairness is preserved. Occam's message-based channels fit the form of the algorithm, and since the contents of messages received are available to affect the choice of the next channel upon which to send, the dynamic nature of the communication is easily captured.

4.2.5 Conclusions

Occam's optimal expressive power is limited to those algorithms with static computation graph creation, and message-passing characteristics. Shared memory semantics can always be obtained at the expense of extra monitor processes. Occam's integration of control and computation make dynamic computation graph traversal possible, and in some cases, such as the quicksort algorithm, allows a dynamic graph algorithm to be successfully transformed to a static graph algorithm. This will usually be at the expense

of extra processes.

There seems no particular reason, except the current limitations of the translator, for Occam not to have recursive processes. This would introduce dynamic process creation in some cases, and would make the quicksort with message passing much more palatable as an alternative, by cutting down on process waste.

4.3 Understandability

Ada and Occam are similar in that the parallel topology of an algorithm must be extracted from the surrounding code. It is somewhat easier to extract relationships between processes from an Occam program because communication is through named channels, which must be specified by both receiving and sending processes, while the corresponding endpoint of an Ada rendezvous is only named by the caller. Interestingly, this property of Ada's is considered to be more general than the two-way naming of CSP (Occam's close relative) in [Wegner 83], so there is a possible tradeoff between power of expression and understandability here.

4.4 Ease of Use

Occam has a smaller set of constructs with simpler semantics than Ada, and this makes it easier to learn to write Occam programs. The difficulty in using Occam comes in having to recast algorithms to fit Occam's static, message-passing virtual machine. Since message-passing is accepted as fundamental to Occam's philosophy, some form of dynamic process cre-

ation would be helpful, at least in the limited form of recursion, to make programming a wider set of algorithms easier.

4.5 Summary

Occam is a simple, easy to learn parallel programming language. However, it does not have two of the characteristics of parallel algorithms: dynamic graph creation and shared memory communication. It makes up for the first lack in part with a capability for dynamic graph traversal, as in the quicksort example. It was able to successfully code all four of the parallel algorithms, with some extra process overhead in transforming the dynamic graph creation algorithms, quicksort and simulation, into static graph algorithms.

Chapter 5

Sisal

5.1 Language Description

Sisal stands for "Streams and Iteration in a Single Assignment Language" [McGraw 85]. It is a functional data-flow language that is designed to allow programmers to ignore issues of explicit parallelism control, and simply state their program in a very compact and high-level fashion. It is assumed that data-flow architectures will be developed that will be capable of large amounts of small-grain parallelism, which the SISAL compiler will extract from the program. The philosophy behind SISAL and other single assignment functional languages is to eliminate the spurious dependencies between operations that appear to arise when imperative languages with global variable side-effects are used. In principle, an intelligent compiler will be able to analyze a SISAL program completely for parallelism, and target it efficiently to a given architecture.

Functional languages describe a program as an expression to be evaluated. Parallelism can arise when independent subexpressions can be evaluated concurrently. The data-flow model corresponds to the computation graph model with the following restrictions:

- nodes of the graph correspond to function evaluations
- arcs of the graph correspond to function arguments and results (data)

- a node has no state that persists between executions. It receives its arguments, executes, and produces its results, returning to its initial state.

A compiler for a functional language, such as SISAL will take the program, and produce an equivalent data-flow graph.

Some of the constructs of SISAL are:

- function definition
- the introduction of the stream data type for communication between functions
- the LET construct which defines a scope for value names within which to evaluate an expression
- the FOR construct, with two forms: non-product (sequential iteration) and product (cross and dot products of indices which can result in parallel evaluation)
- a number of special functions defined on data types. The ones used in the following programs are briefly defined here:
  - array_limh - returns highest index of array
  - array_setl - sets lowest index of array
  - array_adjust - sets bounds of array
  - catenate, || - concatenate arrays or streams
  - array_size - returns length of array
  - array_fill - creates and assigns values to array elements

- stream_first - returns first element of stream
- stream_empty - tests for empty stream
- stream_rest - returns all but first element of stream

SISAL is, as stated, a higher-level declarative language, with little explicit parallelism, except for the special array operations. The sample algorithms were coded as if the schedulable units of computation of the algorithms corresponded to the function definitions of SISAL, and the parameters were data arcs. It is understood that a smaller granularity would probably be extracted by a compiler targeted to a dataflow architecture.

5.2 Expressive Power

5.2.1 Block Triangular Solver

Sisal solutions

```
define Triangle

type vector = array[real];
type matrix = array[vector];
type blockmatrix = array[array[matrix]];
type blockvector = array[vector];

function Solve (Ablock : matrix; Bblock : vector returns vector)
let
    Blockdim := array_limh(Bblock)
in
   for initial
        J := 0;
        Xblock := Bblock;
        Newblock := Bbloc;;
   while J <= Blockdim
   repeat
        J := old J + 1;
        Newblock[J] := old Xblock[J] / Ablock[J,J];
           Xblock:=
               for initial
```

```
                    I := J;
                    Nwblk := Newblock;
                while I <= Blockdim
                repeat
                   I := old I + 1;
                   Nwblk[I] := old Nwblk[I] -
                                (Ablock[I, J] * Newblock[J]);
                returns value of Nwblk
                end for
   returns value of Xblock
   end for
end let
end function

function Matvect (Ablock : matrix; BblockI, BblockJ : vector
                  returns vector)

let
   Blockdim := array_limh(BblockI)
in
   for initial
      Xblock := BblockI;
      J := 0
   while J  <= Blockdim
   repeat
      J := old J + 1;
      Saveblock := old Xblock;
      Xblock :=
            for initial
               I := 0;
               Svblk := Saveblock;
            while I <= Blockdim
            repeat
               I := old I + 1;
               Svblk[I] := old Svblk[I] -
                           (Ablock[I,J] * BblockJ[J]);
            returns value of Svblk
            end for
   returns value of Xblock
   end for
end let
end function

function Transform (A : matrix; B : vector; Numblocks : integer
                    returns blockmatrix, blockvector)
let
   Vectorsize := array_limh(B);
   Blocksize := Vectorsize / Numblocks
in
   for I in 1,Numblocks
         Lo := (I-1)*Blocksize+1;
```

```
        Hi := Lo + Blocksize - 1;
        VectBlock := array_setl(array_adjust(B, Lo, Hi),1);
        Ablock1 := array_setl(array_adjust(A,Lo,Hi),1);
        Matrow :=
             for J in 1,Numblocks
                    Lo2 := (J-1)*Blocksize + 1;
                    Hi2 := Lo2 + Blocksize - 1;
                    Ablock2 := array_setl(array_adjust
                                (ABlock1[J],Lo2,Hi2),1);
             returns value of catenate Ablock2
             end for
   returns value of catenate Matrow,
           value of catenate Vectblock
   end for
end let
end function

function Triangle (A : matrix; B : vector;
                   Blocksize : integer returns vector)
% Blocksize = dimension of blocks
% return value is solution vector X
let
    Vectorsize := array_limh(B);
    Numblocks := Vectorsize / Blocksize;
    Blkmatrix, Blkvector := Transform (A, B, Numblocks)
in
   for initial
     J := 0;
     X := Blkvector
   while J <= Numblocks
   repeat
     J := old J + 1;
     NewXblock := Solve (Blkmatrix[J,J], X[J]);
     NewXsubcol :=
       for I in (J+1),Numblocks
         NewXblk := Matvect (Blkmatrix[I, J], X[I], X[J]);
       returns array of NewXblk
       end for
     First := array_adjust(X,1,J-1);
     X := First || NewXblock || NewXsubcol;
   returns value of X
   end for
end let
end function % Triangle
```

The following solution is another version of the same problem, but with a blocksize of 1 assumed (all operations are on scalars).

```
define Triangle

type vector = array[real];
type matrix = array[vector];

function Triangle (A : matrix; B : vector; Size : integer
                   returns vector)
% Size = dimension of A and B
% return value is solution vector X

     function Solve (Ael, Vel : real returns real)
         Vel/Ael
     end function % Solve

     function Mult (Ael, Vel1, Vel2 : real returns real)
         Vel1 - (Ael * Vel2)
     end function % Mult

for initial
   J := 0;
   X := B;
while J  <= Size
repeat
   J := old J + 1;
   NewXel := Solve (A[J,J], old X[J]);
   NewXcol :=
      for I in J+1,Size
          NXel := Mult (A[I,J], old X[I], old X[J]);
      returns array of NXel
      end for
   First := array_adjust(X, 1, J-1);
   X := First || NewXel || NewXcol;
returns value of X
end for
end function % Triangle
```

Discussion of Sisal solutions

The first solution represents an attempt to translate the algorithm exactly as described, with blocks of arbitrary dimension. Function Triangle takes a matrix, a vector, and an integer indicating block size for input parameters, and returns a vector (solution vector x). It does so by repeatedly transforming its input vector by one column of the matrix at a time with

the sequential FOR construct. This corresponds to producing a block of the solution vector at each iteration, and concatenating it onto the original vector, while at the same time changing the rest of the original vector through the Matvect function. Function Transform is an extra function introduced to transform the matrix and vector into a matrix of matrices (blocks) and a vector of blocks. These "blocked" data structures become the arguments to the Solve and Matvect functions. This solution attempts to create a static graph with message passing semantics similar to the Occam solution. Since it did not seem possible to compose two function types so that each produced input for the other, some of the parallelism was eliminated by processing the matrix of functions one column at a time. The only parallelism now observable is the subcolumn of Matvect functions (the FOR/IN construct in function Triangle). All other expressions are the sequential loops type.

Since the solution to this parallel problem attempting to force a block structure (and thus a given granularity of parallelism) onto the solution was as long as the imperative versions, and much harder to understand, a second solution was attempted with the same computation graph structure and traversal, but an assumed block size of one. That is, each Solve and Matvect function operated upon integers, not blocks, but the same execution order was kept of one column at a time. The result is the second, simpler version of the problem. The outer loop of function Triangle is still a sequential processing of columns, but the extraneous code for block operations has been removed, and the structure is much clearer. The code is much more compact; in fact, the functions Solve and Matvect resolve to one line apiece, and could be substituted in line for even shorter code. This represents a surprisingly

elegant translation of the algorithm, despite the introduction of ordering between the rows of Matvects, and the removal of the block semantics.

5.2.2 Parallel Quicksort

Sisal solution

```
define Quicksort

type Info = array[real]

function Quicksort (Data : Info returns Info)

   function Split (Data : Info returns Info, Info, Info)
          for E in Data
          returns
              array of E when E < Data[1]
              array of E when E = Data[1]
              array of E when E > Data[1]
          end for
   end function % Split

if array_size(Data) < 2
then Data
else
      let L, Middle, R := Split(Data)
      in
          Quicksort(L) || Middle || Quicksort(R)
      end let
end if

end function % Quicksort
```

Discussion of Sisal solution

The solution to this problem is an example in the back of the Sisal Language Reference Manual. This problem is ideally suited to the recursion of functional languages. Each invocation of function Quicksort splits the data (using a special parallel FOR/IN array processing construct) into two partitions and a middle, and then calls itself recursively on the two partitions.

This solution does violate the requirement of the algorithm that the array be stored in shared memory, and so presumably the segments of the array will be duplicated and transmitted in accordance with dataflow semantics, but this is simply the price paid for this model of computation. This is the solution that Occam would be able to encode as well if it had recursion of processes, and represents an advance over the "complete binary tree" formulation, despite the overhead of data copying.

5.2.3 Text Processing Pipeline

Sisal solution

```
define Transfer

type Charstream : stream[character];
type Lineimage : array[character];
type Linestream : stream[Lineimage];

function Disassemble (Inlines : Linestream returns Charstream)
for I in Inlines
returns
   for initial
       Image := stream_first(Inline);
       UpperBound := 80;
       I := 0
   while I <= UpperBound
   repeat
       I := old I + 1;
       Sendchar :=
         if I <= UpperBound
             then Image[I]
             else ' '
   returns stream of Sendchar
   end for
end for
end function

function Squash (Instream : Charstream returns Charstream)
for initial
   Asterisk := '*';
while  ~ stream_empty(Instream)
repeat
```

```
   Firstchar := stream_first(Instream);
   Sendchar :=
       if Firstchar  ~= Asterisk
           then Firstchar
           else
             let
               Nextchar := stream_first(stream_rest(Instream));
             in
               if Firstchar = Nextchar
                   then '@'
                   else Firstchar
             end let
   Instream :=
       if Sendchar = '@'
           then stream_rest(stream_rest(old Instream)
           else stream_rest(old Instream);
returns stream of Sendchar
end for
end function

function Assemble (Instream : Charstream returns Linestream)
for initial
   Newline := array Lineimage [];
while ~ empty_stream(Instream)
repeat
   Newline :=
     for initial
       J := 0;
       UpperBound := 125;
       Nline := array Lineimage[];
     while (J <= UpperBound) & (~empty_stream(Instream))
     repeat
       J := old J + 1;
       Nextchar := stream_first(Instream);
       Nline[J] := Nextchar;
       Instream := stream_rest (old Instream);
     returns Nline
     end for
   Sendline :=
     if array_limh(Newline) = 125
        then Newline
        else Newline ||
             array_fill(1,(125 - array_limh(Newline) + 1),' ')
returns stream of Sendline
end for
end function

function Transfer (Inlines : Linestream returns Linestream)
     Assemble(Squash(Disassemble(Inlines)));
end function
```

Discussion of Sisal solution

This was an opportunity to use streams as the method of communication between functions. A single stream is defined that threads through the three function definitions. Each function accepts a stream and returns a stream, and the entire pipeline is neatly captured as a three-deep nesting of function invocations. The parallelism, if any, rests in the semantics of streams. If an element of a stream is available to its consumer as soon as it is produced, then the three function invocations will execute in parallel on successive stream items. If a stream must be completely produced by a function before it can be consumed, then the pipeline will in fact be a sequential program. Even without assuming the greedy stream consumption implementation, the remaining requirement of this algorithm is met in that the middle function Squash is at liberty to create its own stream of values based on the stream it is receiving (in other words, there is no need to append a dummy value to the output stream in between reading successive input values).

5.2.4 Parallel Simulation

No Sisal version of this algorithm has been coded. The difficulty lies in two factors:

1. the necessity of maintaining state in the nurse node in order to keep the fairness of the queues
2. the difficulty of composing functions that each produce input for each other. This was the same problem that arose in the triangular solver algorithm above. It was possible to constrain that algorithm to allow for

lesser parallelism, and produce the same arithmetic results. However, the value of the parallel simulation algorithm is its emulation of a system of interacting modules with persistent state, not the production of an arithmetic value.

5.2.5 Conclusions

Although the flavor of functional programming is simply to define a value, and not to impose a form or order of evaluation, it has been an interesting exercise to translate these algorithms into functional form, using the naive assumption that the function definitions of SISAL constitute the schedulable units of computation. In all cases but the parallel simulator, it has proven possible to capture the spirit of the algorithm to some degree. Interestingly, attempting to impose a specific granularity of data introduced more complexity into the code of the triangular solver than the attempt to enforce an order of evaluation.

The main result of the exercise is that two-directional forms of module interaction are difficult to encode, given function composition as a means of defining a problem. There may be a way to get around this difficulty using streams, but it is not clear from the language definition.

5.3 Understandability

The constructs of SISAL that limit parallelism are few and clearly defined, so that after some practice, it is possible to see what evaluations are capable of proceeding in parallel. Reconstructing a possible dataflow graph from the code simply requires a knowledge of dataflow and functional

semantics.

It should also be mentioned that the hierarchical nature of module definition and function composition in Sisal would make coding a large system manageable.

5.4 Ease of Use

The ease of use of any functional language depends on how familiar the programmer is with the notion of defining values rather than specifying sequences of steps. It proved not too difficult to code all of the examples, except for the parallel simulation. This may be because the notion of persistent state which is fundamental to the simulation is foreign to a functional language. The first triangular solver solution demonstrated the difficulty of forcing a granularity of data upon the problem. However, it must be noted that the approach taken here to programming the algorithms is counter to the philosophy of high-level programming; the ease of use should come from abandoning granularity choice and execution control to the compiler.

5.5 Summary

One apparent limitation to Sisal's expressive power for parallel computation was the inability to define function modules so that two different modules could provide input to each other. This resulted in limiting the parallelism of the triangular solver algorithm, and in not being able to code the parallel simulation. It has been duly noted however, that declarative languages are not meant to be programmed in such an explicit fashion; rather,

one is meant to define the required value and allow the compiler to extract the implicit parallelism in its evaluation.

Chapter 6

CSL (Computation Structures Language)

6.1 Language Description

CSL is a language for parallelism specification and control, developed in the context of the Texas Reconfigurable Array Computer [Sejnowski 80, CSL 85]. CSL is based on the philosophy of abstraction of parallelism specification and control from the sequential computation of an algorithm. CSL is special-purpose in the sense that its sole purpose is to describe abstract parallelism, but general-purpose in that it is desirable that any parallel algorithm be expressible in CSL.

A CSL program describes an abstract parallel computation graph and its traversal. The actual computation of the schedulable units of computation is encapsulated in programs written in ordinary sequential programming languages. These programs are linked into the CSL program at runtime.

The current version of CSL has the following capabilities:

- configuration of a computation graph.

  The programmer can specify the schedulable units of computation (called tasks) and their communication topology in the Construct statement. Both the shared memory and channel models of communication are provided for. Shared variables are associated only with the subset of

tasks that may actually access them; they are listed in brackets after the name of the file containing the task's object code. Channels are defined as unidirectional FIFO queues between tasks, and the degree of buffering on the channels may be specified by the programmer (the default is one). Each task may have one boolean task condition which is also named in the Construct statement.

Example:

```
CONSTRUCT
    SHARED
      SV1, SV2;
    TASKS
      TASK1 : TASK1OBJ [SV1, SV2] CONDITION TASK1COND;
      TASK2 : TASK2OBJ [SV1, SV2];
    CHANNELS
      CHAN1TO2 = DATACHANNEL FROM TASK1 TO TASK2 BUFFERS 3;
      CHAN2TO1 = DATACHANNEL FROM TASK2 TO TASK1;
END;
```

This Construct statement sets up a computation graph with two nodes, two channels, and two shared variables. No traversal is expressed.

- traversal of a computation graph.

  A Construct statement has a scope in which executable statements can affect its components. Ordinary Pascal-like looping and conditional constructs are provided. The programmer specifies when a task should execute, or should send or receive variables on a channel. The With

statement associates a shared variable with a particular execution of a task according to one of two disciplines: read-only, and read-write. An example is:

```
WITH X : Y
  DO EXECUTE TASK1
```

The meaning of this is: acquire variable X in read-write mode, and variable Y in read-only mode; associate them with TASK1's data space; and then execute TASK1 once. Then the variables are released from TASK1.

The With statement can rename a shared variable from the globally known name to a task-local name using the With/As form; this behaves as a parameterization of the task execution. The runtime system of CSL enforces the necessary mutual exclusion. Access to shared variables can be guarded by a When/With statement, where access is granted only if a condition expression is true, and the variable is available (not being accessed read-write by some other task).

Nondeterminism can be programmed with the When Condition Is statement, another form of Dijkstra's guarded commands. Parallel execution is achieved with the Cobegin statement, which can be nested arbitrarily. The separate arms of a Cobegin statement are denoted by '//'.

The elements of a Construct statement and some executable statements may be ranged with the Range statement. The Range statement replicates the Construct element or executable statement within its scope

with the values of the range variable of the Range statement. For example, the declaration

```
SOLVE (I) : C1 [A(I,I), X(I)] RANGE I = 1 TO NBLOCKS
```

has the same meaning as:

```
SOLVE (1) : C1 [A(1,1), X(1)];
SOLVE (2) : C1 [A(2,2), X(2)];
...
```

up to the current value of the variable NBLOCKS. When a Cobegin statement is enclosed in a range, the semantics is of replicated parallel streams.

The tasks are coded as ordinary sequential programs (shown in the examples in Pascal). Each program consists of two procedures: an init, that will be executed the first time that a task is activated, and a body, that will be executed once each time a CSL Execute statement is encountered. Variables that are global to the program (not local to the init and body procedures) retain their values between executions.

The only form of interface between the task code and the CSL program is a single boolean task condition for each task. If the task condition is in use, it will be listed as a CONDITION clause following the name of the task in the Construct statement. The convention is that task conditions are

false before activation of the task, and can only be set to true by the task code itself. The CSL program can only access, not assign, the value of the task condition.

6.2 Expressive Power

6.2.1 Block Triangular Solver

CSL Solution

```
JOB TRI;

CONST NBLOCKS = 6;
END;

BEGIN

CONSTRUCT
    SHARED
          A(I,J) RANGE I = 1 TO NBLOCKS, J = 1 TO I;
          X(I) RANGE I = 1 TO NBLOCKS;
    TASKS
         INITARRAY : INITBIN [A(I,J), X(I)]
                 RANGE I = 1 TO NBLOCKS, J = 1 TO I;
         SOLVE (I) : C1 [A(I,I),X(I)]
                 RANGE I = 1 TO NBLOCKS;
         MATVECT (I, J) : C2 [A(I,J), X(I), X(J)]
                 CONDITION MATCOND(I,J)
                 RANGE I = 1 TO NBLOCKS, J = 1 TO (I - 1);
 END; (* end construct *)

 WITH A[I,J], X[I] RANGE I = 1 TO NBLOCKS, J = 1 TO I
      DO EXECUTE INITARRAY;     (*initialize X, A *)
COBEGIN
   (// BEGIN
        COBEGIN
        (// WAIT MATCOND (J,I) )RANGE I= 1 TO J-1;
        COEND;
        WITH  X[J] AS XBLOCK: A[J,J] AS ABLOCK
           DO EXECUTE SOLVE (J);
        COBEGIN
        (// WITH X[I]  AS XBLOCKI : X[J] AS XBLOCKJ,
                                     A[I,J] AS ABLOCK
              DO EXECUTE MATVECT (I,J))
                  RANGE I = (J + 1) TO NBLOCKS;
        COEND;
```

```
    END;
     ) RANGE J = 1 TO NBLOCKS;
COEND;
END.

program init (input, output);
var blockmatrix : array[1..6,1..6] of
        array [1..100,1..100] of real;
    blockvector : array[1..6] of
        array [1..100] of real;

procedure init;
begin end;

procedure body;
var i, j, k, l: integer;
begin
    for i := 1 to 6
        for j := 1 to i
             for k := 1 to 100
                  for l := 1 to 100
                       read (blockmatrix[i,j,k,l]);
    for i := 1 to 6
         for j := 1 to 100
              read (blockvector[i,j]);
end;
begin
end.

program solve (input, output);
var xblock [1..100] of real;
    ablock [1..100, 1..100] of real;
procedure init;
begin
end;

procedure body;
var i, j : integer;
begin
for j := 1 to 100 do
    begin
      xblock[j] := xblock[j]/ablock[j,j];
      for i := j+1 to 100 do
           xblock[i] := xblock[i] - ablock[i,j]*xblock[j];
    end;
end;

begin
end.

program matvect (input, output);
```

```
var xblocki, xblockj : array [1..100] of real;
    ablock : array[1..100,1..100] of real;

procedure init;
begin
end;

procedure body;
var i,j : integer;
begin
  for j := 1 to 100 do
      for i := 1 to 100 do
          xblocki[i] := xblocki[i] - ablock[i,j]*xblockj[j];
end;

begin
end.
```

Discussion of CSL solution

The CSL program code proper is in all capitals, and the task code is given for completeness. CSL expresses this algorithm completely as specified. The matrix and vector are stored in shared memory. The set of tasks corresponding to MATVECT and SOLVE are structured to correspond to the matrix indices. A combination of mutual exclusion and task conditions is used to express the complex synchronization relationships in a compact form. A number of parallel streams are started off corresponding to the number of blocks in the problem (the RANGE after the outermost cobegin statement). After the array is initialized, each SOLVE waits for the MATVECTs in its row to signal they are finished by turning on their task condition, MATCOND. Since the first SOLVE has no predecessor MATVECTs, the range statement enclosing it evaluates to a negative, and it begins execution immediately. After it finishes, all the MATVECTs in the column beneath it begin executing, if they can gain mutual exclusion access to the correct block of the vector.

When a MATVECT finishes, it turns on its task condition, and eventually, the next SOLVE task will be triggered, and the next parallel stream will be activated.

This solution meets the requirements of the algorithm: the tasks are properly structured without waste; shared memory is used for the data structures; and maximum parallelism is exposed, since no arbitrary order is imposed on the MATVECTs in a row (the mutual exclusion is without priority).

6.2.2 Parallel Quicksort

There is no way to express parallel quicksort in CSL, because of its central property, separation of computation from control. CSL does not have recursion of tasks, but even the complete binary tree solution (see Chapter Four) is impossible for CSL, since the information about the length and position of the resulting array partitions is embedded in the task code, and the only method of associating data with a task is abstracted in the CSL code. The task condition is inadequate as a means of communication here, because it may only take on two values, true or false, and what needs to be communicated to the CSL program is a set of integers indicating size and location of the two partitions.

6.2.3 Text Processing Pipeline

CSL Solution

```
JOB TRANSFER;
VAR
A, B : INTEGER;
END;
```

```
BEGIN
    A := 80;
    B := 125;
    CONSTRUCT
         TASKS
            DISASSEMBLE : DBIN  CONDITION DQUIT;
            SQUASH : SQBIN CONDITION SQUIT;
            ASSEMBLE : ABIN CONDITION AQUIT;
         CHANNELS
            DTOSQ = DATACHANNEL FROM DISASSEMBLE TO SQUASH
                       BUFFERS = A;
            SQTOA = DATACHANNEL FROM SQUASH TO ASSEMBLE
                       BUFFERS = B;
    END CONSTRUCT;
    COBEGIN
       //  REPEAT
              EXECUTE DISASSEMBLE;
              SEND X TO DTOSQ;
            UNTIL DQUIT;

      //  REPEAT
              RECEIVE Y FROM DTOSQ;
              EXECUTE SQUASH;
              SEND Z TO SQTOA;
           UNTIL SQUIT
              EXECUTE SQUASH;
              SEND Z TO SQTOA;  (* clear pipeline *)

      //   REPEAT
                 RECEIVE Z FROM SQTOA;
                 EXECUTE ASSEMBLE;
            UNTIL AQUIT;
     COEND;
END.

program disassemble (input, output);
var line : array[1..80] of char;
    x : char;
    dquit , nextline: boolean;
    i : integer;
procedure init;
begin
   i := 1;
   nextline := true;
end;
procedure body;
begin
     if nextline
          then begin
              readln(line);
```

```
                nextline := false;
            end;
        if eof
            then begin
                   dquit := true;
                   x := '#';   (* end signal character *)
                 end
            else begin
                   if i = 81
                       then begin
                              x := ' ';
                              i := 1;
                              nextline := true;
                            end
                        else begin
                              x := line[i];
                              i := i + 1;
                             end;
                 end;
end;
begin
end.

program squash (input, output);
var y, z ,first : char;
    squit, clear,flush : boolean;
procedure init;
begin
clear := true;
flush := false;
squit := false;
end;

procedure body;
begin
    if flush then
         z := first
    else
    if clear then
          begin
             first := y;
             clear := false;
             if first = '#'
                then begin
                       z := first;
                       squit := true;
                     end
                else z := '%'; (* send dummy character *)
          end
    else
        begin
```

```
            if (first <> '*') or (first <> y)
                 then  begin
                         z := first;
                         first := y;
                         if first = '#'
                            then begin
                                   squit := true;
                                   flush := true;
                                 end;
                       end
                 else begin
                         z := '@';
                         clear := true;
                       end;
         end;
   end;
end;
begin
end.

program assemble (input, output);
var line : array[1..80] of char;
    z : char;
    aquit: boolean;
    i : integer;
procedure init;
begin
   i := 1;
   aquit := false;
end;
procedure body;
begin
     if z = '#'
          then begin
                 aquit := true;
                 for j := i to 125 do line[i] := ' ';
                 writeln(line);
               end
          else begin
                 if z <> '%' then begin
                     line[i] := z;
                     i := i + 1;
                     if i > 125
                       then begin
                              i := 1;
                              writeln(line);
                            end;
                 end;
               end;
end;
begin
```

```
end.
```

Discussion of CSL Solution

CSL is able to express the problem only with some waste, again because of the too-complete separation of computation and control. CSL can define the pipeline units and interconnections perfectly well, and uses channels. It is when it attempts to control the transmission of data on the channels, that problems arise.

The main problem is how the SQUASH task makes its decision as to whether there are two consecutive asterisks or not in the input stream. The pipeline within SQUASH has two stages, to hold the two consecutive characters for comparison. If the pipeline has been cleared by a previous squash, and the first character received happens to be an asterisk, then SQUASH must wait to receive the next character before transmitting either the asterisk or an '@'. The CSL program has no way of knowing this condition because the task condition is already being used to signal the need to flush the pipe, because of end of input. In other words, there are three conditions that SQUASH may be in: normal (one stage full), pipeline clear, and end of input. There is a way to work around this, and that is to send a dummy value to ASSEMBLE when, one that it will know to ignore. Then on the next execution of SQUASH, the pipeline will be filled again, and the decision to squash can be made. Thus, a receive is always followed by a send, but the principle of abstracting computation from control has been violated by placing more intelligence about the global state of the CSL graph inside the task body.

6.2.4 Parallel Simulation

There is no CSL code for this algorithm either. The problem again lies with CSL's separation of computation from control. CSL can specify all the necessary task types and the channels between them. Although it cannot create tasks at random, CSL can specify an array of tasks and execute them at random, the same approach taken in the Occam solution. However, to effect the communication on the correct channels, CSL must know the identity of the task with which each doctor and patient task needs to communicate. These identification numbers are what the nurse uses to queue the doctors and patients and to make assignments. When the nurse sends the id number to a task, there is no way that it can be communicated from the task body to the CSL program, which needs it immediately to index the correct channel for the next message send or receive. This is another case where the task condition is simply inadequate for communication.

6.2.5 Conclusions

CSL in its current definition has been found not to be general-purpose enough to express this set of algorithms adequately. CSL does have the two models of communication, but it does not have dynamic graph creation or dynamic graph traversal (as defined in chapter two). It should be noted that the ranging of Construct statements represents a sort of parameterized static graph creation, not true dynamic graph creation. The complete separation of task computation from parallelism control, except for a single task condition, leads to problems in three of the four algorithms, and it is apparently important, in that two of the algorithms could not be expressed at

all, even poorly. It is possible to "get around" lack of dynamic graph creation by creating more graph than is necessary to start with, but it is very difficult to fix a lack of dynamic graph traversal.

6.3 Understandability

CSL rates high on the criteria of understandability, since the parallel structure of the algorithm is completely abstracted into the CSL code. All nodes and interconnections are explicitly spelled out in the Construct statement, and the graph traversal actions are in the executable statements. It is part of the design philosophy of CSL that a task be ignorant of its place in a computation graph. However, we have seen a situation in the text processing pipeline, where the lack of knowledge of the CSL program about the tasks forced the tasks to be programmed with more knowledge of the CSL program. To understand the essential nature of the SQUASH task, which is to selectively suppress some data, it is necessary to examine the task code. The CSL code simply describes a task which receives one character, executes, and sends one character every time through the loop. It is only in the task code that we see that some of the characters sent are unnecessary. This undermines understandability because an algorithm's computation graph must now be reconstructed from the CSL program and the task codes together.

6.4 Ease of Use

CSL is easy to program on the limited class of algorithms for which it is suited: completely static graph creation and traversal. It does have one dynamic graph traversal mechanism, a high-level mutual exclusion constraint

on shared variables. This is an improvement over Ada, where the programmer must reprogram this mechanism every time it is needed. CSL is also easy to learn and understand because of its limited domain of parallelism control, and its small set of mechanisms.

Currently, CSL lacks hierarchical definition of computation graphs, which means any large system must be coded at a single level, making it more difficult to express and to understand.

6.5 Suggestions for Extensions to CSL

Two relatively simple extensions to CSL would help it address more general-purpose algorithms. More improvements could be suggested, but these have been chosen as the smallest extensions that can be added to increase CSL's power, and as the simplest changes to implement in the current run-time system.

1. The task condition mechanism should be expanded into task variables. These task variables would define all the conditions that a task could be in after a single execution. In the case of the pipeline, there were three separate conditions. A single variable of type integer could have been declared for the task, and then assigned a value of 1, 2, or 3, which would be interpreted by the CSL program as the three conditions. Alternatively, three boolean variables could be defined, each set to true or false after each execution. This would be more useful in situations where the conditions are not disjoint. It is more reasonable to program a certain amount of knowledge of what is required of it by the CSL

program into the task, than to program around known deficiencies of the CSL system in the task. The task variable solution would fix all of the problems causing the two algorithms to be inexpressible:

- The quicksort algorithm could define a quadruplet of integer task variables, start1, length1, start2, length2, to embody the information about the results of the partitioning to the CSL program, which would then decide whether to activate more partitions, and what subset of shared data to associate with them.
- The parallel simulation algorithm could define an integer task variable for patient and doctor tasks which holds the id number of the matching task. Then the CSL program could send the data on the correct channel.

2. The ability to make dynamic additions to the task configuration defined by the Construct statement would improve expression of some algorithms. Actually, in the case of CSL, defining more tasks than are going to be used is not as inefficient as with Occam, since the tasks will never be activated if an Execute statement is not encountered. However, it is closer to the spirit of a dynamic graph creation algorithm only to mention as many tasks as are needed.

Since tasks that are dynamically created at runtime are very likely to be referred to by a common name with a unique index, an addition to the Construct statement could define a dummy task type with an unevaluated index variable, along with associated object code file name, shared variables, and task variables. During the executable part of the program, a Create

statement could instantiate one of the dummy task type by specifying the name and a value for the index variable. Channels could also be the argument for a Create statement.

Example:

```
CONSTRUCT
    TASKS
         DUMMYTASK(I) : DUMMYOBJ INTEGER DUMMYVAR(I);
    CHANNELS
         DUMMYCHAN(I) = DATACHANNEL FROM DUMMYTASK(I) TO
                                          DUMMYTASK(I+1);
...
BEGIN
...
CREATE DUMMYTASK(10); (* automatically creating and associating
DUMMYVAR(10) for the task *)
CREATE DUMMYTASK(11);
CREATE DUMMYCHAN(10);
```

Extra care would have to be taken by the programmer that channels are not created that have no actual endpoints.

6.6 Extended CSL

An extended CSL has been designed, but not implemented, and used to code the three algorithms that gave ordinary CSL some trouble: pipeline, quicksort and simulation.

6.6.1 Definition of the extensions

The extended CSL has these specific extensions:

1. A Create statement for channels and tasks has been added with the same syntax and semantics as described above.

2. A task may now have any number of task variables of type CONDITION or INTEGER. These variables may be subscripted. If a task variable name has only one subscript in the Construct statement, it corresponds to an unsubscripted variable of the same name in the task body. If a task variable name has more than one subscript in the Construct statement, it corresponds to an array of contiguous variables with the same name in the task body. The subscripts in the Construct statement need not be contiguous; the correspondence to the task variable array will be made in the order in which the subscripted variables are declared in the Construct. There is an example of this in the parallel quicksort code below.

3. Another extension to task variable semantics is that at the start of the CSL program, all task variables are zeroed out (integers set to zero, and conditions set to false) with respect to the CSL program. When a task is first initialized, the value of the task variables that exists within the CSL program at that time is placed in the corresponding variables within the task. This has the side-effect of task to task communication when two tasks share the same subscripted task variable, and one sets its value before the other task is initialized. An example of this is the setting of GO flags in the parallel quicksort below.

6.6.2 Parallel Quicksort

Extended CSL Solution

```
JOB QUICKSORT;

VAR
I,J : INTEGER; (* range variables *)
NUMELS, NUMNODES, NUMPARTS : INTEGER;
END;

BEGIN
   NUMELS := ...;       (* no. elements in array *)
   NUMNODES := (2 ** (NUMELS - 1)) - 1; (* no. nodes in tree *)
   NUMPARTS := NUMELS;  (* count partitions of size 1 *)
   CONSTRUCT
          SHARED
                SORTARRAY(I) RANGE I = 1 TO NUMELS;
          TASKS
                 PARTITION(I) : PARTOBJ
                    [ SORTARRAY(J) RANGE J = 1 TO NUMELS]
                    INTEGER SIZE(I), SIZE(2*I), SIZE((2*I)+1),
                            START(I), START(2*I), START((2*I)+1)
                    CONDITION GO(2*I), GO(2*I + 1)
                    RANGE I = 1 TO NUMNODES;
   END;
   CREATE PARTITION(1);            (* do first partitioning *)
   WITH SORTARRAY(J) RANGE J = 1 TO NUMELS
      DO EXECUTE PARTITION(1);
(* the following parallel streams wait for the signal to execute
   or the signal that the sorting is finished *)
   //  WAIT (GO(I) OR (NUMPARTS = 0));
        IF GO(I)
            THEN BEGIN
                  CREATE PARTITION(I);
                  WITH SORTARRAY(J)
                      RANGE J = START(I) TO (START(I)+SIZE(I)-1)
                      DO
                         EXECUTE PARTITION(I);
                  (* subtract partitioning value *)
                  NUMPARTS := NUMPARTS - 1;
                  (* if partitions have 1 element, subtract*)
                  IF (SIZE(2*I) = 1)
                     THEN NUMPARTS := NUMPARTS - 1;
                  IF (SIZE((2*I) + 1) = 1)
                     THEN NUMPARTS := NUMPARTS - 1;
                 END;
    ) RANGE I = 2 TO NUMNODES;
END.
```

```
program partition (input, output);
const
numels = ...; (* same size as in CSL program *)
var
size, start: array [1..3] of integer;
go : array [1..2] of boolean;
sortarray : array [1..numels] of integer;
procedure init;
begin
end;
procedure body;
begin
(* partitioning logic on sortarray, using size and start
   as guidelines *)
i := start[1];
j := start[1] + size[1] - 1;
mid := start[1];
while i <= j do
   begin
       if sortarray[i] > sortarray[j]
          then begin
                 temp := sortarray[i];
                 sortarray[i] := sortarray[j];
                 sortarray[j] := temp;
                 if mid = i
                     then mid := j
                     else mid := i;
               end;
       if i = mid
         then j := j - 1
         else i := i + 1;
    end;
(* after partitioning, set limits for next processes *)
if start[1] < (mid - 1)
   then begin
          start[2] := start[1];
          go[1] := true;
        end;
size[2] := (mid - 1) - start[1] + 1;
if (mid + 1) < (start[1] + size[1] - 1)
    then begin
           start[3] := mid + 1;
           go[2] := true;
         end;
size[3] := (start[1] + size[1] - 1) - mid;
end;
begin
end.
```

Discussion of Extended CSL Solution

Since the simple extensions of CSL that were proposed do not include recursive tasks, it is necessary to create the complete binary tree of tasks and find some way of signalling the unnecessary tasks that they should not execute. One way is the Occam solution of sending messages, but this requires the tasks to be loaded and executed to process the message. An alternative is to use a global condition within the CSL program that signals the end of the sorting. A global counter is used to count the number of partitions of size 1 that are created. Every partitioning task creates a mid section of size 1, so every execution decrements the counter by 1. Additionally, the partitions that are created may be of size 1. If one or both of the partitions is of size 1, then an additional subtraction is made. This makes use of the fact that during the sorting process, every element will either become a partitioning element (mid), or wind up in a partition of size 1, which is the base case causing recursion to terminate. When all such elements have been counted, the sorting is completed. It should be noted that CSL statements are atomic with respect to each other, so that no synchronization is required on CSL job variables between parallel streams.

The advantage of this solution is that it actually solves the problem, which could not be done with ordinary CSL. The disadvantage is that the overhead of extra task execution has been moved into the CSL program as extra parallel stream execution.

6.6.3 Parallel Simulation

Extended CSL Solution

```
JOB SIMULATION;

VAR I, J : INTEGER; (* range variables *)
END;

BEGIN
   CONSTRUCT
        TASKS
          PATIENT(I) : POBJ INTEGER DN(I), PSELF(I)
              RANGE I = 1 TO 100;
          DOCTOR(I) : DOBJ INTEGER PN(I), DSELF(I)
              RANGE I = 1 TO 3;
          NURSE : NOBJ CONDITION QUIT, MATCH
                       INTEGER PNUM, DNUM;
        CHANNELS
          PTON(I) = DATACHANNEL FROM PATIENT(I) TO NURSE
                    RANGE I = 1 TO 100;
          PTOD(I,J) = DATACHANNEL FROM PATIENT(I) TO DOCTOR(J)
                    RANGE I = 1 TO 100, J = 1 TO 3;
          DTON(I) = DATACHANNEL FROM DOCTOR(I) TO NURSE
                    RANGE I = 1 TO 3;
          DTOP(I,J) = DATACHANNEL FROM DOCTOR(I) TO PATIENT(J)
                    RANGE I = 1 TO 3, J = 1 TO 100;
          NTOP(I) = DATACHANNEL FROM NURSE TO PATIENT(I)
                    RANGE I = 1 TO 100;
          NTOD(I) = DATACHANNEL FROM NURSE TO DOCTOR(I)
                    RANGE I = 1 TO 3;
      END;
(* broadcast identity numbers to each task *)
EXECUTE NURSE;
FOR I = 1 TO 100
    DO BEGIN
         SEND PID(I) TO NTOP(I);
         RECEIVE SELF FROM NTOP(I);
       END;
FOR I = 1 TO 3
    DO BEGIN
         SEND DID(I) TO NTOD(I);
         RECEIVE SELF FROM NTOD(I);
       END;
(* each task stores its identity number *)
COBEGIN
  (// EXECUTE PATIENT(I)) RANGE I = 1 TO 100;
  (// EXECUTE DOCTOR(I)) RANGE I = 1 TO 3;
   // EXECUTE NURSE;
COEND;
```

```
(* start the simulation *)
COBEGIN
// (REPEAT
       EXECUTE PATIENT(I);  (* not withdoctor *)
       SEND SELF TO PTON(I);
       RECEIVE ASSIGNMENT FROM NTOP(I);
       EXECUTE PATIENT(I); (* withdoctor *)
       SEND SYMPTOM TO PTOD (PSELF(I), DN(I));
       RECEIVE RX FROM DTOP (DN(I), PSELF(I));
   UNTIL QUIT;) RANGE I = TO 100;

// (REPEAT
      SEND SELF TO DTON(I);
      RECEIVE ASSIGNMENT FROM NTOD(I);
      EXECUTE DOCTOR(I);  (* seenext *)
      RECEIVE SYMPTOM FROM PTOD (PN(I), DSELF(I));
      EXECUTE DOCTOR(I);  (* diagnosing *)
      SEND RX TO DTOP (DSELF(I), PN(I));
    UNTIL QUIT; ) RANGE I = 1 TO 3;

// REPEAT
     COBEGIN
       ( // IF NOT EMPTY (PTON(I))
               THEN RECEIVE PID(I) FROM PTON(I))
            RANGE I = 1 TO 100;
       (// IF NOT EMPTY (DTON(I))
               THEN RECEIVE DID(I) FROM DTON(I))
            RANGE I = 1 TO 3;
     COEND;
     EXECUTE NURSE;
     IF MATCH THEN BEGIN
        SEND PASSIGNMENT TO NTOP(PNUM);
        SEND DASSIGNMENT TO NTOD(DNUM);
      END;
   UNTIL QUIT;
  COEND;
END.

program nurse (input, output);
const maxiterations = ...;
var
passignment, dassignment, pnum, dnum, i,
numiterations : integer;
pid : array [1..100] of integer;
did : array [1..3] of integer;
match, quit, first, second : boolean;
procedure init;
begin
numiterations := 0;
first := true;
second := false;
```

```
end;
procedure body;
begin
if first
    then begin
          for i := 1 to 100 do pid[i] := i;
          for i := 1 to 3 do did[i] := i;
          first := false;
          second := true;
    end
else if second
       then begin
              (* zero out pid and did arrays *)
              second := false;
            end
else begin
        (* examine pid and did arrays, queueing non-zero entries
*)
        (* zero out pid and did arrays *)
        (* if there is at least one entry in each queue ... *)
           passignment := pqueue[phead];
           dassignment := dqueue[dhead];
           match := true;
        (* adjust queues *)
        (* if there is no match, set match to false *)
        numiterations := numiterations + 1;
        if numiterations > maxiterations
             then quit := true;
    end;
end;
begin
end.

program doctor (input, output);
var
   selfidentify,  (* true before doctor knows own id number *)
   seenext : boolean; (* true when waiting for next patient *)
   dself, (* task variable *)
   self,   (* channel variable *)
   pn,   (* task variable *)
   assignment, (* channel variable *)
   symptom, rx : integer;
procedure init;
begin
   selfidentify := true;
   seenext := true;
end;
procedure body;
begin
  if selfidentify  (* store own identity number *)
         then begin
```

```
                dself := self;
                selfidentify := false;
              end
  else if seenext  (* find out next patient *)
         then begin
                pn := assignment;
                seenext := false;
               end
 else begin  (* diagnose patient *)
              (* rx := some function of symptom *)
              seenext := true;
      end;
end;
begin
end.

program patient (input, output);
var
    self, (* channel variable *)
    pself, (* task variable *)
    assignment, (* channel variable *)
    dn, (* task variable *)
    symptom, rx : integer;
    withdoctor, selfidentify : boolean;
procedure init;
begin
  withdoctor := false;
  selfidentify := true;
end;
procedure body;
begin
   if selfidentify   (* obtain own identity number *)
       then begin
               pself := self;
               selfidentify := false;
           end
   else if withdoctor   (* obtain doctor assignment *)
          then begin
                  dn := assignment;
                  withdoctor := false;
              end
           else begin     (* be well, then get sick *)
                  (* delay = some function of rx  *)
                  symptom := (* some random number *);
                  withdoctor := true;
                end;
end;
begin
end.
```

Discussion of Extended CSL Solution

The extended CSL solution for the parallel simulation problem is fairly straightforward, and similar to the Occam solution. One thing that should be mentioned is that task variables may not be sent on channels. In the simulation, a task variable is used to index a channel, and the value of the variable is based on what has been sent to the task on a channel from the nurse. An extra execution is required simply to assign the channel variable to the task variable.

The doctor and patient tasks each have three different phases of execution: self-identification at the first, and then alternation between communication with the nurse, and communication with the assigned task. This is a good example of how the extensions have somewhat eroded the isolation of the task code from the CSL program. Each task must now know the states it can be in with regard to the CSL computation graph, and perform different actions between each state transition. The task is divided up into three sections with a set of actions for each different state.

The advantage of using the task variables is that the problem is now expressible. The disadvantage is that task code and CSL code must be considered together to get a complete picture of the parallel algorithm. However, this correspondence is done in a very disciplined manner. The problem of multiple states did not arise with the quicksort, even though it used task variables, because each task executed once only.

6.6.4 Text Processing Pipeline

Extended CSL Solution

```
JOB TRANSFER;
VAR
A, B : INTEGER;
END;

BEGIN
    A := 80;
    B := 125;
    CONSTRUCT
        TASKS
           DISASSEMBLE : DBIN  CONDITION DQUIT;
           SQUASH : SQBIN CONDITION SQUIT, CLEAR, DECIDING;
           ASSEMBLE : ABIN CONDITION AQUIT;
        CHANNELS
           DTOSQ = DATACHANNEL FROM DISASSEMBLE TO SQUASH
                       BUFFERS = A;
           SQTOA = DATACHANNEL FROM SQUASH TO ASSEMBLE
                       BUFFERS = B;
    END;

    COBEGIN
         //  REPEAT
                EXECUTE DISASSEMBLE;
                SEND X TO DTOSQ;
             UNTIL DQUIT;

         //  REPEAT
                RECEIVE Y FROM DTOSQ;
                EXECUTE SQUASH;
                IF NOT DECIDING
                     THEN SEND Z TO SQTOA;
             UNTIL SQUIT;
             IF NOT CLEAR
                THEN BEGIN
                       EXECUTE SQUASH;
                       SEND Z TO SQTOA;
                   END;

        //   REPEAT
                RECEIVE Z FROM SQTOA;
                EXECUTE ASSEMBLE;
             UNTIL AQUIT;
    COEND;
```

```
program disassemble (input, output);
var line : array[1..80] of char;
```

```
  x : char;
  dquit , nextline: boolean;
  i : integer;
procedure init;
begin
   i := 1;
   nextline := true;
end;
procedure body;
begin
     if nextline
          then begin
              readln(line);
              nextline := false;
          end;
     if eof
          then begin
                 dquit := true;
                 x := '#';   (* end signal character *)
               end
          else begin
                 if i = 81
                    then begin
                            x := ' ';
                            i := 1;
                            nextline := true;
                         end
                    else begin
                            x := line[i];
                            i := i + 1;
                         end;
                end;
end;
begin
end.

program squash (input, output);
var y, z ,first : char;
    squit, clear, deciding : boolean;
procedure init;
begin
clear := true;
deciding := false;
squit := false;
end;

procedure body;
begin
    if clear then
          begin
             first := y;
```

```
            clear := false;
            if first = '#'
               then begin
                       z := first;
                       squit := true;
                       clear := true;
                    end
               else if first = '*'
                      then deciding := true
                      else deciding := false;
         end
    else
          begin
            if (first <> '*') or (first <> y)
               then  begin
                       z := first;
                       first := y;
                       if first = '#'
                            then begin
                                    squit := true;
                                    clear := false;
                                 end;
                     end
               else begin
                       z := '@';
                       clear := true;
                    end;
          end;
   end;
end;
begin
end.

program assemble (input, output);
var line : array[1..80] of char;
    z : char;
    aquit: boolean;
    i : integer;
procedure init;
begin
   i := 1;
   aquit := false;
end;
procedure body;
begin
     if z = '#'
          then begin
                 aquit := true;
                 for j := i to 125 do line[i] := ' ';
               end
          else begin
```

```
                    line[i] := z;
                    i := i + 1;
                    if i > 125
                      then begin
                             i := 1;
                             writeln(line);
                           end;
                  end;
end;
begin
end.
```

Discussion of Extended CSL Solution

Only a few minor changes were made in the text processing pipeline code, to show how the unnecessary sends and receives could be avoided. The SQUASH task now can express the three states it can be in after an execution (see discussion for ordinary CSL version). A Send can be suppressed when necessary. The SQUASH task now stands in the same hybrid relation to the CSL program as the doctor and patient tasks did in the parallel simulation. However, it was possible to remove some of the extra code from the ASSEMBLE task. Formerly, ASSEMBLE had to know when it was receiving a dummy character, and ignore it. In this version, the task never receives such a dummy message, and so it has become more isolated from the parallelism control. Additionally, using the three states of SQUASH as divisions in the task code makes the code more understandable with relation to the CSL program.

6.7 Summary

CSL, as originally defined, was found to have too static a structure to code a wide variety of algorithms. CSL had no method of dynamic task

creation outside the Construct statement, and no method of dynamic graph traversal because of its separation of computation code from parallelism control code. The only method of communication between task and CSL codes was through a single task boolean-valued condition, which did not cover the number of states a task could be in after a single execution. However, CSL does have the virtue of capturing a parallel computation graph abstractly and compactly. A set of extensions was defined that allowed the remaining algorithms to be expressed in CSL; these extensions were dynamic task creation, and extension of the task condition to a task variable. These are not the only extensions possible, but the ones that seemed simplest to implement. The task variable construct has been shown to erode the isolation of the task code from the parallel graph somewhat, but in a disciplined manner that is preferable to the cases (such as the pipeline) where extra task and CSL code is necessary to execute the algorithm correctly, and to the cases where an algorithm (quicksort, simulation) is completely inexpressible.

Chapter 7

SUMMARY AND CONCLUSIONS

7.1 Summary

A set of characteristics of parallel algorithms was defined, and a set of algorithms that embody all of the characteristics in varying combinations was formulated. A classification of languages based on explicit vs. implicit parallelism control and the degree of abstraction of this control was drawn up, and a representative language chosen from each subclass. The sample languages with which to compare CSL were chosen on the basis of their influentiality, and likeliness to survive the test of time. The languages chosen were Ada, Sisal, and Occam.

7.2 Conclusions

The testbed of algorithms was chosen to contrast models of communication with dynamic/static graph creation. The idea of dynamic graph traversal was brought out by the failure of CSL to express two of the four algorithms, the quicksort and the parallel simulation. Both of the algorithms had dynamic graph creation as a characteristic, but that was not the deciding factor, since both of them were programmable in Occam, which has an even less dynamic node creation mechanism than CSL. In addition, CSL was able to express a third algorithm, the pipeline, but only poorly, and for the same

reason. This is a weakness in the definition of CSL, which fortunately can be remedied in a way that is relatively simple and relatively undamaging to the concept of separation of parallelism control from computation. This is the extension of the task condition to a set of task variables. This corresponds nicely to the CSL concept that a task completes one execution before communicating with the CSL control program again. The task can be left in a number of states after each execution, and this information can be coded into the task variables. This seems preferable to coding around CSL's deficiency in the task bodies, and infinitely preferable to being unable to express an algorithm at all.

CSL has a quality worth preserving, and that is the ability to see clearly, almost at a glance, the structure of the parallel computation. In parallelism-extended languages like Ada and Occam, this structure is embedded in code modules that must be analyzed together to determine the existence of relationships. In extended CSL, the CSL program and the task bodies must sometimes be analyyzed together to obtain the whole picture of the computation graph, but the relationship is very disciplined, based on the notion of a finite number of task states that are captured in the task variables. In Sisal, the structure of the computation is not really a concern of the programmer, as long as he defines it correctly. This is fine for certain purposes, but does not encourage experimentation in developing new parallel algorithms.

Of the languages surveyed, Ada has the most expressive power, based on its ability to program abstract algorithms with little transformation. This is balanced, however, by Ada's complexity and sometimes con-

fusing semantics. Ada has both models of communication available to the programmer. Shared memory was used for the triangular solver and quicksort algorithms, and channel communication for the pipeline and parallel simulation. It was noted that a higher level synchronization construct than the shared pragma would be convenient for the programmer, to avoid having to program simple mutual exclusion every time.

Occam is easier to use when the algorithm is already cast in its channel model of communication, but has no dynamic graph creation capability. It required rethinking of the quicksort algorithm to recast it into a fixed binary tree version that entailed some waste creation and execution of processes. Occam has simpler syntax and semantics than Ada. Since, like Ada, intertask communication is embedded in the algorithm code, dynamic graph traversal is possible. This capability made it possible to transform the quicksort algorithm into the static version.

The Sisal discussion emphasized the difficulty of programming against the philosophy of implicit parallelism. Sisal would be easiest to use for a programmer who wished to avoid the complexity introduced by thinking explicitly about parallelism. When attempting to use Sisal to program the triangular solver and simulation algorithms, it was found impossible to compose functions in a way that allowed for two-way communication between two function modules.

CSL, as currently defined, has severe limitations, but if extended would have this advantage: the parallelism control is abstract and easy to program, while non-parallelism related issues, such as data types and operations, are buried in the task code. A task body language may be chosen that

best suits the computation (or the programmer) and, in some cases, recast slightly to fit into the CSL program.

An ideal explicit parallel language would achieve a workable balance between power and simplicity. A starting point would be to provide language constructs designed to embody the characteristics of parallel algorithms listed in chapter two. Both models of communication should be allowed, possibly in the form of an intermediate model, such as Linda's tuple space, that allows the programmer to use the characteristics of both as needed. The synchronization constructs should be sufficiently high-level to avoid continual recoding of common synchronization requirements. Finally, hierarchical definition of computation graphs should be available in the language.

Further work can be done on extending CSL. The extensions given were the simplest possible that would allow the algorithms to be expressed. Introducing hierarchy into CSL is necessary and fundamental to its computation graph model. Some mechanism of abstracting a CSL graph into a single task should be introduced.

BIBLIOGRAPHY

[Ambler 78] Ambler, Allen L., "Directions in Systems Language Design," *COMPCON Spring 1978*, pp.40-43.

[Andrews 83] Andrews, G.R. and Schneider, F.R., "Concepts and Notations for Concurrent Programming," *Computer Surveys*, v.15, no.1, Mar. 1983, pp.3-44.

[Barnes 82] Barnes, J.G.P., *Programming in Ada*, Addison-Wesley, 1982.

[Bloom 79] Bloom, Toby, "Synchronization Mechanisms for Modular Programming Languages," MIT/LCS/TR-211, MIT, Jan. 1979.

[Browne 85] Browne, J.C., "Formulation and Programming of Parallel Computations: a Unified Approach," *ICPP 1985*, pp.624-631.

[Conway 63] Conway, Melvin E., "Design of a Separable Transition-Diagram Compiler," *CACM*, v.6, no.7, July 1963, pp.396-408.

[CSL 85] *Computation Structures Language Reference Manual*, UT Computer Science Dept., Oct. 1985.

[Dijkstra 75] Dijkstra, E.W., "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs," *CACM*, v.18, no.8, Aug. 1975.

[DoD 83] Dept. of Defense., *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983.

[Gelernter 85A] Gelernter, David, et al., "Parallel programming in Linda," *ICPP 1985*, pp.255-263.

[Gelernter 85B] Gelernter, David, " Generative communication in Linda," *ACM TOPLAS*, v.7, no.1, Jan. 1985, pp.80-112.

[Hoare 78] Hoare, C.A.R., "Communicating Sequential Processes," *CACM*, v.21, no.8, Aug. 1978, pp.666-677.

[Lauer 78] Lauer, H.C., and Needham, R.M., "On the duality of operating system structures," In *Proc. 2nd Int. Symp. Operating Systems* (IRIA, Paris, Oct. 1978); reprinted in *Oper. Syst. Rev.*, v.13, no.2, Apr. 1979, pp. 3-19.

[McGraw 85] McGraw, James, et al., *SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual*, Version 1.2, March 1, 1985.

[Pountain 86] Pountain, Dick, *A Tutorial Introduction to Occam Programming*, INMOS, 1986. (Includes Occam Language Specification, March 12, 1986, as Appendix A)

[Reynolds 79] Reynolds, Paul Francis, Jr., "Parallel Processing Structures: Languages, Schedules, and Performance Results," Ph.D. diss., Univ. of Texas at Austin, Computer Science Dept., December 1979.

[Schwans 82] Schwans, Karsten, "Tailoring Software for Multiple Processor Systems," Ph.D. diss., Carnegie-Mellon University, Computer Science Dept., 1 October 1982.

[Sejnowski 80] Sejnowski, Matthew C., et al., "An Overview of the Texas Reconfigurable Array Computer," *AFIPS Conf. Proc.*, May 1980, pp.631-641.

[Stotts 82] Stotts, Paul David, Jr., "A Comparative Survey of Concurrent Programming Languages," *SIGPLAN Notices*, v.17, no.10, Oct. 1982, pp.50-61.

[Turner 83] Turner, D.A., *SASL Language Manual*, 1976, rev. Aug. 1979, Nov. 1983.

[Vax Ada 85] *VAX Ada Language Reference Manual*, DEC, Feb. 1985, Order no. AA-EG29A-TE.

[Wegner 83] Wegner, Peter, and Smolka, Scott A., "Processes, Tasks, and Monitors: a Comparative Study of Concurrent Programming Primitives," *IEEE TOSE*, v.SE-9, no.4, July 1983, pp. 446-462.