A Study of Fine-Grain Programming Using Cantor

Thesis by

Nanette J. Boden

In Partial Fulfillment of the Requirements
for the Degree of
Master of Science

California Institute of Technology
Pasadena, California

1988

(Submitted Nov 28, 1988)
Caltech CS-TR-88-11

Acknowledgments

The author would like to thank Chuck Seitz for his supervising both of the work presented here and the writing of this thesis; and Bill Athas for creating Cantor and for providing guidance concerning the programs and results presented here.

The research described in this report was sponsored in part by the Defense Advanced Research Projects Agency, DARPA Order number 6202, and monitored by the Office of Naval Research under contract number N00014-87-K-0745; by grants from Intel Scientific Computers and Ametek Computer Research Division; and by an AT&T fellowship.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

As the size of a computation increases, so in general do the opportunities to perform parts of the computation concurrently (in parallel). A *multicomputer* [3] is one of the computer architectures designed to exploit this concurrency. A multicomputer consists of a number of programmable computers, called *nodes*, each with an instruction-interpreting processor and private memory. In addition, each node can support multiple *processes* that share the resources of the physical node. The available memory of the node is partitioned between processes so that each is provided with a private address space. Processes communicate exclusively by *message passing*.

Within the realm of multicomputing, programs and machines are classified according to their *grain* size. For programs, this informal concept is based on the size of the parts of the computation that are performed in parallel. Programs of a larger grain size divide the computation between execution units, or processes, that contain a larger amount of the state of the computation. One metric for evaluating the grain size of a program is the frequency of message-passing operations performed by a process relative to other internal operations. Processes containing more state require information from other processes less frequently, and thus engage in less message passing. When message passing does occur in these processes, messages used to convey the required information are of a length commensurate with the grain size.

Medium-grain programs are composed of processes that perform on the order of hundreds to thousands of instructions between communication actions. Messages passed between processes typically contain tens to hundreds of data items. *Fine-grain* programs consist of processes that execute tens to hundreds of instructions between message-passing operations. The messages present in these programs are quite small, usually only a few data items.

Machines designed to execute medium-grain programs efficiently, appropriately called medium-grain multicomputers, were pioneered in the early 1980s [14], and have become an established class of concurrent computer architectures. These machines typically support megabytes of memory per node, and, in current technology, may have hundreds of nodes. Message-passing costs are generally high in these machines, with several tens to hundreds of instructions being executed in the time necessary to send a single message. Given the infrequency of message passing in medium-grain programs, these costs are not a limiting factor in computation.

Machines optimized for fine-grain computations are still in the developmental stage. A first prototype of the Caltech Mosaic, scheduled for completion in 1989, likely will be the first operational large-scale fine-grain multicomputer. Fine-grain machines will have essentially the same structure as medium-grain ones, yet may be composed of tens of thousands of nodes, each having several tens of kilobytes of memory. Fine-grain and medium-grain multicomputers both will have about the same aggegrate amount of memory. The fine-grain machine can apply more computing power to a computation by partitioning the total storage into smaller units. Since message passing occurs frequently in fine-grain computations, achieving acceptable performance requires that these message-passing costs be much lower than those in the medium-grain variety. The required decrease in message latency will be achieved through a combination of machine organization and improved routing mechanisms [3].

The advantage of the fine-grain multicomputer lies in its ability to exploit nearly all the realizable concurrency in any application. The task to be performed is divided into as many useful subtasks as possible. The fine-grain machine then performs as many tasks as it can concurrently. The task as a whole is thus completed as quickly as possible.

While programming models and techniques for medium-grain machines are well understood, practical programming techniques for the fine-grain machines have only recently begun to develop. Does the constraint of small granularity require new programming models and paradigms? What programming constructs are essential to the efficient expression of fine-grain concurrent computation? Clearly these are not questions to be answered purely from a theoretical perspective. Useful paradigms must be observed through analysis of solutions to a variety of problems. Similarly, the set of syntactic constructs can be evaluated only through use in application programs.

This thesis is an investigation of these areas through the use of a fine-grain programming system that is already available — the Cantor programming system. Cantor was developed by W.C. Athas [1,2] as a tool to explore issues in fine-grain programming. Advances in compiler technology and in program flow analysis provide valuable tools for developing fine-grain programs. The Cantor system currently consists of a compiler for Cantor code, a sequential interpreter, and various program-profiling mechanisms. Cantor has also been ported to a variety of concurrent machines, including the Intel iPSC/1, the Cosmic Cube, and the Sequent.

The approach to this thesis experiment has been to write Cantor programs that will perform a variety of tasks, and then to study those programs. Looking for similarities and differences in a wide range of application programs yields insight into the fundamentals of fine-grain programming. Study of the programs has also influenced the definition of the Cantor language. This thesis reports upon the results of this series of programming experiments.

Thesis flow is as follows:

- The remainder of this chapter describes the essentials of the Cantor programming language and model.
- Chapter 2 describes the implications of the programming experiments on the definition of the Cantor language. Based on the results of program writing, constructs were added to facilitate programming while extraneous features were deleted.
- Chapter 3 contains five of the more interesting programming experiments conducted for this thesis. The operation of each program is explained and important features are noted.
- Chapter 4 concludes with observations about fine-grain programming using Cantor. The assumptions made in the programming model are evaluated and future work is discussed.

1.2 Cantor

Cantor, or the Caltech Actor Notation, is based on the Actor model of computation. The textual representation of a Cantor program is composed of a set of ***definitions***, each of which serves as a template for creating ***objects*** whose behavior is specified by those definitions. Computation performed by an object is message-driven; *ie*, the object remains inactive until a message arrives for it. In essential Cantor [1], objects are constrained to receive messages in the order in which they arrive; all messages sent are guaranteed of eventual delivery. An object can preserve information between message receipts via a set of ***persistent*** variables.

The response of the object to a message must be finite, and may include zero or more of each of the following actions:

- the sending of messages
- the creation of new objects
- the modification of internal variables

After the object has responded to the message as prescribed in the object definition, it must prepare itself to either receive another message or self-destruct.

The first two actions listed above correspond to the Actor primitives, *send* and *new*. Whereas most programming methods rely on shared variables and recursion, Cantor is an experiment in learning to program efficiently with the *send* and *new* primitives.

An illustrative example of a Cantor program computes the factorial function:

```
fac() ::
[  (n : int, requester : ref)
   if (n < 2) then
      send (1) to requester
   else
      send (n−1, self) to fac()
      [  (result : int)
         send (n*result) to requester
      ]
   fi
]

[  (console : ref)
   send (6, console) to fac()
   [  (result : int)
      send (result) to console
   ]
]
```

The fac object definition defines the behavior of the objects that actually compute the factorial function. Each fac object has no persistent variables (denoted by ()), but rather receives the value of the factorial it is to compute in a message. It also receives a reference to the object to which it will send the result. Objects must possess the reference value of an object in order to send a message to that object.

The body of the object definition implements a type of recursion. If the base step has been reached (n < 2), a reply is sent to the requester. Otherwise, another level of recursion is implemented by instantiating a new fac object and sending to it n-1 and reference to this object, self.

Upon completion of this statement, the object can do no more useful work on the problem until the reply from the recursive step has been received. This reply message is received via a nested *message description*, delimited by a pair of matched square brackets ([ ]). This description specifies the object's response to the next message received. If a new message description is reached and the message queue is empty, the object is suspended until a message is received. In the factorial example, the new message description specifies that a message containing an integer, result, will next be received. The message description does not enforce this discretion, however; the program's dynamics alone dictate the message protocols. In the factorial example, when the reply from the recursion result, is received, the object sends the product to its requester.

The second object definition in the program is called the main object. When a Cantor program begins execution, the main object is the only object in existence. The rest of the object graph must be constructed explicitly. The console reference provides the program with output capability.

Chapter 2

Cantor Version 2.2

For a full exposition of Cantor 2.0 (essential Cantor), the reader is referred to the Cantor User Report 2.0 [1]. As a result of numerous programming experiments, in particular those reported in this thesis, a new version, Cantor 2.2, is currently being implemented and documented. The Cantor programs in the following chapters were generally written initially in Cantor 2.0, but were then rewritten in Cantor 2.2. For reference, Figure 2.1 depicts the BNF for Cantor 2.2. In addition, the major changes in the new version will be discussed in the remainder of this chapter.

The Cantor programming notation was originally developed with only a simple and essential set of fine-grain programming constructs. Cantor 2.0 [1] provided the programmer with constructs to send and receive messages, to create new objects, and to modify the internal state of objects. The restriction to finite behavior within objects was enforced by providing no mechanism that could produce infinite behavior. No data structures were built into the language. Instead, all data structures had to be constructed by the programmer using the computing resources of objects and messages. Objects could process messages only in the order received, and were not able to hold unwanted messages in the message queue. The goal of this "minimalist" approach to language design was to start with the bare programming essentials, and then determine the desired higher-level constructs by writing and evaluating programs.

Using this sparse set of constructs, we began to write programs to solve a variety of application problems, including the problems in this thesis. As the library of Cantor programs grew, so did our understanding of Cantor's capabilities and limitations. While Cantor 2.0 was Turing-complete and mathematically elegant, the expression of some very important higher-level constructs was cumbersome and unsatisfactory. In addition, we determined that some language features had been included based on unfounded predictions of their utility in application programming. Consequently, Cantor 2.2 contains conservative additions to the constructs available in Cantor 2.0, as well as

⟨*program*⟩ ⟹ ⟨*definition*⟩* ⟨*description*⟩

⟨*definition*⟩ ⟹ ⟨*object definition*⟩ | ⟨*function definition*⟩

⟨*object definition*⟩ ⟹ ⟨*name*⟩ ⟨*persistent list*⟩ : : ⟨*description*⟩

⟨*function definition*⟩ ⟹ ⟨*name*⟩ ⟨*parameter list*⟩ { =} ⟨*description*⟩

⟨*description*⟩ ⟹ (*[| [ ) ⟨*body*⟩]

⟨*body*⟩ ⟹ ⟨*sequence*⟩ | ⟨*case*⟩ | ⟨*statement*⟩+

⟨*sequence*⟩ ⟹ ⟨*message list*⟩ ⟨*statement*⟩*

⟨*case*⟩ ⟹ case (⟨*name*⟩ : ⟨*data type*⟩) of ⟨*case entry*⟩+

⟨*case entry*⟩ ⟹ ⟨*selector*⟩ : ⟨*sequence*⟩

⟨*statement*⟩ ⟹ ⟨*if*⟩ | ⟨*let*⟩ | ⟨*call*⟩ | ⟨*send*⟩ | ⟨*assign*⟩ | ⟨*control*⟩ | ⟨*description*⟩

⟨*if*⟩ ⟹ if ⟨*expression*⟩ then ⟨*statement*⟩+ { else ⟨*statement*⟩+ } fi

⟨*let*⟩ ⟹ let ⟨*name*⟩ { ⟨*range*⟩ } = (⟨*assign expr*⟩ | vector ⟨*range*⟩ of ⟨*type*⟩)

⟨*call*⟩ ⟹ call ⟨*name*⟩ ⟨*list*⟩

⟨*send*⟩ ⟹ send ⟨*list*⟩ to ⟨*expression*⟩

⟨*assign*⟩ ⟹ ⟨*name*⟩ ⟨*index*⟩ = ⟨*assign expr*⟩

⟨*assign expr*⟩ ⟹ ⟨*expression*⟩ (⟨*newline*⟩ | ;)

⟨*control*⟩ ⟹ exit | repeat | return (⟨*expr*⟩)

⟨*name list*⟩ ⟹ () | (⟨*declaration*⟩ { , ⟨*declaration*⟩ }*)

⟨*declaration*⟩ ⟹ ⟨*name*⟩ { [⟨*range*⟩ }] { , ⟨*name*⟩ { [⟨*range*⟩] } }* : ⟨*data type*⟩

⟨*data type*⟩ ⟹ int | real | bool | sym | ref

⟨*expression*⟩ ⟹ ⟨*expr1*⟩ { (or | xor) ⟨*expr1*⟩ }*

⟨*expr1*⟩ ⟹ ⟨*expr2*⟩ { and ⟨*expr2*⟩ }*

⟨*expr2*⟩ ⟹ ⟨*expr3*⟩ { (= | <> | < | > | <= | >=) ⟨*expr3*⟩ }*

⟨*expr3*⟩ ⟹ ⟨*expr4*⟩ { (+ | -) ⟨*expr4*⟩ }*

⟨*expr4*⟩ ⟹ ⟨*primitive*⟩ { (* | / | mod) ⟨*primitive*⟩ }*

⟨*primitive*⟩ ⟹ ⟨*name*⟩ { ⟨*range*⟩ } | ⟨*selector*⟩ | ⟨*reference*⟩ | ⟨*real*⟩ | abs ⟨*primitive*⟩ | not ⟨*primitive*⟩ | (⟨*expression*⟩)

⟨*selector*⟩ ⟹ ⟨*symbol*⟩ | ⟨*integer*⟩ | ⟨*boolean*⟩
⟨*index*⟩ ⟹ [⟨*expr*⟩ | ⟨*range*⟩]
⟨*range*⟩ ⟹ ⟨*expr*⟩ . . ⟨*expr*⟩
⟨*reference*⟩ ⟹ self | nil | ⟨*object name*⟩ ⟨*list*⟩
⟨*list*⟩ ⟹ () | (⟨*expression*⟩ { , ⟨*expression*⟩ }*)
⟨*persistent list*⟩ ⟹ ⟨*name list*⟩
⟨*parameter list*⟩ ⟹ ⟨*name list*⟩
⟨*message list*⟩ ⟹ ⟨*name list*⟩
⟨*symbol*⟩ ⟹ " ⟨*char*⟩* "
⟨*logical*⟩ ⟹ true | false

Figure 2.1: BNF for Cantor 2.2

changes in some language features. These modifications are discussed in the remainder of this chapter.

2.1 Control Mechanisms

2.1.1 Internal Iteration

In the Cantor programming model outlined in Chapter 1, objects must respond to a message with a *finite* number of actions. After responding, an object must prepare to receive another message or must self-destruct. The motivation for this requirement stems from the need to assure eventual message consumption. If an object exhibits infinite internal behavior, then the eventuality of message consumption would be violated for any message queued for that object.

To prevent infinite behavior within an object, Cantor 2.0 did not support any form of internal iteration. To iterate, an object sent itself a message, thus ensuring that all messages in the queue would be processed eventually. During our programming experiments, this requirement proved to be an unnecessary complication. The following program fragment is an example in Cantor 2.2 of iteration used within a definition:

```
*[ (x : int)
   Code to process x values
   i = 1
   [  if (i ≤ fanout) then
         send (output) to input[i]
         i = i+1
         repeat
      fi
   ]
]
```

A message containing x is received and processed. Next, internal iteration is used to send a message containing output to each reference value in the vector input (see section 2.3). When the loop terminates, another x value can be received and the loop repeated.

The next fragment is the expression of the same activity in Cantor 2.0:

```
*[ (tag: sym, x : int)
   Code to process x values
   send ("iter", 1) to self
   *[   case (tag : sym) of
        "iter"  : (i : int)
                  send (output) to input[i]
                  if (i ≤ fanout) then
                     send ("iter", i+1) to self
                  else
                     exit
                  fi
        "input": (x : int)
                  Code to buffer x values
   ]
   Code to process buffered x values
]
```

This fragment illustrates the implementation of this procedure without using internal iteration. When an input message is received, the object begins iteration by sending itself a message. While iter messages are being received and iteration is not complete, the object executes the body of the loop and then sends itself another iter message. However, if an input message is received during the iteration, the programmer may need to defer processing of the input message until the iteration is complete. Buffering and processing deferred messages is a non-trivial task that can greatly complicate a Cantor program. While internal iteration is obviously not a necessity for Cantor, it is clearer and more concise than iteration by message passing.

Using Cantor 2.0, an object cannot exhibit infinite internal behavior because it cannot iterate internally. However, it may engage in infinite *external* iteration by sending itself

messages. This scenario presents the same difficulty as infinite internal iteration, but at a lower level of abstraction. Eventually, the input queue of the object will overflow with messages, thus violating the assurance of eventual message consumption. In the final analysis, the programmer is charged with the responsibility of ensuring that objects that iterate forever (either internally or externally) are not created.

2.1.2 Basic Blocks

Internal iteration is achieved by the addition of a new control mechanism in Cantor 2.2. In Cantor 2.0, program blocks consisted only of message blocks. To execute (or repeat, or exit) a given program block, the object had first to first receive a message. Program blocks that do not include a message receipt, *ie, basic* blocks, are the actual mechanism for internal iteration.

Basic blocks are also useful when objects are initially created. In Cantor 2.0, an object has to receive a message before it can perform any computation. In Cantor 2.2, an object can begin computation by using basic-block constructs to set up private variables, send start-up messages, and perform other initialization activities.

2.2 Functional Abstraction

Since Cantor 2.0 includes only the most essential constructs, incorporation of functional abstraction into Cantor was deferred to later versions. Rather than use functions, the programmer wrote object definitions with characteristics that were similar to functions. A function call could be emulated by establishing a protocol of message tags between the "calling" and "called" object. This approach was awkward and violated the abstractive principles associated with functions. In Cantor 2.2, invoking a function is conventional, as illustrated in this program fragment that invokes a function min, which returns the minimum of two parameters:

```
*[ (x, y : int)
   minval = min(x, y)
   Code using minval
]
```

Analogous behavior is achieved in Cantor 2.0 with the following program segment.

```
*[ (x, y : int)
   send (self) to min(x, y)
   *[   case (tag : sym) of
        "min"   : (returnval : int)
                  minval = returnval
                  exit
        "input" : (x, y : int)
                  Code to buffer additional pairs
                  received before the min message
   ]
   Code using minval
   Code to process buffered (x,y) pairs
]
```

Each program fragment receives a pair of integers, x and y, and then assigns the minimum of those values to minval. The first fragment, written in Cantor 2.2, uses a call to a user-defined function min. Execution of the object will not continue until the function min has returned a value that is then assigned to minval.

The second fragment illustrates the Cantor 2.0 expression of this procedure. When an input message is received, a function call is emulated by sending the values x and y to a new object, min. This object computes the minimum of the values and replies with a min message containing the result. Before this min message is received, however, additional pairs of x,y values may be received. These values will probably need to be buffered and then processed after the min message is received. The complexity that was introduced into the program by the lack of functional abstraction convinced us to incorporate functions in Cantor 2.2.

Cantor 2.2 supports two types of user-defined functions (described in the following subsections). Function definitions of either type are constructed in the same manner as object definitions. It is important to note that ***functions, like objects, can engage in message-passing operations.*** Function definitions differ from object definitions in that they usually use a return statement to return a value to the invoking object or function. If side-effects are the motivation for invocation, the return statement can be omitted or the return value can be discarded by using a call statement to invoke the function. (Similarly, using a call statement to instantiate an object discards the new object's reference value.)

2.2.1 Functions

The fundamental function abstraction in Cantor is referred to as a ***function.*** In the program text, function definitions do not have a token (such as :: for objects) separating the function name from the definition. For example,

```
min(x, y : int)
```

```
[ if (x < y) then return (x)
  else return (y) fi
]
```

To allow freedom in implementation, including in-line expansion, subroutine call, or remote function (described below), recursive function definitions are not permitted.

A function, upon invocation, executes its definition and then returns a value (or a list of values) to the creating object or function. A function does not possess a unique reference value; it assumes the reference value of the creating object. Within the function, any message-passing operations are conducted using the message queue of the invoking object. When the invoking object receives the return value, it regains control of its message queue and resumes execution.

Functions simplify the programmer's task by allowing him to create clearer, more succinct, object definitions. Function definitions are also easier to recycle in future programs than are object definitions. Libraries of user-defined functions eliminate the need for additional operators in language.

2.2.2 Remote Functions

From our program-writing experiments, we detected the need for another type of function, namely the equivalent of a remote procedure-call mechanism [15]. Using functions, the object can initiate activities that perform any message-passing operations directly on the object's message queue and then wait for these activities to complete. Frequently, however, the programmer would like to create activities that perform message-passing operations on a private message queue. This capability is provided by a second type of function, a *remote* function. In this application, invocation of a remote function suspends the execution of the invoking object until the reply from the function is received. Processing of messages that precede the reply is deferred. Remote function definitions are declared by using an = token to separate the function name and definition.

For example, a distributed queue of objects with multiple consumers, as in Figure 2.2, is difficult to manipulate in Cantor 2.0. Consider the following scenario: The queue master object, QM, receives put and get operations that it should perform on the distributed queue. Assume that a get message is received. The QM sends the get message to the head of queue. Upon receiving the get message, the head queue element object sends the enqueued value to the requesting object. It then must advance the head of the queue by sending its next link to the QM before terminating. If there is arbitrary delay on the delivery of this update message and consumers are not tightly constrained, an unbounded number of new get messages may be received before the update message. Since there is no discretion over message receives, the QM must explicitly queue those new get requests until the new head of the queue is received. One can see that requiring a new unbounded queue to manipulate a queue will continue recursively; the same problem will

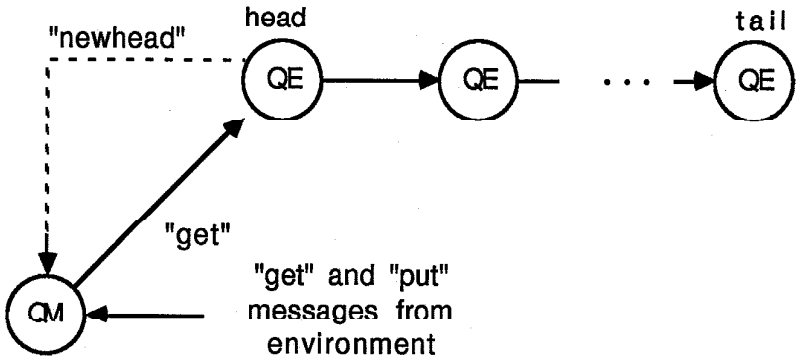

Figure 2.2: Distributed-queue data structure

occur when the queue of get messages is processed.

Since unbounded queues will eventually exhaust the physical resources of any machine, the underlying machine runtime systems must possess mechanisms to handle them. Additionally requiring the programmer to explicitly handle the above scenario is not necessary and is not acceptable in application programming. A remote function allows the programmer to initiate the get operation and suspend execution until the new head of the queue has been received. The function coordinating the operations on the distributed queue possesses its own reference value and, thus, can conduct message-passing operations privately; additional get messages queued for the QM do not affect the message passing of the function. When the function returns a value, via a special reply message, the QM detects and processes that message before it accepts any new get messages. The QM then resumes normal execution. The problem of handling the unbounded number of get messages is then pushed to a lower level of programming, namely that of the custom object/function level (Section 2.4).

A call to an remote function results in the creation of a custom function and an instance of the invoked function definition. The invoked function will possess a unique reference value and a private message queue. The invoked function executes its code and eventually sends a reply message to the invoking object. The custom function manipulates the message queue of the invoking object by examining each message received and replacing it *in order* in the queue if it is not the reply. When the reply is received, it is immediately sent to the invoking object. The message queue, which now contains messages that physically preceded or followed the reply, is returned to the invoking object. The custom function in this case then is a function that exercises system-level discretion over an object's message queue.

A remote function should be considered an experimental construct in Cantor 2.2. The manner in which limited message discretion should be provided to the programmer

is an issue that has not been fully explored. Remote functions are one type of mechanism for providing this capability. Other methods include using a special custom function to instantiate an object and then halt the creating object until a reply message has been received from the new object. This mechanism permits discretion over messages sent by objects and by functions. This approach, as well as remote functions, will be evaluated in future experimentation with Cantor.

2.3 Vectors

Cantor 2.0 provided no mechanisms for representing data structures internally to an object. All data structures were created by the programmer using the computing resources of objects and messages. Without preconceived notions about which data structures would be most appropriate in a general-purpose concurrent programming language, analysis of programming experiments was used to make that determination. In most application programs, array-type data structures were needed, but the nature of the Cantor 2.0 expression of these structures made them awkward to manipulate. Consequently, Cantor 2.2 supports the use of one-dimensional arrays, or *vectors*.

As an illustration, consider the program fragments discussed in section 2.1.1. The first fragment refers to a vector input that contains reference values to some other objects. The array was either defined within the object by

```
let input = vector 1..fanout of ref,
```

or received as part of the contents of a message,

```
(..., input[1..fanout] : ref, ...).
```

Without a vector construct, the reference values contained in input would have to be explicitly named, or represented using a chain of objects. Especially with the internal iteration feature, the ability to maintain and manipulate data internally is an important convenience. These data structures are encouraged to be small, however, because the machine model dictates that each object must be small enough to fit on one node.

One of the most interesting aspects of Cantor vectors is that dynamically-sized vectors can be sent and received in message-passing operations. Given the dynamic nature of Cantor computations, restricting all vector sizes at compile-time would make vectors less useful. In many applications, the size of vectors should grow or shrink as those vectors are passed between objects. Dynamically-sized vectors provide the programmer with the flexibility of sending vectors of exactly the required size in messages.

As usual, a programming convenience translates into an implementation problem. The fine-grain machine model implies that runtime systems for such machines will also be fine-grained. Managing dynamically-allocated vectors on-the-fly is non-trivial and

may require too much computing time and/or available memory. Further investigation of this question is planned using the Mosaic runtime system.

If manipulating vectors at runtime proves to be infeasible, an alternative that represents vectors as custom objects (section 2.4) will be investigated. In this scheme, custom *vector objects* would be used to represent each vector in the user program. User operations on a vector are translated by the compiler to remote functions that communicate with the appropriate vector object. For example,

```
parent = input[i]
```

would be compiled into a remote function call that sends the index i to the reference value of input and then assigns the reply value to parent. Since a reply is not required, write operations on vectors could be compiled as remote functions invoked with the call statement.

The advantage of this approach is that the runtime system no longer deals with the exception of vectors, and especially of dynamic vectors. All vectors (and subvectors) are represented by reference values, thus minimizing the storage required to represent the vector inside the object and in any messages. A key disadvantage is that, since variables within an object are local data, whenever the object sends a message containing a vector, a new vector object must be created to represent the vector in the receiving object. We expect that additional advantages and disadvantages will become apparent as this idea is further explored.

2.4 Custom Objects and Custom Functions

One of the most powerful additions to Cantor is the custom object/function facility, which was first suggested in [2, page 16] and is currently being implemented and documented. Custom objects are externally compiled objects that interface with a Cantor computation via the established message-passing system. Although the behavior of these objects conforms externally to Cantor object specifications, internally they may be written in another programming language (C, Fortran, assembly, *etc.*) and perform tasks not conveniently expressible in Cantor. For example, a custom object to perform I/O is currently being developed.

Most importantly, custom objects enable the programmer to optimize computations for a target machine without sacrificing aspects of the Cantor programming model. For example, a Cantor program that contains a "tree-master" object, an object that manipulates a distributed tree, would execute on any machine that supports Cantor. If, however, the target machine were a Tree machine, optimized for expressing and handling tree data structures, a custom object could replace the "tree-master" object. The custom object could exploit the special properties of the target machine, tailoring the representation and handling of the tree data structure to the machine model. The

custom object would communicate with the same message-passing protocols as the "tree-master," yet it would perform operations on the tree more efficiently on the given target machine. Custom objects thus allow the programmer to integrate program segments developed with different levels of machine abstraction.

Libraries of custom objects, written in Cantor or in other languages, will likely include objects to manage meshes, trees, and other aggregate data structures. Providing these objects as tools to the programmer could drastically increase program efficiency, since these objects would typically exhibit improved performance due to specialization. Development of this library is one of the most important areas of ongoing Cantor research.

2.5 Static *vs.* Dynamic Typing

One negative result of the Cantor programming experiments was the use of dynamic typing. Since message passing is late-binding, the ability to dynamically type message contents was initially thought to be useful. In practice, however, this facility was almost exclusively used in the coercion between integers and real numbers. Naturally, this particular application should not be the focus of polymorphic typing since storage class sizes for integer and reals differ widely in target machines. Since dynamic typing was rarely used and required a significant amount of hardware and/or runtime support, Cantor 2.2 supports only static typing of variables. This issue may be re-examined in later experiments, particularly if application programming suggests the need for polymorphic functions.

Chapter 3

Cantor Programs

In this chapter, five Cantor programs are presented and discussed in detail. These particular programs were selected for their individual characteristics that explore the application span of fine-grain programming. Programs whose object graphs are constructed recursively using small objects have a straightforward mapping to the fine-grain computational model; however, relatively few algorithms can be formulated in this manner. How efficiently can other, more representative, classes of computations be expressed using the fine-grain model? A goal of this thesis is to provide evidence that fine-grain programming can be applied successfully to solving problems that do not exhibit recursive object graphs and message patterns. This evidence consists of examples of programming solutions to problems that require arbitrary object graphs and message traffic. Graph algorithms comprise an important class of problems that require these complicated behaviors.

The majority of the programs presented in this chapter are solutions to graph problems. Concurrent algorithms to solve each problem are developed and then expressed in Cantor. That so many of the coded algorithms are elegant, often maximally concurrent, solutions of the problem for arbitrary graph instances is most encouraging. For each of the following programs, the graph problem and the concurrent algorithms are first explained. The Cantor implementation of the algorithm is presented and key program features are noted.

3.1 Shortest-Path Algorithms

One of the most studied graph problems is the shortest-path problem. In a directed graph G = (V,E), where V is the set of vertices and E is the set of edges, each edge $(v_i, v_j) \in$ E has an associated cost, or weight, w_{ij}. The elementary shortest-path problem involves discerning the distance of a vertex from a distinguished source vertex, which is

the minimum of the summed weights along each path from the source to the destination vertex. Some more involved shortest-path problems include finding the distances from a source vertex to each of the other vertices in the graph (Single Point Shortest Path, or SPSP), and finding the distances from each vertex to every other vertex (All Points Shortest Path, or APSP). These shortest-path problems and other related computations occur naturally in many application areas, including computer-aided design, simulation, and nearly all types of routing.

Since efficient solution of this problem is frequently required, much research has been focused on shortest-path solutions. In 1959, Dijkstra developed an algorithm [8] that remains the definitive sequential solution to the SPSP problem. In this algorithm, a vertex of the graph is said to be *visited* when a path from the source to that vertex has been established. A vertex has been *expanded* when new paths to each *successor* vertex (a vertex to which an outgoing edge is directed) are found by appending the path to the vertex being expanded with the edge connecting the successor vertex. Vertices of the graph that have been visited but not yet expanded are kept in a list of increasing distance from the source. The source vertex begins computation by visiting its successor vertices, and inserting them in order into the list. Next, the first vertex in the list is expanded and then deleted from the list. This process of modifying the list by adding visited vertices and removing expanded vertices continues until the list is empty. When computation terminates, each vertex has computed its distance from the source. The strength of Dijkstra's algorithm lies in its economy of effort. No redundant work is performed in the computation. Using a sequential computational model, the SPSP problem cannot be solved more efficiently.

Given Dijkstra's optimal algorithm for sequential computers, performance improvements in SPSP solutions can only be achieved by a concurrent formulation. Chandy and Misra [5] have modified Dijkstra's algorithm to exploit the potential concurrency in the expansion of vertices. Rather than expanding a single successor vertex, all successor vertices are expanded in a single step. Unlike the sequential version, Chandy and Misra's algorithm will, with high probability, perform redundant work, expanding vertices repeatedly. For this reason, the algorithm exhibits exponential time complexity for certain pathological graphs [6]. For the vast majority of graphs, however, the algorithm is simple and elegant, and achieves a high degree of concurrency. Details of the algorithm and its implementation in Cantor are presented in Section 1.1.1.

For the APSP problem, the Floyd-Warshall [9,17] algorithm remains the best sequential solution. Incremental construction of paths is used to find the shortest path between any pair of vertices, v_i and v_j, represented by the ij^{th} entry in an $|V| \times |V|$ matrix. In the k^{th} step of the algorithm, shortest paths through vertices numbered less than or equal to k are found for all pairs of vertices. After $O(|V|)$ steps, the i^{th} row of the matrix contains the distance of all vertices from vertex i. Thus, this algorithm has time complexity $O(|V|^3)$, and space complexity $O(|V|^2)$.

For each value of k, computation for each pair of vertices is independent of that of other pairs. This available concurrency has been exploited in the Parallel Floyd-Warshall algorithm [10]. In this formulation, shortest paths for all pairs of vertices are computed concurrently for each value of k. The mechanism that controls the computation distinguishes various implementations of the Parallel Floyd-Warshall algorithm. One of the most interesting of these implementations is van de Snepscheut's processor-array formulation [16]. This stategy for managing the phases of computation is explained in section 1.1.2, and the Cantor version is presented.

3.1.1 Chandy-Misra SPSP Algorithm

The algorithm presented by Chandy and Misra in [5] assumes the computational model of a network of communicating processes, where each process p_i represents a vertex v_i; and two processors, p_i and p_j, are neighbors, *ie*, they are able to communicate directly, if and only if an edge exists between them. In addition, only p_i contains information about the weight w_{ij} on each of its outgoing edges. Requiring this minimal knowledge of the network connectivity enables the algorithm to behave correctly even if the edge relation is dynamically changing. It appears that the robustness of algorithm in this respect is obtained only at the cost of having loose synchronization of vertex expansion, which is the cause of the worst-case exponential time complexity. For more synchronized algorithms, such as Dally's SPSP solution [6], more data about the network must be provided at each vertex.

It should be noted that the Chandy-Misra algorithm solves the SPSP problem even in the presence of negative cycles, *ie*, cycles of vertices whose summed path length is negative. Two phases of computation are then required: one phase to compute the lengths of the paths, and another phase to halt the computation and notify vertices reachable from a negative cycle that they are indeed infinite. For simplicity, in this implementation, the presence of negative weight cycles in the graph is not allowed. Consequently, only the first phase of computation is required. A brief outline of this phase follows.

Computation is initiated by the source sending a length message to each of its successor vertices. These length messages consist of the length of the shortest path to the successor computed thus far and the identity of the predecessor node that sent the message. Each vertex, upon receiving a length message, checks the path length value against the minimum path length received previously. If the new one is less, the vertex records the new length as the smallest received, records the sending node as its predecessor in the shortest path, and then sends a length message containing its own identity and the new length plus w_{ij} along each of the edges to its successor nodes. Thus, the length messages propagate through the graph, updating the shortest-path information as they proceed. When all length messages have been observed, each vertex has computed its

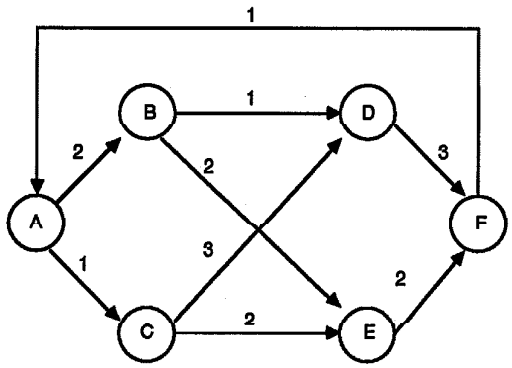

Figure 3.1: Example graph used in Chandy-Misra SPSP program

distance from the source and its predecessor in the path to the source.

As presented, the algorithm will solve the shortest-path problem, but will not announce termination. Instead of layering termination detection on top of the computation, Chandy and Misra utilize the diffusing nature of the computation, *viz*, the length messages, to incorporate the termination-detection schema of Dijkstra and Scholten [7] into the algorithm. Each length message sent to a successor node is eventually acknowledged, signifying that all vertices reachable from the successor have observed the message. Each vertex maintains a count of the number of unacknowledged length messages that have been sent to successor nodes. The rules of acknowledgement are simple:

- if a length message is received that contains a length that is not the new minimum length, then the message is acknowledged immediately.
- if the number of unacknowledged length messages is zero, an acknowledgement is sent to the predecessor in the shortest path to the source.

Thus, when the count of the source vertex is zero, all length messages have been observed and the computation has terminated.

The Cantor implementation of the Chandy-Misra algorithm is shown in Appendix A.1.

The example graph used in the computation is shown in Figure 3.1.

The main object of the program begins by instantiating each of the vertices in the graph as a vertex object. After vectors have been loaded with the edge and weight information for dispatch, the main object begins creating a stack of join objects, with itself as the bottom reference in the stack. Each reference in the stack is associated with the add edge message that informs a vertex of the existence of an outgoing edge. When the vertex has received the message, it sends an empty message, (), to the join object. When the join object has received two empty messages, one from the vertex object and one

from its predecessor in the stack that signifies that the predecessor has been "popped," the object "pops" itself from the stack by sending an empty message to its successor in the stack and then expiring. The main object begins the "popping" by sending an empty message to the original top of the stack. Thus, when the main object receives an empty message, all add edge messages have been received, denoting that other phases of computation can now safely be started. This paradigm of a synchronization stack is vitally important to fine-grain programming. Distributing the receipt of acknowledges over many nodes provides valuable concurrency while at the same time minimizing the amount of state that must be maintained in the constituent objects. Each of the programs in the remainder of this chapter at some point will utilize a variant of this fundamental synchronization mechanism.

Another LIFO data structure is used to represent the set of outgoing edges from a vertex. Initially, the vertex has no outgoing edges and, thus, has a stack consisting of a nil_edge object. The use of this object will be demonstrated in a moment. With receipt of each add edge message, an edge object is created and "pushed" onto the stack of outgoing edges. Thus when the first length message is received, the set of edges has already been constructed.

The vertex object responds to length messages exactly as described in the description of the algorithm given above. The termination-detection part of the algorithm, however, has been changed to better fit the fine-grain model. Instead of having the vertex receive and count acknowledgements for each of the length messages it sends, another join synchronization structure is used, in this case, a tree. As the length messages are sent to the successors, each edge object creates a join object to receive the acknowledgement of that message and the reply from lower levels of the tree. The newly created join object will reply to a current leaf of the tree, parent, and then become the new leaf. The nil_edge object, upon receiving a length message, acknowledges immediately, and, thus, begins collapsing the tree. When the root of tree, the main object, receives an empty message, the computation has halted.

The implementation of this algorithm highlights some of the special programming techniques suggested by the fine-grain programming model. With object creation equal in cost to message passing, and with optimized communication of short messages, distributed synchronization structures, such as the stack and the tree, are not expensive to create, while their usefulness is clear. Using the stack to signal the main object that the next phase of computation can proceed simplifies the code within the object. Distributing the termination detection using the synchronization tree enables the vertex object to concentrate on processing length messages, a job that certainly has a higher priority than counting acknowledgements. More applications of these structures will be seen in the programs in the remainder of this chapter.

3.1.2 Van de Snepscheut's Distributed Warshall Algorithm

In addition to the SPSP problem, the Chandy-Misra algorithm can be applied to solve the APSP problem. Each vertex in the graph behaves as a source vertex, initiating a length message phase to apprise the other vertices of their distance from this source vertex. When termination of the $|V|$ SPSP problems is detected, each vertex has computed its distance from every other vertex in the graph.

By design, the Chandy-Misra algorithm is most efficient when the graph of interest exhibits relatively sparse connectivity. For graphs with richer edge connectivity, the Parallel Floyd-Warshall algorithm is frequently more effective. This algorithm achieves $O(|V|)$ time complexity using $O(|V|^2)$ processors, one processor to compute the shortest path between each pair of vertices. The requirement of such a large number of processors to achieve a linear-time solution is certainly a disadvantage of this algorithm, yet does not render it useless. For richly connected graphs, the Parallel Floyd-Warshall algorithm still achieves significant speedup over the sequential version if fewer than $O(|V|^2)$ processors are available.

Regardless of the number of physical processors, $|V|^2$ processes are used in the Parallel Floyd-Warshall algorithm to compute the shortest paths. These processes are logically arranged in a $|V| \times |V|$ mesh. The ij^{th} process is responsible for computing the length of the shortest path, or the distance, from vertex i to vertex j. In the first step of the algorithm, the ij^{th} distance value is set to W_{ij}, where W is a weight matrix containing the initial distance between the processes, defined as the smallest weight on an edge between them, or ∞, if no edge connects the processes. Since the presence of negative cycles is not allowed, elements on the diagonal of the weight matrix have distance 0.

As in the sequential version, step k of the $O(|V|)$ required steps computes the length of the shortest path between vertices i and j that passes through an intermediate vertex numbered less than or equal to k for all ij vertex pairs. Let d_{ij} be the length of the shortest path from process i to process j computed thus far. For process ij, this length value, d_{ij}, is obtained by taking the minimum of the values of d_{ij} and $d_{ik} + d_{kj}$ that were computed during the previous iterations. Thus, after $O(|V|)$ steps, process ij has computed the shortest path from vertex i to vertex j that passes through zero or more intermediate vertices.

Van de Snepscheut's algorithm [16] is a processor-array implementation of the Parallel Floyd-Warshall algorithm. Logically, the processors, or more generally, processes, operate in synchrony: Each step consists of receiving zero or more messages, performing local computation, and sending zero or more messages to neighboring processes (at most one per neighbor per step). The algorithm performed is the same as that described immediately above. However, a clever timing strategy by van de Snepscheut yields an elegant implementation.

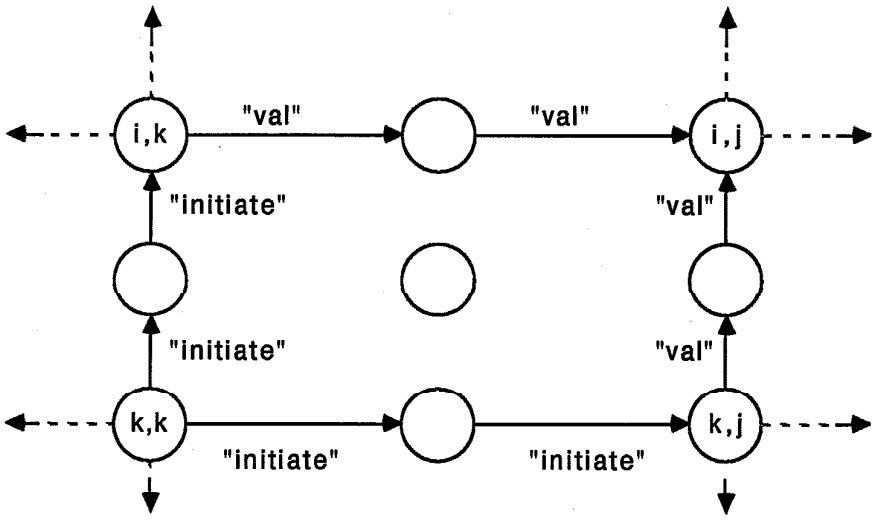

Figure 3.2: Message-passing protocols in Warshall APSP

For the k^{th} step of the computation, process *ij* assigns

$$d_{ij} = min(d_{ij}, d_{ik} + d_{kj}).$$

To perform this operation, however, process *ij* must contain the previous step distance values from processes *ik* and *kj*. Since in general the processes are not in direct communication, these values must be transmitted through the mesh of processes. A well-designed solution would have the d_{ik} and d_{kj} values arrive simultaneously to process *ij* so that storage of values within the process is not required. Van de Snepscheut employs the synchrony of a processor array to achieve this result.

If processes with indices *ik*, *ij*, and *kj* are three corners of a rectangle within the mesh, the fourth corner is represented by process *kk* (Figure 3.2). For the k^{th} step of computation, process *kk* sends initiate messages to its four neighbors in the k^{th} row and column. A process that receives the initiate message via a row sends its distance value out along its column, and *vice versa*. Thus, distance values from processes *ik* and *kj* are sent towards process *ij*. Since the values have an equal distance to travel and since the processor array ensures the same rate of progress, the distance messages arrive at process *ij* simultaneously. Of course, the initiate message is not exclusively directed at producing data for a single process. The initiate message sent by process *kk* as it travels through the k^{th} column and the k^{th} row prompts d_{ik} and d_{kj}, respectively, to be sent toward processor *ij*.

Even more concurrency becomes available with the realization that phases of computation can be pipelined within the mesh. A start message is propagated through the mesh; when it is received by elements on the diagonal, it initiates phases of computation. For all elements, a single phase of computation is marked by receipt of two val messages. According to van de Snepscheut, successive phases of computation must be separated by

at least three time steps. This requirement is satisfied by routing the start message from process *kk* to process *(k+1)(k+1)* via an intermediate process (*eg*, *k(k+1)*).

Thus, it is the synchronous nature of the processor array that makes possible this elegant solution. Our asynchronous model permits few assumptions about the relative timing of events in the algorithm. In the solution above, only one message can be transported over a connection during a time step. The fine-grain multicomputer model assumes infinite buffering between processes. Given the dissimilarity of models, the use of Cantor to implement van de Snepscheut's algorithm underlines some assumptions made in the programming model.

The Cantor implementation of the algorithm is presented in Appendix A.2.

As mentioned in the introduction to Cantor programming, a computation begins with the execution the main object, which is, at that point, the only object in existence. From the main object, the object graph required in the computation can be constructed either explicitly in the program or by employing the *custom object* facility discussed in Chapter 2. In the program above, the mesh of cell objects that emulate a processor array is created explicitly to illustrate a technique for object graph assembly. If a custom object were to be used, the call to the setup function and nearly half of the cell object definition would be eliminated.

Assembly of the object graph begins when the setup function creates $|V|$ cell objects, each representing the first element in a row of the mesh. Each cell object then creates another in the same row until the mesh is completed. The last cell object in each row replies to a branch of a synchronization tree, composed of sync objects, that responds to the main object when all $|V|^2$ objects have been created.

Since the function of the cell object array is to simulate the behavior of a processor array, each object must be able to communicate with its four neighbors in the mesh. The reference values of neighboring objects to the west and east are computed during mesh creation, but the reference values of the north and south neighbors must be obtained explicitly. This is accomplished using a "stitching" message pattern (*cf.* Figure 3.3). Cell objects located on all rows (except the last one) receive up messages containing reference to the cell object in the same column in the next row, its south neighbor. In response to this message, the cell object sends a down message, which contains its own reference and a reference to its east neighbor, to the south cell object. Receiving a down message prompts a cell object to send an up message to the northeast cell reference. When all the up and down messages have been observed, each cell object possesses the references of its neighbors in the mesh. The objects in the last column of the mesh again respond to sync objects to signal termination of the computation phase. The setup function returns a vector containing the reference values of the first object in each row of the mesh to the main object.

As mentioned in the general discussion of the algorithm, the synchrony of the processor array supplies timing information that is used to simplify the code executed by

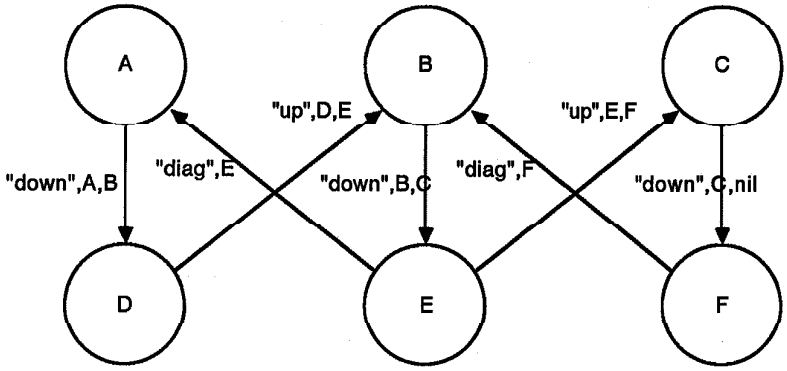

Figure 3.3: Establishing reference connectivity in a mesh using a "stitching" message pattern

a process in the mesh. Due to the asynchronous nature of the programming model, additional constraints must be added to the Cantor program to ensure correct behavior.

For instance, the start message cannot simply be routed to sufficiently separate successive phases of computation. Delay in message delivery between a pair of objects potentially could cause a start message to be received before the object has completed the previous phase of computation. The successive phases of computation then interfere and incorrect results may be calculated. For proper operation, cell *kk* must have received *k-1* pairs of val messages before initiating the k^{th} phase. Consequently, in the Cantor program, an internal variable phase, initially 1, is incremented for each pair of val messages received. When phase is equal to *k*, cell *kk* starts a new phase by sending itself a start message.

Another aspect of the algorithm that is underconstrained in an asynchronous environment concerns the init messages. For cell *ij*, $i < j$, an init message is received traveling east along the i^{th} row; later another is received traveling north along the j^{th} column. Cell *ij* below the diagonal of the mesh, $i > j$, receives init messages in the reverse order, traveling in the south and west directions, respectively. In the original formulation, the synchrony of the processor array ensures that, before receiving either of the two init messages, cell *ij* above the diagonal will have processed the *i-1* previous phases of computation. Cells *ij* below the diagonal will have already processed *j-1* pairs of val messages. In the asynchronous version, delays between cells could allow init messages to be received prematurely. If these messages are processed immediately, incorrect values of length might be sent to other cells in the mesh. So, when an init message is received by cell *ij*, if $phase < i < j$ or $phase < j < i$, the init message is not processed immediately. The direction of the message is preserved in initdir. When the proper number of pairs of val messages has been received, the init message is re-transmitted.

The constraints on start and init messages that were applied in the Cantor program solve the problem of skew of computation phases in the mesh. However, handling of val messages requires additional constraints because of the Cantor assumptions about buffering between objects. In the Cantor programming model, a potentially infinite number of messages may be in transit between objects in direct communication, *ie*, one possessing a reference to another. In the Cantor program, as many as N (where N is $|V|$) val messages may be in transit between cells. In addition, val messages traveling toward a cell from another direction may be arbitrarily delayed. Thus, a cell may have to examine and store as many as N val messages received consecutively along a row or column before a matching val message is received along a column or row, respectively. The premature val messages must be stored in a FIFO data structure, in this case, a queue of qelement objects.

When a val message is received, if one has already been received traveling in that direction (either along the row or column), the new one is made the "tail" of the appropriate queue, either ROW or COLUMN. Note that only two queues are necessary to store val messages from the four directions because of the constraints on the start messages. All val messages traveling along one of the directions of the row or column will be received and processed before subsequent phases of computation prompt val messages traveling in other direction of the row or column.

If the newly received val message is the first to be received from that direction, then the other queue must be checked for previously received val messages. If val messages have been received along the other direction, the pair of val messages is processed. A shorter path has been found if the sum of the distance values contained in the messages is less than the current length. The value of length is modified accordingly. In any case, the two val messages are then propagated in their original direction of travel so they can be used to compute paths for other cells in the mesh. Since the other queue may contain additional val messages that have been received, the head of that queue is accessed and advanced using the function capability of Cantor 2.2.

Termination of computation is detected by the main object counting the number of val messages it receives from cell *11*. Since that cell is the last to compute a length value on the N^{th} phase, and since it does not compute a length value on the first phase, processing terminates when the number of val messages received by main is N-1.

This program illustrates that the synchrony of a processor array can be successfully emulated in Cantor. The timing information available in the processor array is replaced by adding local constraints to the behavior of the individual parts of the mesh. Using the message-order-preservation property and queues maintained by the receiving process, the strict message protocols of the channels in the processor array are relaxed.

The property of message-order preservation between objects in direct communication was used implicitly throughout the program. Current versions support preservation of message order between objects in direct communication. This assumption is not unrea-

sonable given the current fine-grain machine model. In the model, oblivious, dimension-order routing between physical nodes is provided over FIFO channels. Message-order preservation is thus a by-product. In future machine models, however, adaptive routing will probably be used, which has the potential of destroying message order. In most programs, preservation of message order will not be a critical concern. However, for implementation of structures, such as processor arrays, that "pump" data in a systolic fashion through some network, message order is an important issue. Having each individual object handle the manipulation of incoming messages to restore order hardly seems a fine-grain solution. More likely, future Cantor systems will, in addition to "shipping" messages where arrival order is unimportant, provide a facility for sending messages to a destination with order preserved.

3.2 Quickhull

Since the advent of concurrent programming in the early 1980s, some types of algorithms have traditionally proved more amenable to concurrent execution than others. One important such class contains divide-and-conquer algorithms. The reasons for the efficient mapping of these algorithms onto concurrent machines are clear: Concurrent activities are defined at a coarse grain level by the algorithm; those activities, in general, do not need to communicate. Although Cantor is based on a fine-grain model of computation, thus encouraging frequent communication via message passing, divide-and-conquer algorithms are implemented in Cantor cleanly and efficiently. This study of concurrent programming in Cantor would not be complete without an example of the divide-and-conquer strategy.

The Cantor program discussed in this section finds the convex hull of a set of points. Briefly stated, the convex hull of a set of points is the smallest convex polygon that surrounds all the points. The convex hull thus defines the natural boundary of a point set. Clearly, members of the original point set make up the vertices of the convex polygon that defines the hull.

Since the convex hull of a set of points is useful in a variety of important application areas, several efficient algorithms have been developed to compute it. One interesting algorithm is the QUICKHULL algorithm [13]. Convex hull algorithms historically have been inspired by sorting techniques due to the similar nature of the problems. In the case of QUICKHULL, the sorting strategy of QUICKSORT [11] is adapted to the convex hull problem. Rather than a set of numbers being recursively partitioned into subsets of the sorted list, the set of points is recursively partitioned into subsets, each containing the vertices of part of the desired convex polygon.

The original set of points is divided into two subsets (Figure 3.4), S_1 and S_2, by the line *lr* which connects the points with the smallest and largest abscissa. Each step

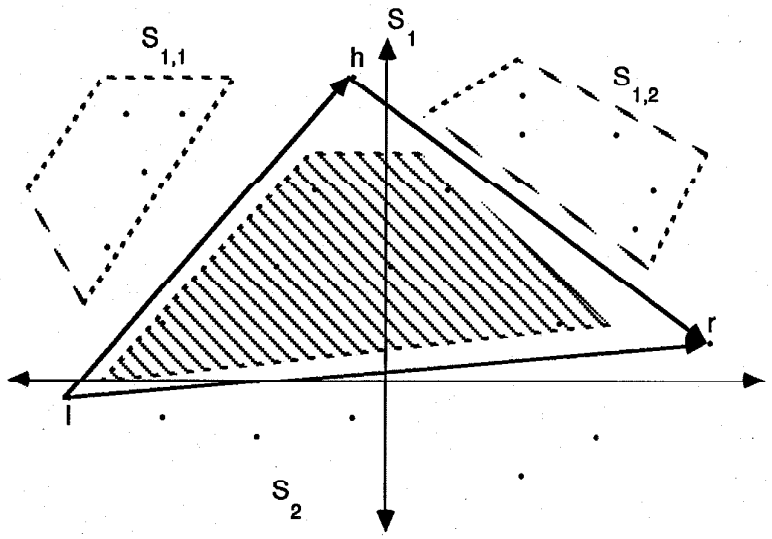

Figure 3.4: Example graph for Quickhull algorithm

of the algorithm performs the following procedure: From the points in S_1, a point *h* is determined such that the triangle (*hlr*) has maximum area over all other triangles (*plr*) where $p \in S_1$. If there are multiple points that satisfy this criterion, then the leftmost of those points is selected, *ie*, *h*, such that $\angle(hlr)$ is maximized. A key observation is that the point *h* then must belong to the convex hull. Next, two lines are constructed: L_1, from *l* to *h*, and L_2, from *h* to *r*. Each point in S_1 is then tested relative to these two lines. Each point must lie either on or to the left of L_1, to the right of both L_1 and L_2, or on or to the left of L_2. If the point is in the interior of the triangle *(lhr)*, *Ie*, to the right of both L_1 and L_2, then clearly it cannot lie on the convex hull. The points on or to the left of L_1 form a point set, $S_{1,1}$, just as the points on or to the left of L_2 form a point set, $S_{1,2}$. These sets are then passed to the next level of recursion.

Just as in the QUICKSORT algorithm, the time complexity of the QUICKHULL algorithm is dependent upon the relative sizes of the partitions at each level of recursion. If, at each level, the point set S_i is divided into two subsets of size at most a constant fraction of the size of S_i, then QUICKHULL has time complexity $O(N \log N)$. Just as in QUICKSORT, however, for the worst-case partitioning, QUICKHULL demonstrates $O(N^2)$ time complexity. (For a full discussion of the QUICKHULL algorithm, see [13].)

The Cantor implementation of the QUICKHULL algorithm is presented in Appendix A.3.

In designing the Cantor solution to the convex hull problem, the key decisions con-

cerned the representations of the point set and the convex polygon. Using a vector to represent the point set would make the step of dividing the set trivial. As usual, however, the disadvantage of this approach is that for reasonably large point sets, use of a vector to represent that set does not qualify as a fine-grain solution. For large amounts of data, using a chain of objects is the preferred solution. For the representation of the convex polygon, a vector containing vertices of the polygon is used. Although another chain of objects should be used to represent this list of vertices, a vector was chosen for simplicity.

The main object begins the computation by initializing two vectors, pointsx and pointsy, to represent the x and y coordinates, respectively, of the members of the point set. The chain of objects representing these points is then created tail-first. At the tail of the chain is a nilpoint object. This object's function is to initiate a new phase of computation after the latest phase has been completed by all objects in the chain. The other objects in the chain are created using the pointsx and pointsy vectors. Note that since the chain represents a set of points, the order of points in the chain is unimportant.

Once the chain has been created, the point with the smallest abscissa in the point set, *l*, is determined using the indexmin function. In the QUICKHULL algorithm described above, the initial step begins with the point set already divided into two subsets about the line *lr*, where *r* is the point with the largest abscissa. Rather than implement this special case step, *lr* is initially defined as the vertical line passing through *l*. The point *r* is then defined as some point that is distance ϵ from *l* along this line. The QUICKHULL computation is initiated by a call to a quickhull object. The Cantor call statement instantiates an object (or function) and then discards the reference value (or return value).

The quickhull object sends a message containing the coordinates of *l* and *r*, and the initial zero index and coordinates of *h* to its list of points to determine the actual point *h* defined above. The reply from the chain of objects is either an edge message or an h message. The edge message denotes that the set of points represented by the object chain defines an edge of the desired convex polygon. The quickhull object responds to this message by sending a vector containing its left value, the leftmost point lying on that edge, to replier, an object that collects and concatenates polygonal chains. An h message, containing the index and coordinates of the point *h*, causes the quickhull object to initiate another level of recursion. A collector object is created to collect and concatenate the polygonal chains that will be computed by the subsequent steps of the QUICKHULL algorithm. As the current set of points must be divided into two subsets as described in the general discussion of the algorithm, left and right lists consisting of nilpoint objects are created. An h message containing reference to these two lists is sent to the chain of objects so that each member of that chain will be added to the proper list (if any). When the reply from the list of objects is received, the enclosed references to the chains of objects representing points to the left of *lh* and to the left of *hr*, respectively, are used

to call new quickhull objects. Since a collector object has been created to rendezvous with the reply from recursion, the quickhull object then self-destructs.

The operation of the collector object, as mentioned before, is to collect and concatenate polygonal chains. Since each recursive step of the QUICKHULL algorithm divides a set of points into two subsets, the collector object will receive two rendezvous messages, each containing a vector representing the vertices of that chain. The messages also contain a tag to indicate that the chain is either the first or the last part of the composite polygonal chain. That composite chain is then sent to the parent of the collector object.

Since only the individual members of the object chain contain information about points in the point set, the point object definition performs the arithmetic computation related to determining the convex hull. Upon receiving a message containing the current indices and coordinates of *h*, *l*, and *r*, the point object checks the point represented by index against the current value of *h*. As outlined in the general discussion of the algorithm, the point *h* is chosen to be the vertex that maximizes the pair (area $\triangle(lhr)$, $\angle(lhr)$). If the point index, chosen as the point *h*, produces a higher value for this pair than does the current value of h, then h is assigned the value of index and the values for pairs corresponding to the remaining points in the set are compared to the values for point index.

It is important to note that the actual values of area $\triangle(lhr)$ and $\angle(lhr)$ are not required, only that some metric defines the same ordering of the points. In the point definition, the value returned by the function cross is used as the metric for the area of $\triangle(lhr)$. This substitution is justified by the fact that the area of the triangle is maximized for the point (or points) farthest away from the line *lr*. The unsigned cross product of the lines *lh* and *lr* produces the value of the height of the triangle (*lhr*) multiplied by the magnitude of *lr*. Since the line *lr* is constant for each point set, the absolute value of the value returned by the function cross, *ie* the unnormalized cross product, is an acceptable metric for the area of $\triangle(lhr)$. Similarly, the metric for the $\angle(lhr)$ is obtained by dividing the unsigned cross product of *lh* and *lr* by the magnitude of *lh*. This value actually represents the sine of the angle between *lh* and *lr* multiplied by the magnitude of *lr*. Since the angles between these lines will always be less than π, the sine of the angle multiplied by a positive constant will suffice as the metric for $\angle(lhr)$.

If the point object determines that the values of the functions cross and angle for the point index exceed the metric values maxcross and maxangle for the current value of h, it replaces the value and metric values of h with those of index. The current value and metric values of h are then sent to the next point object in the chain.

This process continues until each point object in the chain has compared its point index against the current value of h. Thus, when the message containing h reaches the nilpoint object, h has been determined. If h is zero, indicating that no point in the point set lies off the line *lr*, then an edge of the convex polygon has been identified, so the edge message is sent to the quickhull object requester; otherwise, the point set represented by the

chain must be divided as described in the discussion of the algorithm. The nilpoint object sends a h message to the quickhull object requester to initiate this phase. Upon receiving this h message, each point object in the chain calls the function outside to determine if the point index lies to the left of *lh* or of *hr*.

The function outside simply returns the sign of the cross product of the line connecting (ax,bx) and (x,y) and the line connecting (ax,ay) and (bx,by). If that sign is negative, then the point (x,y) lies outside, or to the left, of the line connecting (ax,ay) and (bx,by).

So, in the point object definition, if the point index lies outside the line *lh*, the point object adds itself to the left list of points. Likewise, if the point lies outside *hr*, the object is added to the right list. Since the chain of objects represents a set of points, the fact that the order of the points in the newly created lists is reversed from the current order is unimportant. The h message is then sent to the next object in the chain.

When the original point set has been completely divided into chains representing edges of the convex polygon, the collector tree will complete the concatenation of lists of vertices representing those edges and eventually reply to the main object. The main object then prints the vertices of the convex polygon in order of traversal.

In addition to the implementation of the divide-and-conquer strategy, this program exhibits several important characteristics of fine-grain programming: First of all, the representation and manipulation of a set using objects rather than an internal data structure, such as a vector, are critical fine-grain programming techniques. Although the specification of the behavior of the object structure is more complicated than that of a vector, the distributed structure will scale to the size of the fine-grain machine, while the size of internal structures would likely be severely limited.

Another reason this program was selected for inclusion in the study lies in its use of functions. This program exemplifies a major purpose of the addition of functional abstraction to Cantor. Without functional abstraction, in-line code would be used to perform the operations of cross, outside, angle, and indexmin. As object definitions become longer and more complicated, this type of code replication becomes less acceptable. Functional abstraction was included in Cantor 2.2 in part to enable the programmer to define functions to simplify object definition code. It should be noted that object instantiation, not functions, are used to implement the recursion of the QUICKHULL algorithm. The use of recursive functions cannot be reconciled with the fine-grain programming model.

3.3 Enumeration of Paraffins

The study of organic chemistry was one of the first application areas of graph theory. In 1874, Cayley [4] used tree theory to enumerate saturated hydrocarbons. Hydrocarbons, as implied by the name, are constructed from carbon atoms, which have valence 4, and hydrogen atoms, which have valence 1. In a saturated hydrocarbon, all bond positions

are occupied. Saturated hydrocarbons, or *paraffins*, share the empirical formula C_nH_{2n+2}.

As n, the number of carbon atoms, increases, the number of paraffin isomers (distinct molecules with the same formula) increases dramatically. Since the four bond positions of a carbon atom are indistinguishable, enumerating paraffin isomers only entails counting structures that are not reflections or rotations of other structures. Paraffins with five or fewer carbon atoms are shown in Figure 3.5. While enumerating paraffin isomers for a given n through combinatorial analysis would be an interesting exercise, enumeration by generating a list of the isomers proves to be a challenging fine-grain programming experiment.

Two approaches can be taken to this enumeration problem: Either generate all structures that represent paraffins, and by testing for isomorphism, eliminate duplicate structures; or, generate only those structures that are unique representations of paraffins. As testing for graph isomorphism in all the possible structures would be quite time-consuming, the second approach is preferable. The key difficulty in the second approach lies in the need for a canonical representation for each molecule. One such representation can be found by employing graph theory to solve an equivalent problem.

To represent each paraffin molecule, Cayley used a tree with $3n + 2$ points, where each point has degree 1 or 4. However, since all the bond positions are occupied, the structure of the carbon atoms completely specifies the structure of the molecule. Thus, a tree representing the carbon atoms suffices. This tree representing the carbon atoms is an unrooted, or *free*, tree. Enumerating paraffin isomers of n carbon atoms is therefore equivalent to enumerating free trees of n vertices with degree less than or equal to 4. Of course, the difficulty in enumerating the free trees lies again in the lack of a canonical representation.

According to Knuth's discussion of free trees [12], it is possible to select any vertex X of a free tree and, by assigning directions to edges in a unique way, make vertex X the root of a rooted, or *oriented*, tree. Let k be the number of subtrees of the root X. Let s_i be the number of vertices in the i^{th} subtree, where $1 \leq i \leq k$. So for a given number of vertices, n, $k \leq 4$ and

$$s_1 + s_2 + s_3 + s_4 = n - 1. \tag{3.1}$$

The *weight* of vertex X is defined as the maximum value of some s_i. A vertex of the free tree that has minimum weight is the *centroid* of the tree. Knuth proves that if vertex X is the only centroid of the tree, then

$$s_j \leq s_1 + s_2 + s_3 + s_4 - s_j, \tag{3.2}$$

where $1 \leq j \leq k$. Knuth also proves that *there are at most two centroids in a free tree, and if two centroids exist, they are adjacent.* A free tree with two centroids has an even number of vertices, and the weight of each centroid is $n/2$.

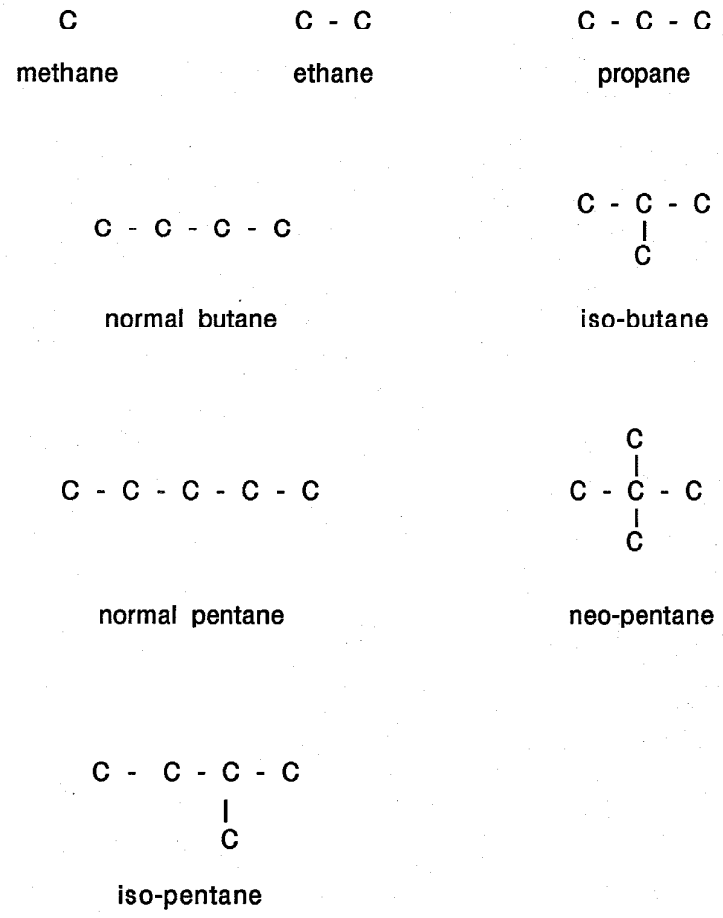

Figure 3.5: Carbon atom structure of some paraffin molecules

For the single centroid case, a rooted tree with at most 4 subtrees represents a free tree if and only if those subtrees satisfy equation 3.2. However, not all rooted trees satisfying that equation are distinct. For example, a rooted tree with $s_1 = 2$ and $s_2 = 1$ is isomorphic to another rooted tree with $s_1 = 1$ and $s_2 = 2$. Adding the condition

$$s_1 \geq s_2 \geq s_3 \geq s_4 \tag{3.3}$$

ensures that a rooted tree represents a unique set of free trees.

Note that the four-tuple (s_1, s_2, s_3, s_4) represents more than one free tree if any $s_i > 2$. Each of the subtrees of vertex X are rooted trees containing s_i vertices that do not necessarily satisfy equations 3.2 and 3.3. For $s_i > 2$, there is more than one such rooted tree of s_i vertices. The total number of ways to combine non-isomorphically the rooted trees of size s_1, s_2, s_3 and s_4 is the number of free trees represented by the tree rooted at vertex X.

For the double centroid case, a rooted tree is unique if it is composed of two rooted trees, each containing $n/2$ vertices, that are combined in non-isomorphic ways. That is, if there are p rooted trees of size $n/2$, these subtrees can be combined in $\frac{1}{2}p(p+1)$ non-isomorphic ways, producing that number of distinct free trees of size n.

From this discussion, we derive an algorithm for generating all free trees of n vertices.

First, generate all four-tuples (s_1, s_2, s_3, s_4) that satisfy 3.2 and 3.3. Each of these tuples represents a set of unique free trees of size n with single centroids. In addition, if n is even, investigate bicentroid cases by non-isomorphically combining rooted trees of size $n/2$.

Since $0 \leq s_i \leq n/2$ for all four-tuples generated above, all rooted trees with m vertices, $1 \leq m \leq n/2$, will be required to generate the free trees. To generate rooted trees of size m, generate all three-tuples (t_1, t_2, t_3), where $0 \leq t_j \leq m-1$ and

$$t_1 + t_2 + t_3 = m - 1 \tag{3.4}$$

Combine non-isomorphically the sets of rooted trees corresponding to the value of each t_j.

The concurrency available in this algorithm is not trivial, as it would have been in the first approach to the problem, which generated all structures corresponding to free trees and then tested for isomorphism. In this formulation, completion of generation of rooted trees of size m requires completion of generation of rooted trees of up to $m/2$ vertices. Concurrency is obtained by having each tuple (either s_i or t_j) be computed independently of other tuples corresponding to trees of size m. When the four (three) lists of rooted trees corresponding to s_i (t_j) are completed, those trees can be immediately combined non-isomorphically. Thus, as n and the number of paraffin isomers increases, there is more opportunity to perform parts of the computation concurrently.

The Cantor program implementing this algorithm is shown and discussed in Appendix A.4.

The main object begins the generation of paraffin isomers of size N by instantiating a rooted object to generate rooted trees of size 1. These rooted trees will be used in generating rooted and free trees of size greater than 1. If $N = 1$, a free object is created to generate free trees of size 1.

The index of a rooted object is the number of vertices in the rooted trees it will generate. The index thus corresponds to m in the general discussion of the algorithm. A newly created rooted object with $index < N/2$ creates another rooted object that will compute the rooted trees of size $index + 1$. A rooted object with index equal to $N/2$ creates a free object to combine rooted trees to form the free trees of size N.

The chain containing rooted objects and the free object is thus created sequentially. When a rooted object is created, a list of reference values of rooted objects with lower index values is sent to the new end of the rooted object chain. Reference values to rooted objects with lower index values are required because rooted objects computing trees of size m will need to request the lists of trees of size up to $m/2$.

The chain containing the rooted objects and the free object is not required to be complete before each object can begin to generate trees. After a rooted object has created the next object in the chain, it generates a set of tuple objects, one for each tuple (t_1, t_2, t_3) satisfying equation 3.4. The tuple objects are connected in a ring with the rooted object. The first tuple has a next reference to the rooted object. Other tuples have a next reference to the previously created tuple. After creating all the tuples, the rooted object begins accumulating the trees that correspond to each tuple by sending a solutions message around the ring of tuples. When a tuple has finished generating the trees corresponding to its three-tuple (t_1, t_2, t_3), and has received a solutions message, the list of rooted trees satisfying the three-tuple is added to the rooted-trees list received from other tuples in the ring. Thus, when the rooted object receives a solutions message, the generation of trees of size m has terminated. Since rooted objects operate concurrently, request messages from other rooted objects may be received before a solutions message. These requests are queued via a LIFO of stackelement objects. When the solutions message is received, the requests represented by the stackelements are satisfied. Subsequently received request messages are satisfied immediately by the rooted object.

The synchronization mechanism of the ring, which is a derivative of the termination-detection scheme of the Chandy-Misra SPSP program, is used extensively in this program to collect information upon completion of some set of concurrent tasks. This paradigm is also used throughout the program that is the remainder of this chapter.

The behavior of the free object is essentially the same as that of a rooted object. Again, the objects generating lists of trees are organized in a ring. If N is even, the first object created in the ring is a tuple object that generates the bicentroid free trees by combining pairwise rooted trees of size $N/2$. Additional tuple objects are created for every four-tuple (s_1, s_2, s_3, s_4) that satisfies equations 3.2 and 3.3. In this case, the ring responds to the main object rather than to the free object because when the objects in this

ring terminate processing, the generation of free trees of size N has been completed, and so the main object should be notified of termination. The free object begins accumulating a list of the free trees by sending a solutions message to the ring.

All lists of trees, either free or rooted, are represented by linked lists of tree objects. These objects represent trees by containing reference values of other tree objects that represent the subtrees of size i, j, k, and l. Thus, these reference values are, in effect, pointers to particular subtrees. This approach to storing the generated trees minimizes the amount of memory required to represent the total set of generated trees. The details of tree-object operation will be described later.

The persistent variables of a tuple object include i, j, k, and l, which are respectively the number of vertices in each of the subtrees of the trees to be generated by the object. The tuple object must collect and combine the rooted trees of sizes i, j, k, and l. For this reason, the reference values of the rooted objects that generate those trees are also contained in the persistent variable list.

Another ring structure is used to gather and combine the lists of subtrees. The first object created in the ring is a product object; this will respond to the tuple object. The tuple object then creates a child object for each of the four variables to request and collect the lists of subtrees from the rooted objects. The tuple object sends a solutions message to the ring to begin accumulation of the lists. When the list of trees corresponding to the tuple i, j, k, and l is compiled, the product object then responds to the tuple with a reply message containing a reference to the head of the list of trees. When both the reply and the solutions messages have been received, the two contained lists of tree objects are concatenated and the tuple object sends the combined list of trees to the next tuple in the ring.

A child object requests the list of subtrees from a particular rooted object. The rooted object will respond with a reply message containing the number of trees of size subtree and the reference value of the head of the list of tree objects representing the trees. Since the child objects in the ring are created by the tuple object in l-k-j-i order, when the solutions message is received, the child assigns the reference value of the received list of tree objects to its place in the vector Refs.

When collection of the lists of subtrees has been accomplished, the vector Refs will be sent to the product object, the first object created in the ring. This object instigates the combination of the lists of subtrees to produce a new list of trees that will be sent to the creator tuple object. If the Refs vector contains only nil references, a list consisting of a tree object, (representing a tree of zero vertices) and an endlist object is sent to the tuple object; otherwise, if non-empty lists are to be combined, a product message is sent to head of the list of tree objects representing rooted trees of size m.

The combination of lists of tree objects is probably the most complicated part of the algorithm. If the lists were always distinct, a distributed Cartesian product of the lists would suffice. Instead, the same list of tree objects may represent each of the subtrees

to be combined. Applying the Cartesian product operation to non-distinct lists would produce isomorphic trees.

The algorithm to combine the possibly non-distinct lists follows: Let S_1, S_2, S_3, and S_4 be the four lists of trees of size s_1, s_2, s_3, and s_4, as described in the general discussion of the algorithm. For rooted trees, S_1, S_2, and S_3 represent subtrees of size t_1, t_2, and t_3, while S_4 represents a subtree of zero vertices.

If the non-empty S_i lists are distinct, the Cartesian product of the lists is performed. For non-empty lists, each element of S_3 is combined pairwise with every element of S_4. Each element of S_2 is combined with every such pair to produce triples. Each element of S_1 is then combined with every triple to create the desired four-tuple unique trees. Naturally, if some S_i list is empty, fewer lists will be combined so fewer elements will be required to represent a tree.

If some of the S_i lists are not distinct, the above procedure must be modified. For $S_3 = S_4$, the first element of the list is combined with every element, including itself, in the list to form pairs. The second element of the list is combined with every element in the list *except the first element*. In general, each element of the list is combined pairwise with elements occurring equal to or later in the list than itself. Let this process be called the *expansion* of an element of the list. If $S_2 = S_3 = S_4$, triples are formed by combining each element of the list with pairs containing elements of the list occurring equal to or later in the list than that element. A similar operation produces four-tuples that represent unique trees if $S_1 = S_2 = S_3 = S_4$.

In the Cantor version of this algorithm, the first element of S_1 is sent a product message. This message contains references to the lists to be combined in the vector Refs, and begins the expansion of the receiving tree object. This tree object, after some processing that will be described below, sends a product message to the first element in S_2 and waits for a reply message containing the list of combinations of elements in the remaining lists. The tree object in S_2, upon receiving a product message, begins its own expansion by passing the product message to S_3, and waiting for a reply message. This process continues until the first element in the last non-empty list receives a product message.

As part of the product message, a tree object receives the vector replier, containing the reference values for the destinations of reply messages; the vector head, containing reference values of the first elements of the lists being combined; depth, the index of the list being accessed; num, the number of unique trees already generated for the parent tuple object; and list and tail, respectively, the head and tail of the list of trees generated thus far. Also included in the product message is a vector containing the reference values of elements currently being expanded and whether or not bicentroid trees are being generated.

Upon receiving a product message, a tree object assigns its own reference value to the $depth^{th}$ cell of Refs and begins the combination of lists of higher depth. If the reference

value of this tree object is also the reference value of the element to be expanded in the (depth+1) list, then the lists are clearly not distinct. Rather than send multiple product messages to itself, the tree object expands itself by incrementing the depth and assigning its self value to one element of the vector replier. This operation is performed as many times as there are instances of the non-distinct list.

If the tree object receiving the product message is an element in the last non-empty list being combined, *ie*, Refs[depth]+1 = nil, a new tree object is created to represent the unique tree made up of the subtrees represented by Refs[1..4]. The new tree object is added to the list of trees already generated corresponding to the original tuple. The product message is then sent to the next element in the list so that Refs[1..depth-1] can be combined with other elements of the list.

When a product message reaches the end of a list, the endlist object for that list sends the accumulated list of trees to the element in the (depth-1) list that was expanded with the product message. Upon receiving the reply message containing that list, the expanded tree object sends a new product message to the next element in the list at that level, thus beginning expansion of the rest of that list.

When a product message reaches the endlist object of the first list to be combined, combination of the lists has been achieved, and a list of the tree objects representing the unique trees is then sent to replier[1], the tuple object.

Although the algorithm used here to combine lists is sequential, many combining operations that uses the same lists may be occurring concurrently. Since the state of the combination is completely contained in the product and reply messages, pipelining of combination operations is trivial. A concurrent algorithm to combine the lists was investigated, but handling the resulting fragmentation of the list of generated trees proved to be quite a messy operation. For simplicity, the sequential version is presented here.

The complication of combining the lists of tree objects can be immediately resolved, however, for small N. If a vector is used to represent the set of tree objects, the combining of the lists can be performed within the product object. Of course, this way of combining the lists is much faster than the distributed version; however, as N grows, the size of the required vectors increases quite dramatically. For $N = 12$, the size of the vectors must be greater than 350. For $N = 15$, the vectors must contain more than 4000 reference values. Clearly for any but the smallest N, using vectors to represent the lists of trees is not a fine-grain solution to the problem. By using a distributed list of objects to represent the lists, the program's operation is limited only by the number and size of objects that can be created.

A table of the number of paraffin isomers for N up to 16 is shown in Figure 3.6.

Developing a solution for enumerating paraffins is perhaps the most challenging problem studied in this thesis. As the concurrency available in the problem was non-trivial, a fine-grain formulation of a solution was critical. Selection and manipulation of the data structures representing lists of trees was an integral part of that formulation. Future

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P	1	1	1	2	3	5	9	18	35	75	159	355	802	1858	4347	10359

Figure 3.6: Number of Paraffins P of N Carbon Atoms

work on the problem includes extensive profiling of the concurrency in this solution to further evaluate the programming techniques that were employed.

3.4 Checkmate Analyzer

As part of the quest for devices that can best human opponents in games of skill, chess-playing computers have received much attention. In general, the chess-playing programs select moves from a set of possible moves by using tree look-ahead. A tree is maintained with the root as the current configuration of the board. At the next level, a vertex represents the state of the board if some candidate move were to be taken. The number of levels available in the tree defines the number of moves the program can "look-ahead." Operating on the premise that the opponent always makes the best move, the program chooses its own moves by evaluating the deepest levels of the subtrees emanating from the vertices representing the possible moves. Since a high degree of concurrency is available for generating and scoring the subtrees, and since the amount of state required by each vertex is relatively small, a multicomputer is a logical choice as a target machine for chess-playing programs.

A Cantor program that exploits the available concurrency in chess-playing is planned, but is well beyond the scope of this thesis. As a first step in investigating programming techniques that would facilitate a fine-grain solution to playing chess, a Cantor program is developed and presented here to detect whether or not checkmate has occurred. Given a legal board configuration and an indication of the side whose turn it is, the Cantor program analyzes the situation and outputs whether or not the king is in check, and provides a list of moves that would remove the check on the king. If no such moves exist, checkmate is reported.

As in most non-trivial fine-grain programming experiments, achieving high performance requires a fine-grain algorithm. Formulating the algorithm at some coarse-grain level of concurrency and then attempting to exact fine-grain performance is not generally successful. In the case of the checkmate analyzer, several design decisions were predicated upon knowledge of fine-grain programming paradigms and techniques.

The first of these decisions concerned the representation of the configuration of pieces on the playing board. Both piece-oriented and board-oriented approaches were considered. In the piece-oriented approach, objects representing the playing pieces are the chief computing agents, and each keeps track of its piece's current position on the board. In

the board-oriented approach, objects representing each square of the playing board keep track of the current resident status of that square, whether empty or occupied by one of the 32 playing pieces.

In comparing these two approaches, one notices that the first approach requires half the number of objects as the second one. However, when target machines support potentially hundreds of thousands of concurrent objects, the advantage of saving 32 objects is negligible. In this case, the choice between the approaches illustrates one of the important lessons of fine-grain programming. In the piece-oriented approach, the number of objects is fewer, but the amount of communication between objects is greatly increased. With no global knowledge about the configuration, the object representing a piece must in essence broadcast information to all other piece objects. In a board-oriented approach, objects representing squares possess local information about the squares around them. So, in this case, minimizing the number of created objects would increase communication overhead and decrease useful concurrent activity. For this reason, the board-oriented approach was selected and implemented.

Another design decision focused on the methods of communication between playing pieces. For instance, how does one determine if the king piece is in check? If playing pieces communicate by comparing their coordinates, then communication between all playing pieces is potentially required. The number of extraneous messages in this scenario makes this approach unattractive. Another approach would be to have playing pieces attempt to "hit" the king piece with messages that traverse the same path to the king as the piece would. For example, a object containing a bishop piece would send messages along diagonal squares, perhaps encountering the king piece. The disadvantage of this approach is that all opposing pieces could potentially check the king piece. Is it necessary to have all those pieces send messages in an attempt to hit the king piece?

The chosen approach to communication between playing pieces is a derivative of the one presented immediately above. Rather than having opposing playing pieces send messages attempting to hit the king piece, the object representing the square with the king piece sends a message for each opposing piece in the possible directions of attack by that piece. For example, the king piece square object sends a message seeking the opposing king's bishop along all diagonal directions. If the king's bishop is hit by one of these messages, then the king piece is in check. The primary advantage of this approach over the other two is in the relative economy of message traffic. While sending many short messages is not a disadvantage in fine-grain programming, sending extraneous messages is rarely justifiable.

The implemented algorithm for analyzing the playing board configuration follows: Sixty-four objects are created to represent the squares of the board. Each individual square can communicate directly with each of the eight neighboring objects (including the four diagonal neighbors). The configuration of playing pieces is then mapped onto the object grid. The object representing the king piece of the color whose turn it is is

identified. Hereafter, the objects in the algorithm are identified by the playing piece they contain or as empty squares.

The king then begins to send messages to determine whether or not it is in check by opposing pieces. Concurrently, the king begins to investigate possible moves by the king to one of the surrounding eight squares. The standard rules of movement of the king are observed. When these two parallel activities are completed, the king is aware of its attackers (if any) and the feasibility of its moves to escape check.

If only one opposing piece is attacking the king, then the possibility exists for that piece to be either captured or blocked; these possibilities are explored concurrently. The attacking piece sends out messages to determine if it can be captured. If the attacking piece is a "line-of-sight" piece (a queen, bishop, or rook), then the empty squares between the king and the attacking piece send messages to determine whether they can be captured by a piece friendly to the king; at the completion of these two parallel activities, the king is aware of possible moves by friendly pieces to remove check.

However some of these moves may be invalid. For instance, a move to capture or block the attacker may expose the king to another attacker, leaving the king still in check. To identify the valid moves, a new board of objects is created for each of the potential blocking or capturing moves. The configuration of playing pieces on each of the boards corresponds to the configuration of the original board after each of the potential moves has been taken. Concurrently, each of these boards is analyzed to determine if the king remains in check. For each of the boards in which the king is not in check, a blocking or capturing move is reported.

The output of the checkmate analyzer consists first of the status of the king, *ie*, whether or not the king is in check. If the king is not in check, it is clear that checkmate has not occurred; if the king is in check, then a list of moves for the king piece to escape check is reported. A list of valid moves to block or capture an attacking piece is also reported. If the king is in check and no moves can be found to escape or remove check, checkmate is reported.

The Cantor checkmate analyzer is presented in Appendix A.5.

The first part of the computation involves the set-up of the board configuration to be analyzed. As in previous examples, external input capabilities are not employed. The board configuration to be tested is explicitly loaded in the row and col vectors. The index into the vectors is determined by the color of the piece and by its original position on the playing board, *eg*, king's knight. For the purpose of printing the configuration of pieces with the final output, identifying names for the playing pieces are loaded into the vector piecename. The configuration is then printed.

The stategy discussed previously for analyzing the board configuration uses a mesh of objects that represent the squares of the playing board. A call to a buildboard object establishes that mesh of objects.

Two synchronization chains are used to ensure that phases of the construction do not

become skewed. The childlist chain, composed of child objects, accumulates the reference values of the first square object in each of the eight rows. The synclist chain, composed of sync objects, detects when the all objects in the mesh have reached some synchronization point in the computation. After the buildboard object instantiates each of the objects in these lists, it begins instantiation of the mesh objects by assigning each element of the rows vector to a connectsquare object.

The function of the connectsquare objects is an example of the use of a temporary object structure to construct a more permanent object graph. The primary advantage of this approach is to minimize the amount of code within the permanent graph object by factoring that code into temporary objects. The primary disadvantage is that the total amount code is increased due to the added overhead of managing the state of both types of objects. In this case, each element of the mesh of connectsquare objects creates a square object and constructs a vector containing the reference values of all eight neighboring square objects.

The method to establish this reference connectivity is the "stitching" method discussed in Section 3.1.2 with the addition of diagonal reference connection. When the buildboard object is notified by the synclist that all connectsquare and square objects in the mesh have been created, it begins the linking phase by sending up messages to the first element of each connectsquare row. These up messages carry the reference values of the connectsquare and square objects in the same column as the receiver in the next row, *ie*, to the south. When the up message is received by a connectsquare object, the included reference values are preserved in the array dir and a down message is sent to the connectsquare reference just received. Down messages include reference values to connectsquare and square objects to the north, northeast, and northwest of the receiving object. These messages prompt the sending of more up messages and of diag messages. Diag messages are sent to northwest neighbors to establish diagonal connectivity.

Based on its coordinates in the mesh, a connectsquare object determines which of the three types of messages it will receive. When the required messages have been received, objects on the edges of the mesh create nilsquare objects to act as the perimeter of the mesh. The completed dir vector is sent to the associated square object. If that square of the playing board contains the king piece of the side whose turn it is, the reference of the square piece and the reference values of the neighboring squares are sent to the buildboard object. The connectsquare structure then self-destructs after synchronizing through the synclist. After receiving the king piece reference values, the buildboard object sends those values to the main object and then self-destructs.

The actual analysis of the board configuration begins when the main object receives the reference values of the king piece square and of the neighboring squares. At this point, the main object initiates a search for playing pieces that are attacking the king piece, *ie*, to detect if the king piece is in check. This search is instigated by a call to a notmoveking object. This object also manages the investigation of blocking or capturing

moves to remove the king from check. Concurrently, a movekíng object evaluates the king's neighboring squares as possible moves by the king piece to escape check.

The movekíng object actually serves only to create a synchronization chain of eight collectmoves objects. Each of these objects will rendezvous with a reply message from the neighboring square in one direction. The neighboring squares, upon receiving a check? message, investigate the possible attackers of that square. The reply message from the neighboring square contains the identity of the playing piece currently occupying the square, and the identity, reference values, and direction of pieces that could attack that square. The collectmoves chain constructs a list containing directions from the king square of squares that are safe from attack and do not contain a playing piece, or that have a playing piece that can be captured by the king. The chain of collectmoves objects returns a vector containing these directions to the main object in a movekíng message.

Meanwhile, the notmovekíng object waits to receive the report of attackers on the king square. If there are no attacking pieces, then the king is not in check and a notmovekíng message is sent to the main object. If there are multiple attacking pieces, then no blocking or capturing move will remove check, so messages are sent to the main object signifying that no blocking or capturing moves exist. If, however, there is only one attacker, that attacker potentially can be blocked or captured. If the attacking piece is a "line-of-sight" piece, *eg*, a queen, rook, or bishop, then a blockpiece object is called to investigate possible blocking moves; otherwise, the main object is notified that there are no blocking moves. Capturing moves are investigated by instantiation of a capturepiece object. The specific operation of these two objects will be discussed later.

As described in the general discussion of the algorithm, a square object seeking to determine if that board square can be captured by the opponent's pieces sends messages to neighboring squares seeking specific pieces. When a check? message is received by a square object, a checksquare object is created to manage this process. The checksquare object begins by creating a checkpiece object for each playing piece of the opponent's color. These sixteen objects are arranged in a synchronization chain to accumulate the replies of the inquiries to other squares on the board.

The role of the checkpiece object is to rendezvous with the replies from inquiries seeking a particular piece that have been sent in all directions of motion, and then to forward that information to other checkpiece objects in the chain. As the reader may suspect, another synchronization chain is used to accumulate the replies from all possible directions of motion of the particular playing piece. Each element of this chain is a missile object that is responsible for sending hit? messages in the given direction of motion in an attempt to find a particular attacking piece. When the reply from this inquiry is received, its information about the potential attacker is assimilated and sent to the rest of the chain in a hit message.

When a square object receives a hit? message, it first determines if the playing piece being sought is a knight. As the motion of a knight piece is exceptional, the messages

seeking that piece are also exceptional. Only one hit? message is sent by the checkpiece object in each of the four diagonal directions. The receiving square object detects that the message has been sent along a diagonal and then sends hit? messages out in directions that carry the messages to squares that could contain attacking knight pieces. For instance, if the hit? message is received traveling in the northeast direction, then the two new messages should be send out in north and east directions. If the piece being sought is not a knight, and the square object does not contain a playing piece, then if the piece being sought is a "line-of-sight" piece, it sends the hit? message out along the original direction. Otherwise, the square object replies as to whether or not it contains the sought playing piece.

In summary, square objects respond to missile objects with hit? messages containing information about attacking playing pieces. Missile objects assimilate information about the direction of attack of a particular type of piece. This information is sent in a hit message to a checkpiece object that is responsible for seeking a particular playing piece. The information from all checkpiece objects is collected and then sent to the object that called the checksquare object.

The capturepiece object uses this process to determine if the lone attacking piece can be captured by pieces friendly to the king piece. A check? message is sent to the square object that contains the attacking piece, with the color of attacking pieces reversed. Thus, if the white king is being analyzed for checkmate and a single black piece has the king is check, then the check? message seeks white pieces to capture the black attacker. If any capturing pieces are found, then a collectconfig object is created to test the potential board configurations that would result from that capturing move being taken. As mentioned in the general discussion of the algorithm, some of these moves to remove check from the king may expose the king piece to other attackers. Naturally, these collectconfig objects are arranged in a chain that will eventually respond to the main object.

The blockpiece object sends a block? message along the direction of motion of the attacking "line-of-sight" piece. Each of the square objects between the king and the attacker responds to the block? message by creating a collectblocks object to collect the reply from the checksquare object it calls to determine if that square can be captured by a friendly piece. The block? message is then continued in its original direction of motion. When the block? message reaches the square containing the attacking piece, that square object replies to the chain of collectblocks objects, beginning the collapsing of that chain. Eventually, this list replies to the blockpiece object with a list of potential blocking moves. As in the case of capturing moves, some of these blocking moves may be invalid. Again, a collectconfig object chain is created to test and collect each of the potential board configurations.

The collectconfig object tests the board configuration using a checkconfig object. The checkconfig object uses the buildboard object to create a mesh of objects representing the playing board configuration. When the reply is received from the object mesh, a check?

message is sent to the square object containing the king piece to determine if the king piece is still in check. The result of this inquiry, once received, is sent to the parent collectconfig object in a reply message. Upon receiving this reply message, the collectconfig object assimilates the result of that inquiry into the list of blocking or capturing moves (depending upon the type of moves this chain of collectconfig objects is collecting). When the chain collapses, it responds to the main objects with a list of blocking or capturing moves.

The main object in total receives a moveking message and either a notmoveking message or else a block message and a capture message. When one of these sets of messages has arrived, the following information is reported: whether the king is in check, and, if it is in check, lists of moves by the king piece to escape check and moves by friendly pieces to capture or block an attacking piece.

This program to analyze chess board configurations is the largest Cantor program written thus far. Its contribution to the Cantor library is that it non-trivially uses programming techniques that were developed in smaller, less-involved programs. For example, a technique called *code factoring* is used extensively in this program to reduce the complexity of an object's code. Without using discretionary receipt of messages, an object's code must provide a response for each potential type of message that could be received. Without code factoring, the potential number of messages could be quite large, thus requiring a large case statement to specify the object's response. Many times, however, the response of a particular object to a message can be factored into another object. For instance, in the square object code, receipt of a check? message triggers the initiation of an inquiry concerning the status of that board square. If the reply from this inquiry were to come back to the square object, the case statement that is the majority of that object's code would be even larger. Likewise, the reply from the block? inquiry would increase the size of the case statement. A simple, but powerful, technique is to create a new object specifically to handle the rendezvous with the reply from the inquiry. In the case of the block? inquiry, a synchronization chain of collectblocks exists for this purpose. Throughout this and the other programs, this technique is used to minimize the increase in complexity of object definitions introduced by the lack of message discretion. This technique also yields potentially better performance as the number of messages in a single object's message queue is reduced and the number of objects that can execute concurrently is increased. The synchronization structures that have been presented are examples of the usefulness of the code-factoring technique.

This program also illustrates the use of temporary object structures in the generation of permanent object structures. The original intent for using such ephemeral structures was to separate the concerns of creating an object structure versus performing the actual computation. Although this goal is an important one, the practicality of using temporary object structures to achieve it is unproven. Most often, use of temporary object structures adds significantly to the size and complexity of the object definitions for the

temporary and permanent objects. In future work, alternative methods for establishing a computation graph, including methods that free the programmer from explicitly constructing highly regular object graphs, will be examined.

Chapter 4

Conclusions

The approach to this thesis experiment was outlined in Chapter 1. By writing Cantor programs to perform a variety of tasks, and then studying those programs, we gain insight into the fundamentals of fine-grain programming. Some of the interesting programs written as part of this experiment have been presented and discussed in Chapter 2. To conclude, we answer the question, *"What did we learn from these programming experiments?"*

4.1 About Cantor

These programming experiments were in part designed to provide evidence, either supporting or refuting, for assumptions that were present in the original Cantor programming model. Most of these assumptions were determined to be well-founded and conducive to efficient fine-grain programming. A few were found to be unnecessary, while others either were not supported or were refuted by the analysis of programs.

4.1.1 The Programming Model

The Cantor model defines objects as message-driven computing agents. A Cantor object cannot modify the global state of a computation without first receiving a message. While this requirement was motivated by runtime and program analysis issues, programming with "reactive" objects was usually intuitive to the programmer. The convenience and generality of the reactive programming approach was a positive result of the programming experiments.

We were also encouraged by the degree of concurrency we immediately achieved using Cantor. Since the Cantor programming model follows closely the object model of computation [1], Cantor computations are inherently concurrent. Specifically, objects do not share variables, and they communicate only via message passing, thus allowing them to be executed independently, *ie*, concurrently. Since message-passing operations must be

explicitly expressed in Cantor, the programmer is more likely to recognize unnecessary dependencies. By minimizing the message and object dependencies, and maximizing the number of useful computing objects, the programmer can exploit the amount of concurrency available in his application. Consequently, this programming model has been very useful in expressing fine-grain concurrency.

As discussed in Section 2.2, the original Cantor programming model supported no mechanism for discretionary receipt of messages. Originally, it was believed that the unbounded queue problem introduced by the use of message discretion could be avoided at the level of language implementation. Any unbounded queues then occurred at the program level and thus were the programmer's responsibility. Unfortunately, the unbounded queue problem surfaced during the implementation of object creation. When a future has been assigned to represent a new object, but the object has not yet been instantiated, messages sent to the new object must be queued. Discretion between ordinary messages and the initiating message must be exercised. Given this exception, message discretion in the form of functional abstraction as described in Section 2.2 was a natural extension of the programming model.

One assumption in the programming model that has not been conclusively supported or refuted is the preservation of message order between pairs of communicating objects. In some example programs, particularly programs with systolic message protocols, message-order preservation is a useful programming tool for minimizing program complexity and maximizing concurrency. In many other programs, the lack of message-order preservation would not seriously affect the program design or performance. Experiments are planned to investigate this issue further, including more program writing and examination of related implementation details.

4.1.2 The Programming Language

Another focus of the analysis of the programming experiments was the evaluation of Cantor as a programming language. Since its beginning, the design of Cantor was motivated by the desire for efficiency in compilation and execution. As a result of this emphasis, Cantor 2.0 was a minimal notation in which abstractive power was routinely sacrificed for efficiency. This series of programming experiments was designed to identify aspects of the language that could be modified to provide more abstractive power without sacrificing efficiency.

After analyzing the programs, we made conservative additions and deletions to the Cantor language definition. The additions include one-dimensional vectors, new control mechanisms, function abstraction, and custom objects/functions. These modifications substantially enhance the programmer's abstractive power while not seriously affecting efficiency. Features removed from the language include dynamic typing of variables, which has been replaced by static typing and compiler type inferencing. Full description

of these Cantor modifications can be found in Chapter 2.

Since the application programs written in Cantor are quite varied in nature, analysis of these programs significantly influenced the development of all phases of the Cantor programming system. The Cantor compiler has evolved to be an optimizing compiler that translates Cantor into an intermediate format, while minimizing the state and complexity of objects. The Cantor code generator performs program-flow analysis on intermediate code, seeking to minimize the required amount of runtime support. Naturally, these two minimizations are critical for efficient execution of Cantor programs on fine-grain machines. The Cantor interpreters, and the preliminary runtime systems, have been developed as software prototypes of runtime systems for fine-grain machines. We continue to use these tools to experiment with schemes for object creation and placement, custom object interface, and computation debugging and profiling.

4.2 About Fine-Grain Programming

Perhaps the most important goal of this thesis was to examine the nature of fine-grain programming. Using Cantor as a experimental tool, we have written enough programs in a fine-grain style to draw some conclusions. In addition to the programs presented in this thesis, Cantor applications that have been written include: fast-Fourier transform, a Mandelbrot generator, the game of Life, R-C chain-circuit simulation, digital logic simulation, a Collatz sieve, and the enumeration of spanning trees. Although formulations for Cantor programs are myriad, we have detected three general paradigms for the development of fine-grain programs.

First, functional program specifications can be mapped directly into message-driven programs. Simply using Cantor functions to represent each function invocation would not be efficient. However, representing each function call as a message send-and-receive sequence is typically an efficient fine-grain solution. The transformation of the functional program is straightforward: The receive of each sequence and the code that follows the receive is factored into a new object, and the rendezvous with the reply message is directed to the new object. The splitting and factoring operations can be readily automated and are repeatedly applied until all calls are eliminated. The factorial program in [2, page 13] is an example of this type of transformation.

The second paradigm is mapping specifications into message-driven programs. This approach is particularly applicable for problems from combinatorics. The strategy is to split the solution into three phases: enumeration of the possible solution set, concurrent evaluation of the potential solutions, and accumulation of the correct solutions. The solution to the N-queens problem illustrates this technique [2, pages 39–40]. The problem is specified as:

1. A solution is a vector q of integers q[1 . . . N];

2. $i = 1 \ldots N$: q[i] $\in 1 \ldots N$;

3. $1 \leq i < j \leq N$: q[i] $\neq$ q[j] and $j - i \neq$ | q[i] - q[j] |.

From rule 2, the size of the initial set of possible solutions is N^N. From rule 3, the size of the initial set of possible solutions is reduced to $N!$. Thus, one simple solution is to generate all $N!$ permutations, and then apply the test in rule 3. A further refinement is to combine the generation of permutations with the test of rule 3, so that once a partial permutation is determined to be wrong, all permutations that have the partial permutation as a prefix are immediately discarded. The concurrency in programs of this type is usually achieved by a breadth- or depth-first search of some tree structure.

The third paradigm is the object program as a "logical apparatus." Simulation of physical systems fits this paradigm in which each Cantor object is a simulation object. Most of the programs in this thesis fall into this category. For instance, the checkmate analyzer constructs a mesh of objects to represent the squares of the chessboard. Also, the Chandy-Misra SPSP program uses objects to represent the graph being analyzed.

With the exception of simulation, this third paradigm is the least characterized, yet it is also the most interesting. The amount of concurrency available in the program is very dependent on the object apparatus that is constructed and manipulated. While object creation is usually tightly-controlled, the message-passing patterns in programs of this type are typically non-trivial and not homogenous throughout the object structure. In spite of this complication, we have generally been encouraged by the success of programming experiments that were developed using this paradigm. Apparently, being able to reason about the apparatus as a whole enables Cantor programmers to formulate elegant solutions to some complicated problems.

4.3 Future Work

This iteration of program-writing has proved very fruitful in the evaluation and improvement of Cantor and in the development of understanding about the nature of fine-grain programming. Consequently, we are planning to continue our programming experiments, including some large application programs. In these programs, we intend to concentrate on evaluating the specifications of Cantor functions and custom objects/functions.

More work needs to be done concerning the low-level implementation of Cantor. Various schemes to solve Cantor-implementation problems, like fast object creation and efficient vector representation, will be investigated using the Mosaic runtime system. More work is also needed to develop a better programming environment for Cantor, including facilities for program profiling, simulation, and debugging.

Our experiments have shown that Cantor is a convenient, powerful notation for expressing concurrency in computation. We have identified and implemented a set of primitive constructs that are sufficient for the construction of efficient programs. Future

work will likely include development of a set of higher-level constructs that are compiled to Cantor objects or functions. This particular avenue of research is pursued with the goal of developing an efficient high-level specification of concurrent behavior.

Appendix A

Program Listings

A.1 Chandy-Misra SPSP

```
join(parent : ref) ::
[  ()
   [  ()
      send () to parent
   ]
]

nil_edge() ::
*[ (tag : sym, length : int, parent : ref)
   send () to parent
]

edge(head : ref, weight : int, tail : ref) ::
*[ (tag : sym, length : int, parent : ref)
   let j = join(parent)
   send (tag, length+weight, j) to head
   send (tag, length, j) to tail
]

vertex(label : sym, console : ref) ::
[  let listhead = nil_edge()
   [  case (cmd : sym) of
      "add edge" : (vj : ref, weight : int, sender : ref)
                   listhead = edge(vj, weight, listhead)
                   send () to sender
                   repeat
      "length"   : (length : int, pred : ref)
                   send ("length", length, pred) to listhead
```

```
                *[ (tag : sym , new_length : int , new_pred : ref)
                   if new_length < length then
                      length = new_length
                      pred = new_pred
                      send ("length", length, pred) to listhead
                   else
                      send () to new_pred
                   fi
                ]
   ]
]

[  (console : ref)
   let a = vertex("A", console)
   let b = vertex("B", console)
   let c = vertex("C", console)
   let d = vertex("D", console)
   let e = vertex("E", console)
   let f = vertex("F", console)
   let num_edges = 9
   let vi = vector 1..9 of ref
   let vj = vector 1..9 of ref
   let weight = vector 1..9 of int
   vi[1] = a; vj[1] = b; weight[1] = 2
   vi[2] = a; vj[2] = c; weight[2] = 1
   vi[3] = b; vj[3] = d; weight[3] = 1
   vi[4] = b; vj[4] = e; weight[4] = 2
   vi[5] = c; vj[5] = d; weight[5] = 3
   vi[6] = c; vj[6] = e; weight[6] = 2
   vi[7] = d; vj[7] = f; weight[7] = 3
   vi[8] = e; vj[8] = f; weight[8] = 2
   vi[9] = f; vj[9] = a; weight[9] = 1
   let i = 1
   let listhead = self
   [  if (i ≤ num_edges) then
         listhead = join(listhead)
         send ("add edge", vj[i], weight[i], listhead) to vi[i]
         i = i+1
         repeat
      fi
   ]
   send () to listhead
   [  ()
      send ("length", 0, self) to a
   ]
```

```
    [  ()
    ]
]
```

A.2 Warshall APSP

```
let NORTH= 1
let EAST  = 2
let SOUTH= 3
let WEST  = 4
let UP    = 1
let DOWN = 2
let N     = 4
let INF   = 256
let EVEN  = 2
let ODD   = 1


put(value, direction : int, tail: ref) =
[  let newtail = qelement(value, direction, nil)
   send ("add", newtail, self) to tail
   [  ()
   ]
   return (newtail)
]

Value(head : ref) =
[  send ("value", self) to head
   [  (value : int)
      return (value)
   ]
]

Dir(head : ref) =
[  send ("dir", self) to head
   [  (direction : int)
      return (direction)
   ]
]

get(head : ref) =
[  send ("next", self) to head
   [  (newhead : ref)
      return (newhead)
```

```
   ]
]

qelement(value, direction : int, next : ref) ::
*[  case (tag : sym) of
    "value" : (requester : ref)
              send (value) to requester
    "add"   : (newnext, requester : ref)
              next = newnext
              send () to requester
    "dir"   : (requester : ref)
              send (direction) to requester
    "next"  : (requester : ref)
              send (next) to requester
]

sync(parent : ref) ::
*[ (tag : sym)
   [  (tag : sym)
      send (tag) to parent
   ]
]

cell(row, col, length : int, main : ref) ::
[  let dir = vector 1..4 of ref
   dir[NORTH] = nil; dir[EAST] = nil; dir[SOUTH] = nil; dir[WEST] = nil
   [  (direction : int, creator, replier : ref, val[col..N] : int)
      dir[direction] = creator
      if (col < N) then
         dir[EAST] = cell(row, col+1, val[col+1], main)
         send (WEST, self, replier, val[col+1..N]) to dir[EAST]
      else
         send ("sync") to replier
      fi
      let rcvmsg = vector 1..2 of bool
      rcvmsg[UP] = false; rcvmsg[DOWN] = false
      let msg_vect = rcvmsg
      msg_vect[UP] = (row <> N)
      msg_vect[DOWN] = (row <> 1)
      *[   [   case (tag : sym) of
               "down" : (n : ref, ne : ref)
                        rcvmsg[DOWN] = true
                        dir[NORTH] = n
                        send ("up", dir[EAST]) to ne
               "up"   : (s : ref)
```

```
                    rcvmsg[UP] = true
                    dir[SOUTH] = s
                    send ("down", self, dir[EAST]) to dir[SOUTH]
      ]
      if (rcvmsg = msg_vect) then
         exit
      fi
   ]
   if (col = N) then
      send ("sync") to replier
   fi
]
let phase = 1
let initdir = 0
let rcvd = vector ODD..EVEN of bool
rcvd[ODD] = false; rcvd[EVEN] = false
let Direction = vector ODD..EVEN of int
let val = vector ODD..EVEN of int
let tail = vector ODD..EVEN of ref
tail[ODD] = qelement(0, 0, nil); tail[EVEN] = qelement(0, 0, nil)
let head = tail
*[   case (tag : sym) of
     "start" : ()
               send ("init", NORTH) to dir[NORTH]
               send ("init", EAST) to dir[EAST]
               send ("init", SOUTH) to dir[SOUTH]
               send ("init", WEST) to dir[WEST]
     "init"  : (direction : int)
               if (phase < row < col) or (phase < col < row) then
                  initdir = direction
               else
                  send ("init", direction) to dir[direction]
                  if ((direction mod 2) = 0) then
                     send ("val", NORTH, length) to dir[NORTH]
                     send ("val", SOUTH, length) to dir[SOUTH]
                  else
                     send ("val", EAST, length) to dir[EAST]
                     send ("val", WEST, length) to dir[WEST]
                  fi
               fi
     "val"   : (direction : int, value : int)
               let index1 = EVEN
               let index2 = ODD
               if ((direction mod 2) <> 0) then
                  index1 = ODD
```

```
            index2 = EVEN
        fi
        if (rcvd[index1]) then
            tail[index1] = put(value, direction, tail[index1])
        else
            rcvd[index1] = true
            val[index1] = value
            Direction[index1] = direction
            if (rcvd[index2]) then
                if ((val[1]+val[2]) < length) then
                    length = val[1]+val[2]
                fi
                send ("val", Direction[1], val[1]) to dir[Direction[1]]
                send ("val", Direction[2], val[2]) to dir[Direction[2]]
                phase = phase+1
                if (row = col = phase) then
                    send ("start") to self
                else
                    if ((initdir = NORTH) and (phase = row)) or ((initdir = SOUTH) and (phase = col)) then
                        send ("init", initdir) to self
                    fi
                fi
                Direction[index2] = Dir(head[index2])
                if (Direction[index2] <> 0) then
                    val[index2] = Value(head[index2])
                    head[index2] = get(head[index2])
                    rcvd[index2] = true
                else
                    rcvd[index2] = false
                fi
                rcvd[index1] = false
            fi
        fi
    ]
]

[   (console : ref)
    let rows = vector 1..N of ref
    let w = vector 1..N*N of int
    w[1] = 0; w[2] = 1; w[3] = INF; w[4] = 100
    w[5] = 30; w[6] = 0; w[7] = 2; w[8] = 7
    w[9] = 8; w[10] = 50; w[11] = 0; w[12] = 3
    w[13] = 4; w[14] = INF; w[15] = 40; w[16] = 0
    let i = 1
    let synclist = self
```

```
    let requester = self
    [   if (i ≤ N) then
            synclist = sync(synclist)
            rows[i] = cell(i, 1, w[(i−1)*N+1], self)
            send (WEST, requester, synclist, w[(i−1)*N+1..i*N]) to rows[i]
            requester = nil
            i = i+1; repeat
        fi
    ]
    send ("sync") to synclist
    [   (tag : sym)
    ]
    i = 1
    [   if (i < N) then
            send ("up", rows[i+1]) to rows[i]
            i = i+1; repeat
        fi
    ]
    send ("sync") to synclist
    [   (tag : sym)
    ]
    send ("start") to rows[1]
    let phase = 1
    *[   case (tag : sym) of
         "val" : (direction : int, value : int)

                 phase = phase+1
                 if (phase = N) then
                     exit
                 fi
    ]
]
```

A.3 Quickhull

```
let NUMPOINTS= 16
let EPSILON    = 0.10


cross(x, y, ax, ay, bx, by : real)
[   let dx = bx−ax
    let dy = by−ay
    return (dy*(x−ax)−dx*(y−ay))
]
```

```
outside(x, y, ax, ay, bx, by : real)
[  return (cross(x, y, ax, ay, bx, by) < 0.00)
]

angle(x, y, ax, ay, cross : real)
[  let hypo = (x−ax)*(x−ax)+(y−ay)*(y−ay)
   if (hypo = 0.00) then
      return (0.00)
   else
      return (cross/hypo)
   fi
]

indexmin(points[1..NUMPOINTS] : real)
[  let i = 2
   let index = 1
   [  if (i ≤ NUMPOINTS) then
         if (points[i] < points[index]) then
            index = i
         fi
         i = i+1; repeat
      fi
   ]
   return (index)
]

nilpoint() ::
[  (h : int, leftx, lefty, rightx, righty, hx, hy, maxcross, maxangle : real, requester : ref)
   if (h = 0) then
      send ("edge") to requester
   else
      send ("h", h, hx, hy) to requester
   fi
   [  (tag : sym, hx, hy, leftx, lefty, rightx, righty : real, leftlist, rightlist : ref)
      send (leftlist, rightlist) to requester
   ]
]

point(index : int, x, y : real, next : ref) ::
*[ (h : int, leftx, lefty, rightx, righty, hx, hy, maxcross, maxangle : real, requester : ref)
   let pointcross = abs cross(x, y, leftx, lefty, rightx, righty)
   let pointangle = angle(x, y, leftx, lefty, pointcross)
   if (pointcross > maxcross) or ((pointcross = maxcross) and (pointangle > maxangle)) then
      send (index, leftx, lefty, rightx, righty, x, y, pointcross, pointangle, requester) to next
   else
```

```
      send (h, leftx, lefty, rightx, righty, hx, hy, maxcross, maxangle, requester) to next
   fi
   let new_next = next
   [  (tag : sym , hx, hy, leftx, lefty, rightx, righty : real , leftlist, rightlist : ref)
      if (outside(x, y, leftx, lefty, hx, hy)) then
         new_next = leftlist
         leftlist = self
      else
         if (outside(x, y, hx, hy, rightx, righty)) then
            new_next = rightlist
            rightlist = self
         fi
      fi
      send ("h", hx, hy, leftx, lefty, rightx, righty, leftlist, rightlist) to next
      next = new_next
   ]
]

collector(tag : sym , parent : ref) ::
[  (tag1 : sym , num1, vi1[1..num1] : int)
   [  (tag2 : sym , num2, vi2[1..num2] : int)
      let vertices = vector 1..NUMPOINTS of int
      if (tag1 = "first") then
         vertices[1..num1] = vi1
         vertices[num1+1..num1+num2] = vi2
      else
         vertices[1..num2] = vi2
         vertices[num2+1..num1+num2] = vi1
      fi
      send (tag, num1+num2, vertices[1..num1+num2]) to parent
   ]
]

quickhull(tag : sym , left, right : int , leftx, lefty, rightx, righty : real , listhead, replier : ref) ::
[  send (0, leftx, lefty, rightx, righty, 0.00, 0.00, 0.00, 0.00, self) to listhead
   [  case (tag : sym ) of
      "edge" : ()
               let vertices = vector 1..1 of int
               vertices[1] = left
               send (tag, 1, vertices) to replier
      "h"    : (h : int , hx, hy : real )
               let newreplier = collector(tag, replier)
               let leftlist = nilpoint()
               let rightlist = nilpoint()
               send ("h", hx, hy, leftx, lefty, rightx, righty, leftlist, rightlist) to listhead
```

```
            [  (leftlist, rightlist : ref)
               call quickhull("first", left, h, leftx, lefty, hx, hy,
                              leftlist, newreplier)
               call quickhull("last", h, right, hx, hy, rightx,
                              righty, rightlist, newreplier)
            ]
   ]
]

[  (console : ref)
   let pointsx = vector 1..NUMPOINTS of real
   let pointsy = vector 1..NUMPOINTS of real
   pointsx[1] = −1.00 ; pointsy[1] = 0.50
   pointsx[2] = −2.50 ; pointsy[2] = 0.50
   pointsx[3] = −0.50 ; pointsy[3] = 1.50
   pointsx[4] = −0.50 ; pointsy[4] = 2.50
   pointsx[5] = −1.50 ; pointsy[5] = 2.00
   pointsx[6] = 1.00 ; pointsy[6] = 1.00
   pointsx[7] = 1.00 ; pointsy[7] = 2.90
   pointsx[8] = 2.50 ; pointsy[8] = 2.90
   pointsx[9] = 5.00 ; pointsy[9] = 1.00
   pointsx[10] = 0.50 ; pointsy[10] = −0.50
   pointsx[11] = 0.50 ; pointsy[11] = −2.00
   pointsx[12] = 3.00 ; pointsy[12] = −1.00
   pointsx[13] = −0.50 ; pointsy[13] = −1.50
   pointsx[14] = −1.50 ; pointsy[14] = −1.50
   pointsx[15] = 4.00 ; pointsy[15] = 0.00
   pointsx[16] = −2.50 ; pointsy[16] = 0.50
   let list = nilpoint()
   let i = 1
   [  if (i ≤ NUMPOINTS) then
         list = point(i, pointsx[i], pointsy[i], list)
         i = i+1; repeat
      fi
   ]
   let left = indexmin(pointsx[1..NUMPOINTS])
   call quickhull("first", left, left, pointsx[left], pointsy[left],
                  pointsx[left], pointsy[left]−EPSILON, list, self)
   [  (tag : sym , num : int , vertices[1..num] : int)
      i = 1
      [  if (i ≤ num) then
            send (vertices[i]) to console
            i = i+1; repeat
         fi
      ]
```

```
      send (vertices[1]) to console
   ]
]
```

A.4 Paraffin Isomers

```
let N= 10


stackelement(requester, next : ref) ::
[  (number : int, list : ref)
   send ("reply", number, list) to requester
   send (number, list) to next
]

endlist() ::
*[   case (tag : sym) of
     "product" : (replier[1..4] : ref, head[1..4] : ref, depth, num : int, Refs[1..5] : ref,
                                          Index[1..4] : int, list, tail : ref, bicentroid : bool)
                 Refs[depth] = head[depth]
                 send ("reply", replier, head, depth−1, num, Refs, Index,
                                                        list, tail, bicentroid) to replier[depth]
]

tree(tuple[1..4] : int, pointers[1..4], next : ref, bicentroid, Tail : bool) ::
*[   case (tag : sym) of
     "product" : (replier[1..4] : ref, head[1..4] : ref, depth, num : int, Refs[1..5] : ref,
                                          Index[1..4] : int, list, tail : ref, Bicentroid : bool)
                 Refs[depth] = self
                 [  if (Refs[depth+1] = self) then
                       depth = depth+1
                       replier[depth] = self
                       repeat
                    fi
                 ]
                 if (Refs[depth+1] = nil) then
                    list = tree(Index, Refs[1..4], list, Bicentroid, num = 0)
                    if (num = 0) then
                       tail = list
                    fi
                    send (tag, replier, head, depth, num+1, Refs, Index,
                                                            list, tail, Bicentroid) to next
                 else
                    replier[depth+1] = self
```

```
                send (tag, replier, head, depth+1, num, Refs, Index,
                                                    list, tail, Bicentroid) to Refs[depth+1]
            fi
    "reply"  : (replier[1..4] : ref, head[1..4] : ref, depth, num : int, Refs[1..5] : ref,
                                    Index[1..4] : int, list, tail : ref, Bicentroid : bool)
            if not Tail then
               let i = depth+1
               [  if (i ≤ 4) then
                     if (Index[depth] = Index[i]) then
                        Refs[i] = next
                     fi
                     i = i+1; repeat
                  fi
               ]
            fi
            send ("product", replier, head, depth, num, Refs, Index,
                                                    list, tail, Bicentroid) to next
    "next"   : (Next, requester : ref)
            next = Next
            Tail = false
            send () to requester
    "print"  : (console : ref)
            if bicentroid then
               send (self, N, tuple, "left", pointers[1], "right", pointers[2]) to console
            else
               send (self, tuple[1]+tuple[2]+tuple[3]+tuple[4]+1, tuple, pointers) to console
            fi
            send (tag, console) to next
            send (tag, console) to pointers[1]
            send (tag, console) to pointers[2]
            send (tag, console) to pointers[3]
            send (tag, console) to pointers[4]
            *[ (tag : sym, console : ref)
            ]
]

child(subtree : int, Ref, next : ref) ::
[  send ("request", self) to Ref
   let replyrcvd = false
   let solsrcvd = false
   let Refs = vector 1..5 of ref
   let list = nil
   let index = 0
   [  [  case (tag : sym) of
         "solutions" : (Index : int, refs[1..5] : ref)
```

```
                    replyrcvd = true
                    index = Index
                    Refs = refs
          "reply"   : (Number : int, List : ref)
                    solsrcvd = true
                    list = List
      ]
      if not (replyrcvd and solsrcvd) then
         repeat
      fi
   ]
   Refs[index] = list
   send ("solutions", index+1, Refs) to next
]

product(requester : ref, Index[1..4] : int, bicentroid : bool) ::
[  (tag : sym, index : int, Refs[1..5] : ref)
   let replier = vector 1..4 of ref
   let head = Refs[1..4]
   if (Refs[1] = nil) then
      replier[1] = nil
      let list = endlist()
      let tail = tree(Index, Refs[1..4], list, bicentroid, true)
      send ("reply", replier, head, 1, 1, Refs, Index, tail, tail, bicentroid) to requester
   else
      replier[1] = requester
      send ("product", replier, head, 1, 0, Refs, Index, endlist(), nil, bicentroid) to Refs[1]
   fi
]

tuple(i, j, k, l : int, iref, jref, kref, lref : ref, next : ref, bicentroid : bool) ::
[  let Index = vector 1..4 of int
   let Refs = vector 1..5 of ref
   Index[1] = i; Index[2] = j; Index[3] = k; Index[4] = l
   Refs[1] = iref; Refs[2] = jref; Refs[3] = kref; Refs[4] = lref; Refs[5] = nil
   let listhead = product(self, Index, bicentroid)
   let I = 4
   [  if (I > 0) then
         if (Index[I] <> 0) then
            listhead = child(Index[I], Refs[I], listhead)
         fi
         I = I-1; repeat
      fi
   ]
   send ("solutions", 1, Refs) to listhead
```

```
    let replyrcvd = false
    let solsrcvd = false
    let number = 0
    let totalnum = 0
    let list = nil
    let sublist = nil
    let tail = nil
    [   [   case (tag : sym) of
            "solutions" : (Number : int, List : ref)
                          solsrcvd = true
                          totalnum − Number
                          list = List
            "reply"     : (replier[1..4] : ref, head[1..4] : ref, Depth, Number : int,
                                                    Dummy1[1..5] : ref, Dummy2[1..4] : int,
                                                          List : ref, Tail : ref, Bool : bool)
                          replyrcvd = true
                          number = Number
                          sublist = List
                          tail = Tail
        ]
        if not (replyrcvd and solsrcvd) then
            repeat
        fi
    ]
    if (list <> nil) then
        send ("next", list, self) to tail
        [   ()
        ]
    fi
    send ("solutions", totalnum+number, sublist) to next
]

rooted() ::
[   (index : int, subtrees[0..(N+1)/2] : ref, main : ref)
    subtrees[index] = self
    if (index < (N)/2) then
        send (index+1, subtrees[0..(N+1)/2], main) to rooted()
    else
        send (subtrees[0..(N+1)/2], main) to free()
    fi
    let listhead = self
    let i = index−1
    [   if (i ≥ index/3) then
            let j = i
            [   if (j ≥ 0) then
```

```
                let k = j
                [   if (k ≥ 0) then
                        if (i+j+k = index−1) then
                            listhead = tuple(i, j, k, 0, subtrees[i], subtrees[j], subtrees[k],
                                                                                nil, listhead, false)
                        fi
                        k = k−1; repeat
                    fi
                ]
                j = j−1; repeat
            fi
        ]
        i = i−1; repeat
    fi
]
send ("solutions", 0, nil) to listhead
let finished = false
let stack = nil
let list = nil
let number = 0
*[  case (tag : sym) of
    "solutions" : (Number : int, List : ref)
                  number = Number
                  list = List
                  finished = true
                  send (number, list) to stack
    "request"   : (requester : ref)
                  if not finished then
                      stack = stackelement(requester, stack)
                  else
                      send ("reply", number, list) to requester
                  fi
]
]

free() ::
[ (subtrees[0..(N+1)/2], main : ref)
  let listhead = main
  if (N mod 2 = 0) then
      listhead = tuple(N/2, N/2, 0, 0, subtrees[N/2], subtrees[N/2], nil, nil,
                                                                        listhead, true)
  fi
  let i = N/2
  [ if (i ≥ N/4) then
        let j = N−i
```

```
        [  if (j ≥ 0) then
             let k = N−i−j
             [  if (k ≥ 0) then
                  let l = N−i−j−k
                  [  if (l ≥ 0) then
                       if (i ≤ j+k+l) and (i+j+k+l = N−1) and (i ≥ j ≥ k ≥ l) then
                         listhead = tuple(i, j, k, l, subtrees[i], subtrees[j],
                                                  subtrees[k], subtrees[l], listhead, false)
                       fi
                       l = l−1; repeat
                     fi
                  ]
                  k = k−1; repeat
                fi
             ]
             j = j−1; repeat
           fi
        ]
        i = i−1; repeat
      fi
   ]
   send ("solutions", 0, nil) to listhead
]

[  (console : ref)
   let nilvect = vector 0..(N+1)/2 of ref
   nilvect[0] = nil
   if (N > 1) then
      send (1, nilvect, self) to rooted()
   else
      send (nilvect, self) to free()
   fi
   [  (tag : sym, number : int, list : ref)
      send ("There are", number, "paraffin isomers of", N, "nodes.") to console
   ]
]
```

A.5 Checkmate Analyzer

```
let NUMPIECES= 16
let NUMDIR   = 8
let NUMROWS  = 8
let WHITE    = 1
let BLACK    = −1
```

```
let UP          = 1
let DOWN        = 2
let DIAG        = 3
let N           = 1
let NE          = 2
let E           = 3
let SE          = 4
let S           = 5
let SW          = 6
let W           = 7
let NW          = 8
let EMPTY       = 0
let QKNIGHT     = 2
let QROOK       = 1
let QBISHOP     = 3
let KKNIGHT     = 7
let KROOK       = 8
let KBISHOP     = 6
let KING        = 5
let QUEEN       = 4
let PAWN1       = 9
let PAWN2       = 10
let PAWN3       = 11
let PAWN4       = 12
let PAWN5       = 13
let PAWN6       = 14
let PAWN7       = 15
let PAWN8       = 16


nilsquare() ::
[  let empty = vector 0..1 of int
   let nilvect = vector 0..1 of ref
   empty[1] = 0
   nilvect[1] = nil
   *[   case (tag : sym) of
        "check?" : (requester : ref, whitesturn : bool)
                   send ("reply", EMPTY, 1, empty, nilvect, empty) to requester
        "hit?"   : (requester, sender : ref, chesspiece : int, direction : int, plus : bool)
                   send ("reply", false, nil) to requester
   ]
]

square(row, col, piece : int) ::
[  (dir[N..NW] : ref)
```

```
    *[   case (tag : sym) of
         "check?" : (requester : ref, whitesturn : bool)
                    let checker = checksquare(requester, dir, piece, whitesturn, false)
         "hit?"   : (requester, sender : ref, chesspiece : int, direction : int, plus : bool)
                    if (( abs chesspiece = QKNIGHT) or ( abs chesspiece = KKNIGHT))
                                                        and (sender = requester) then
                       if plus then
                          send ("hit?", requester, self, chesspiece, (direction mod
                                       NUMDIR)+1, false) to dir[direction mod NUMDIR+1]
                       else
                          send ("hit?", requester, self, chesspiece, (direction mod
                                                NUMDIR)+1, false) to dir[direction-1]
                       fi
                    else
                       if (piece = EMPTY) and (( abs chesspiece = QUEEN) or
                              ( abs chesspiece = QROOK) or ( abs chesspiece = KROOK) or
                         ( abs chesspiece = QBISHOP) or ( abs chesspiece = KBISHOP)) then
                          send ("hit?", requester, self, chesspiece, direction, false)
                                                                    to dir[direction]
                       else
                          send ("reply", piece = chesspiece, self) to requester
                       fi
                    fi
         "block?" : (requester : ref, attacker : ref, direction : int, whitesturn : bool)
                    if (attacker = self) then
                       let empty = vector 0..0 of int
                       send ("block", 0, empty, empty, empty) to requester
                    else
                       let collector = collectblocks(requester, row, col)
                       let checker = checksquare(collector, dir, piece,
                             not whitesturn, true)
                       send ("block?", collector, attacker, direction, whitesturn) to dir[direction]
                    fi
    ]
]

checksquare(requester : ref, neighbors[N..NW] : ref, piece : int, whitesturn : bool, block : bool) ::
[   let num = WHITE
    if whitesturn then
       num = BLACK
    fi
    let King = checkpiece(requester, piece, neighbors, KING*num, N, NW, 1, false)
    let Queen = checkpiece(King, piece, neighbors, QUEEN*num, N, NW, 1, false)
    let KKnight = checkpiece(Queen, piece, neighbors, KKNIGHT*num, NE, NW, 2, true)
    let QKnight = checkpiece(KKnight, piece, neighbors, QKNIGHT*num, NE, NW, 2, true)
```

```
    let KRook = checkpiece(QKnight, piece, neighbors, KROOK*num, N, NW, 2, false)
    let QRook = checkpiece(KRook, piece, neighbors, QROOK*num, N, NW, 2, false)
    let KBishop = checkpiece(QRook, piece, neighbors, KBISHOP*num, NE, NW, 2, false)
    let QBishop = checkpiece(KBishop, piece, neighbors, QBISHOP*num, NE, NW, 2, false)
    let start = NE; letend = NW; letincr = 6
    if whitesturn then
       start = SE; end = SW; incr = 2
       if block then
          start = N; end = N; incr = 1
       fi
    else
       if block then
          start = S; end = S; incr = 1
       fi
    fi
    let Pawn1 = checkpiece(QBishop, piece, neighbors, PAWN1*num, start, end, incr, false)
    let Pawn2 = checkpiece(Pawn1, piece, neighbors, PAWN2*num, start, end, incr, false)
    let Pawn3 = checkpiece(Pawn2, piece, neighbors, PAWN3*num, start, end, incr, false)
    let Pawn4 = checkpiece(Pawn3, piece, neighbors, PAWN4*num, start, end, incr, false)
    let Pawn5 = checkpiece(Pawn4, piece, neighbors, PAWN5*num, start, end, incr, false)
    let Pawn6 = checkpiece(Pawn5, piece, neighbors, PAWN6*num, start, end, incr, false)
    let Pawn7 = checkpiece(Pawn6, piece, neighbors, PAWN7*num, start, end, incr, false)
    let Pawn8 = checkpiece(Pawn7, piece, neighbors, PAWN8*num, start, end, incr, false)
    let empty = vector 0..0 of int
    let nilvect = vector 0..0 of ref
    send ("reply", piece, 0, empty, nilvect, empty) to Pawn8
]

checkpiece(requester : ref, piece : int, neighbors[N..NW] : ref, chesspiece : int,
                                                start, end, increment : int, knight : bool) ::
[   let i = start
    let current = self
    [   if (i ≤ end) then
            current = missile(current, neighbors[i], i, chesspiece, true)
            if knight then
               current = missile(current, neighbors[i], i, chesspiece, false)
            fi
            i = i+increment
            repeat
        fi
    ]
    let empty = vector 0..0 of int
    let nilvect = vector 0..0 of ref
    send ("hit", false, nil, 0) to current
    let hit = false
```

```
    let hitref = nil
    let hitdir = 0
    let hitrcvd = false
    let numtrouble = 0
    let troublepiece = vector 0..NUMPIECES of int
    let troubleref = vector 0..NUMPIECES of ref
    let troubledir = vector 0..NUMPIECES of int
    let replyrcvd = false
    *[  [  case (tag : sym) of
           "reply" : (piece : int, Numtrouble : int, Troublepiece[0..Numtrouble] : int,
                                                     Troubleref[0..Numtrouble] : ref,
                                                     Troubledir[0..Numtrouble] : int)
                     replyrcvd = true
                     numtrouble = Numtrouble
                     troublepiece[0..numtrouble] = Troublepiece[0..Numtrouble]
                     troubleref[0..numtrouble] = Troubleref[0..Numtrouble]
                     troubledir[0..numtrouble] = Troubledir[0..Numtrouble]
           "hit"   : (Hit : bool, Hitref : ref, Hitdir : int)
                     hitrcvd = true
                     hit = Hit; hitref = Hitref; hitdir = Hitdir
       ]
       if (replyrcvd and hitrcvd) then
          exit
       fi
    ]
    if (hit) then
       numtrouble = numtrouble+1
       troublepiece[numtrouble] = chesspiece
       troubleref[numtrouble] = hitref
       troubledir[numtrouble] = hitdir
    fi
    send ("reply", piece, numtrouble, troublepiece[0..numtrouble], troubleref[0..numtrouble],
                                                     troubledir[0..numtrouble]) to requester
]

missile(requester : ref, neighbor : ref, direction : int, piece : int, knight : bool) ::
[  send ("hit?", self, self, piece, direction, knight) to neighbor
   let trouble = false
   let troubleref = nil
   let hit = false
   let hitref = nil
   let hitdir = 0
   let hitrcvd = false
   let replyrcvd = false
   *[  [  case (tag : sym) of
```

```
            "reply" : (Trouble : bool, Troubleref : ref)
                      replyrcvd = true
                      trouble = Trouble; troubleref = Troubleref
            "hit"   : (Hit : bool, Hitref : ref, Hitdir : int)
                      hitrcvd = true
                      hit = Hit; hitref = Hitref; hitdir = Hitdir
        ]
        if (replyrcvd and hitrcvd) then
            exit
        fi
    ]
    let Ref = nil
    let Dir = 0
    if (trouble) then
        Ref = troubleref; Dir = direction
    else
        if (hit) then
            Ref = hitref; Dir = hitdir
        fi
    fi
    send ("hit", trouble or hit, Ref, Dir) to requester
]

connectsquare(row, col : int, requester : ref, Row[-NUMPIECES..NUMPIECES],
                          Col[-NUMPIECES..NUMPIECES] : int, whitesturn : bool) ::
[   let dir = vector N..NW of ref
    dir[N] = nil; dir[NE] = nil; dir[E] = nil; dir[SE] = nil
    dir[S] = nil; dir[SW] = nil; dir[W] = nil; dir[NW] = nil
    let i = -NUMPIECES
    let piece = 0
    [   if (i ≤ NUMPIECES) then
            if (Row[i] = row) and (Col[i] = col) then
                piece = i
            fi
            i = i+1; repeat
        fi
    ]
    let child = square(row, col, piece)
    [   (direction : int, creator : ref, element : ref)
        dir[direction] = element
        send ("ref", child) to creator
        if (col < NUMROWS) then
            let eastconnect = connectsquare(row, col+1, requester, Row, Col, whitesturn)
            send (W, self, child) to eastconnect
            [   (tag : sym, tmp : ref)
```

```
        dir[E] = tmp
    ]
fi
if (col = NUMROWS) then
    send ("ack") to requester
fi
let rcvmsg = vector UP..DIAG of bool
rcvmsg[UP] = false; rcvmsg[DOWN] = false; rcvmsg[DIAG] = false
let msgvect = rcvmsg
msgvect[UP] = (row < NUMROWS)
msgvect[DOWN] = (row > 1)
msgvect[DIAG] = (row < NUMROWS) and (col < NUMROWS)
*[  [  case (tag : sym) of
        "down" : (ne : ref, nw : ref, n : ref, netemp : ref, nwtemp : ref)
                 rcvmsg[DOWN] = true
                 dir[N] = n; dir[NE] = ne; dir[NW] = nw
                 send ("up", child, dir[E], eastconnect) to netemp
                 send ("diag", child) to nwtemp
        "up"   : (sw : ref, s : ref, sconnect : ref)
                 rcvmsg[UP] = true
                 dir[SW] = sw; dir[S] = s
                 send ("down", dir[E], dir[W], child, eastconnect, creator) to sconnect
        "diag" : (se : ref)
                 rcvmsg[DIAG] = true
                 dir[SE] = se
   ]
   if (rcvmsg = msgvect) then
      i = 1
      [  if (i ≤ NUMDIR) then
            if (dir[i] = nil) or (dir[i] = requester) then
               dir[i] = nilsquare()
            fi
            i = i+1; repeat
         fi
      ]
      send (dir) to child
      if ( abs piece = KING) and (whitesturn = (piece > 0)) then
         send ("king", child, dir) to requester
      fi
      exit
   fi
]
if (col = NUMROWS-1) then
   send ("ack") to requester
fi
```

```
    ]
]

buildboard(requester : ref, row[-NUMPIECES..NUMPIECES],
                              col[-NUMPIECES..NUMPIECES] : int, whitesturn : bool) ::
[   let rows = vector 1..NUMROWS of ref
    let children = rows
    let i = 1
    let j = 0
    let kingrcvd = false
    let king = nil
    let neighbors = vector N..NW of ref
    [   if (i ≤ NUMROWS) then
            rows[i] = connectsquare(i, 1, self, row, col, whitesturn)
            send (W, self, self) to rows[i]
            *[    case (tag : sym) of
                    "ref" : (tmp : ref)
                            children[i] = tmp
                            exit
                    "ack" : ()
                            j = j+1
            ]
            i = i+1; repeat
        fi
    ]
    [   if (j < NUMROWS) then
            [   (tag : sym)
                j = j+1
            ]
            repeat
        fi
    ]
    i = 1
    [   if (i < NUMROWS) then
            send ("up", nil, children[i+1], rows[i+1]) to rows[i]
            i = i+1; repeat
        fi
    ]
    i = 0
    [   [   case (tag : sym) of
              "king" : (King, Neighbors[N..NW] : ref)
                       king = King
                       neighbors = Neighbors
                       kingrcvd = true
              "ack"  : ()
```

```
                i = i+1
      ]
      if (i < NUMROWS) or not kingrcvd then
         repeat
      fi
   ]
   send (king, neighbors) to requester
]

checkconfig(requester : ref, row[-NUMPIECES..NUMPIECES],
                              col[-NUMPIECES..NUMPIECES] : int, whitesturn : bool) ::
[  let board = buildboard(self, row, col, whitesturn)
   [  (king : ref, neighbors[N..NW] : ref)
      send ("check?", self, whitesturn) to king
      [  (tag : sym, piece : int, numtrouble : int, troublepiece[0..numtrouble] : int,
                              troubleref[0..numtrouble] : ref, troubledir[0..numtrouble] : int)
         send ("reply", numtrouble = 0) to requester
      ]
   ]
]

collectconfig(requester : ref, row[-NUMPIECES..NUMPIECES], col[-NUMPIECES..
                           NUMPIECES] : int, piece, Row, Col : int, whitesturn : bool) ::
[  let checker = checkconfig(self, row, col, whitesturn)
   let replyrcvd = false
   let blockrcvd = false
   let block = false
   let numblockers = 0
   let blockers = vector 0..NUMPIECES of int
   let destrow = blockers
   let destcol = blockers
   let command = ""
   *[  [  case (tag : sym) of
         "capture" : (Numblockers : int, Blockers[0..Numblockers] : int,
                                 Destrow[0..Numblockers], Destcol[0..Numblockers] : int)
                     blockrcvd = true
                     command = tag
                     numblockers = Numblockers
                     blockers[0..numblockers] = Blockers[0..Numblockers]
                     destrow[0..numblockers] = Destrow[0..Numblockers]
                     destcol[0..numblockers] = Destcol[0..Numblockers]
         "reply"   : (Block : bool)
                     replyrcvd = true
                     block = Block
         "block"   : (Numblockers : int, Blockers[0..Numblockers] : int,
```

```
                                    Destrow[0..Numblockers], Destcol[0..Numblockers] : int)
                        blockrcvd = true
                        command = tag
                        numblockers = Numblockers
                        blockers[0..numblockers] = Blockers[0..Numblockers]
                        destrow[0..numblockers] = Destrow[0..Numblockers]
                        destcol[0..numblockers] = Destcol[0..Numblockers]
        ]
        if (blockrcvd and replyrcvd) then
            exit
        fi
    ]
    if (block) then
        numblockers = numblockers+1
        blockers[numblockers] = piece
        destrow[numblockers] = Row
        destcol[numblockers] = Col
    fi
    send (command, numblockers, blockers[0..numblockers], destrow[0..numblockers],
                                                    destcol[0..numblockers]) to requester
]

collectblocks(requester : ref, row, col : int) ::
[   let blockrcvd = false
    let replyrcvd = false
    let numtrouble = 0
    let troublepiece = vector 0..NUMPIECES of int
    let troubleref = vector 0..NUMPIECES of ref
    let troubledir = vector 0..NUMPIECES of int
    let numblockers = 0
    let blockpiece = vector 0..NUMPIECES of int
    let destrow = vector 0..NUMPIECES of int
    let destcol = vector 0..NUMPIECES of int
    [   if not (replyrcvd and blockrcvd) then
            [   case (tag : sym) of
                "reply" : (Piece : int, Numtrouble : int, Troublepiece[0..Numtrouble] : int,
                                                    Troubleref[0..Numtrouble] : ref,
                                                    Troubledir[0..Numtrouble] : int)
                        replyrcvd = true
                        numtrouble = Numtrouble
                        troublepiece[0..numtrouble] = Troublepiece[0..Numtrouble]
                        troubleref[0..numtrouble] = Troubleref[0..Numtrouble]
                        troubledir[0..numtrouble] = Troubledir[0..Numtrouble]
                "block" : (Numblockers : int, Blockpiece[0..Numblockers] : int,
                                    Destrow[0..Numblockers], Destcol[0..Numblockers] : int)
```

```
                    blockrcvd = true
                    numblockers = Numblockers
                    blockpiece[0..numblockers] = Blockpiece[0..Numblockers]
                    destrow[0..numblockers] = Destrow[0..Numblockers]
                    destcol[0..numblockers] = Destcol[0..Numblockers]
        ]
        repeat
    fi
  ]
  if (numtrouble > 0) then
    blockpiece[numblockers+1..numblockers+numtrouble] = troublepiece[1..numtrouble]
    let i = numblockers+1
    [  if (i ≤ (numblockers+numtrouble)) then
         destrow[i] = row
         destcol[i] = col
         i = i+1; repeat
       fi
    ]
    numblockers = numblockers+numtrouble
  fi
  send ("block", numblockers, blockpiece[0..numblockers], destrow[0..numblockers],
                                                    destcol[0..numblockers]) to requester
]

blockpiece(requester : ref, attackdir : ref, row[-NUMPIECES..NUMPIECES], col[-NUMPIECES..
        NUMPIECES] : int, piece : int, pieceref : ref, piecedir : int, whitesturn : bool) ::
[  send ("block?", self, pieceref, piecedir, whitesturn) to attackdir
  let i = 1
  let empty = vector 0..0 of int
  [  (tag : sym, numblockers, blockpiece[0..numblockers], destrow[0..numblockers],
                                                    destcol[0..numblockers] : int)
    if (numblockers > 0) then
      let current = requester
      let i = 1
      let Row = vector -NUMPIECES..NUMPIECES of int
      let Col = Row
      [  if (i ≤ numblockers) then
           let Row = row
           let Col = col
           Row[blockpiece[i]] = destrow[i]
           Col[blockpiece[i]] = destcol[i]
           current = collectconfig(current, Row, Col, blockpiece[i], destrow[i],
                                                    destcol[i] = whitesturn)
           i = i+1; repeat
           fi
```

```
        ]
            send ("block", 0, empty, empty, empty) to current
            else
              send ("block", 0, empty, empty, empty) to requester
            fi
    ]
]

capturepiece(requester : ref, row[−NUMPIECES..NUMPIECES], col[−NUMPIECES..
                NUMPIECES] : int, attacker : int, attackref : ref, whitesturn : bool) ::
[  send ("check?", self, not whitesturn) to attackref
   [  (tag : sym, piece : int, numtrouble : int, troublepiece[0..numtrouble] : int,
                          troubleref[0..numtrouble] : ref, troubledir[0..numtrouble] : int)
      let empty = vector 0..0 of int
      if (numtrouble > 0) then
        let current = requester
        let i = 1
        let Row = vector −NUMPIECES..NUMPIECES of int
        let Col = Row
        [  if (i ≤ numtrouble) then
             let Row = row
             let Col = col
             Row[troublepiece[i]] = row[attacker]
             Col[troublepiece[i]] = col[attacker]
             Row[attacker] = 0
             Col[attacker] = 0
             current = collectconfig(current, Row, Col, troublepiece[i], row[attacker],
                                                                col[attacker] = whitesturn)
             i = i+1; repeat
           fi
        ]
        send ("capture", 0, empty, empty, empty) to current
      else
        send ("capture", 0, empty, empty, empty) to requester
      fi
   ]
]

notmoveking(requester, king, neighbors[N..NW] : ref, whitesturn : bool,
            row[−NUMPIECES..NUMPIECES], col[−NUMPIECES..NUMPIECES] : int) ::
[  (tag : sym, piece : int, numtrouble : int, troublepiece[0..numtrouble] : int,
                          troubleref[0..numtrouble] : ref, troubledir[0..numtrouble] : int)
   let empty = vector 0..0 of int
   if (numtrouble = 0) then
     send ("notmoveking") to requester
```

```
    else
      if (numtrouble > 1) then
        send ("block", 0, empty, empty, empty) to requester
        send ("capture", 0, empty, empty, empty) to requester
      else
        if ( abs troublepiece[1] = QROOK) or ( abs troublepiece[1] = KROOK) or
                ( abs troublepiece[1] = QBISHOP) or ( abs troublepiece[1] = KBISHOP) or
                                                    ( abs troublepiece[1] = QUEEN) then
          let block = blockpiece(requester, neighbors[troubledir[1]], row, col,
                                  troublepiece[1] = troubleref[1], troubledir[1], whitesturn)
        else
          send ("block", 0, empty, empty, empty) to requester
        fi
        let capture = capturepiece(requester, row, col, troublepiece[1],
                                                                troubleref[1] = whitesturn)
      fi
    fi
]

collectmoves(requester, neighbor : ref, index : int, whitesturn : bool) ::
[   let replyrcvd = false
    let movercvd = false
    let numtrouble = 0
    let troublepiece = vector 0..NUMPIECES of int
    let troubleref = vector 0..NUMPIECES of ref
    let troubledir = vector 0..NUMPIECES of int
    let nummoves = 0
    let direction = vector 0..NUMPIECES of int
    let piece = 0
    [   if not (replyrcvd and movercvd) then
        [   case (tag : sym) of
            "moveking" : (Nummoves : int, Direction[0..Nummoves] : int)
                         movercvd = true
                         nummoves = Nummoves
                         direction[0..nummoves] = Direction[0..Nummoves]
            "reply"    : (Piece : int, Numtrouble : int, Troublepiece[0..Numtrouble] : int,
                                                         Troubleref[0..Numtrouble] : ref,
                                                         Troubledir[0..Numtrouble] : int)
                         replyrcvd = true
                         piece = Piece
                         numtrouble = Numtrouble
                         troublepiece[0..numtrouble] = Troublepiece[0..Numtrouble]
                         troubleref[0..numtrouble] = Troubleref[0..Numtrouble]
                         troubledir[0..numtrouble] = Troubledir[0..Numtrouble]
        ]
```

```
        repeat
      fi
    ]
    if (numtrouble = 0) and ((piece = EMPTY) or ((piece < 0) = whitesturn)) then
      nummoves = nummoves+1
      direction[nummoves] = index
    fi
    send ("moveking", nummoves, direction[0..nummoves]) to requester
]

moveking(requester, neighbors[N..NW] : ref, whitesturn : bool, kingrow, kingcol : int) ::
[ let i = 1
  let current = requester
  [ if (i ≤ NUMDIR) then
      current = collectmoves(current, neighbors[i], i, whitesturn)
      send ("check?", current, whitesturn) to neighbors[i]
      i = i+1; repeat
    fi
  ]
  let empty = vector 0..0 of int
  send ("moveking", 0, empty) to current
]

[ (console : ref)
  let whitesturn = true
  let row = vector -NUMPIECES..NUMPIECES of int
  let i = 1
  [ if (i ≤ NUMPIECES) then
      row[i] = 0
      i = i+1; repeat
    fi
  ]
  let col = row
  row[WHITE*KING] = 1; col[WHITE*KING] = 6
  row[WHITE*QROOK] = 2; col[WHITE*QROOK] = 4
  row[WHITE*KROOK] = 8; col[WHITE*KROOK] = 1
  row[WHITE*QBISHOP] = 1; col[WHITE*QBISHOP] = 4
  row[WHITE*PAWN1] = 2; col[WHITE*PAWN1] = 7
  row[WHITE*PAWN2] = 4; col[WHITE*PAWN2] = 2
  row[WHITE*QKNIGHT] = 3; col[WHITE*QKNIGHT] = 4
  row[WHITE*KKNIGHT] = 5; col[WHITE*KKNIGHT] = 5
  row[BLACK*KING] = 8; col[BLACK*KING] = 8
  row[BLACK*QUEEN] = 4; col[BLACK*QUEEN] = 6
  row[BLACK*KROOK] = 1; col[BLACK*KROOK] = 1
  row[BLACK*KBISHOP] = 4; col[BLACK*KBISHOP] = 8
```

```
row[BLACK*PAWN1] = 2; col[BLACK*PAWN1] = 8
let piecename = vector −NUMPIECES..NUMPIECES of sym
piecename[EMPTY] = "_"
piecename[QKNIGHT] = "wN"
piecename[KKNIGHT] = "wN"
piecename[QROOK] = "wR"
piecename[KROOK] = "wR"
piecename[QBISHOP] = "wB"
piecename[KBISHOP] = "wB"
piecename[QUEEN] = "wQ"
piecename[KING] = "wK"
piecename[PAWN1] = "wP"
piecename[PAWN2] = "wP"
piecename[PAWN3] = "wP"
piecename[PAWN4] = "wP"
piecename[PAWN5] = "wP"
piecename[PAWN6] = "wP"
piecename[PAWN7] = "wP"
piecename[PAWN8] = "wP"
piecename[−QKNIGHT] = "bN"
piecename[−KKNIGHT] = "bN"
piecename[−QROOK] = "bR"
piecename[−KROOK] = "bR"
piecename[−QBISHOP] = "bB"
piecename[−KBISHOP] = "bB"
piecename[−QUEEN] = "bQ"
piecename[−KING] = "bK"
piecename[−PAWN1] = "bP"
piecename[−PAWN2] = "bP"
piecename[−PAWN3] = "bP"
piecename[−PAWN4] = "bP"
piecename[−PAWN5] = "bP"
piecename[−PAWN6] = "bP"
piecename[−PAWN7] = "bP"
piecename[−PAWN8] = "bP"
let boardrow = vector 1..NUMROWS of sym
i = NUMROWS
[  if (i > 0) then
      let j = 1
      [  if (j ≤ NUMROWS) then
            boardrow[j] = piecename[EMPTY]
            let k = −NUMPIECES
            [  if (k ≤ NUMPIECES) then
                  if (row[k] = i) and (col[k] = j) then
                     boardrow[j] = piecename[k]
```

```
                else
                    k = k+1; repeat
                fi
            fi
        ]
        j = j+1; repeat
    fi
  ]
  send (boardrow) to console
  send () to console
  send () to console
  i = i-1; repeat
 fi
]
let start = buildboard(self, row, col, whitesturn)
[  (king : ref, neighbors[N..NW] : ref)
  let color = BLACK
  if whitesturn then
    color = WHITE
  fi
  let move = moveking(self, neighbors, whitesturn,
       row[color*KING] = col[color*KING])
  send ("check?", notmoveking(self, king, neighbors, whitesturn, row, col), whitesturn) to king
  let numrcvd = 0
  let status = "king in check"
  let reply = "checkmate"
  let nummoves = 0
  let direction = vector 0..NUMPIECES of int
  let numblockers = 0
  let blockers = vector 0..NUMPIECES of int
  let bdestrow = blockers
  let bdestcol = blockers
  let numcapturers = 0
  let capturers = vector 0..NUMPIECES of int
  let cdestrow = capturers
  let cdestcol = capturers
  [  if (numrcvd < 3) then
       [   case (tag : sym) of
           "notmoveking" : ()
                           numrcvd = numrcvd+2
                           reply = "not checkmate"
                           status = "king not in check"
           "moveking"    : (Nummoves, Direction[0..Nummoves] : int)
                           numrcvd = numrcvd+1
                           nummoves = Nummoves
```

```
                    direction[0..nummoves] = Direction[0..Nummoves]
        "block"     : (Numblockers, Blockers[0..Numblockers],
                                                  Destrow[0..Numblockers],
                                              Destcol[0..Numblockers] : int)
                    numrcvd = numrcvd+1
                    numblockers = Numblockers
                    blockers[0..numblockers] = Blockers[0..Numblockers]
                    bdestrow[0..numblockers] = Destrow[0..Numblockers]
                    bdestcol[0..numblockers] = Destcol[0..Numblockers]
        "capture"   : (Numcapturers, Capturers[0..Numcapturers],
                                                  Destrow[0..Numcapturers],
                                              Destcol[0..Numcapturers] : int)
                    numrcvd = numrcvd+1
                    numcapturers = Numcapturers
                    capturers[0..numcapturers] = Capturers[0..Numcapturers]
                    cdestrow[0..numcapturers] = Destrow[0..Numcapturers]
                    cdestcol[0..numcapturers] = Destcol[0..Numcapturers]
    ]
    repeat
  fi
]
let dirrow = vector 1..NUMDIR of int
let dircol = vector 1..NUMDIR of int
dirrow[N] = -1; dirrow[NE] = -1; dirrow[E] = 0; dirrow[SE] = 1
dirrow[S] = 1; dirrow[SW] = 1; dirrow[W] = 0; dirrow[NW] = -1
dircol[N] = 0; dircol[NE] = 1; dircol[E] = 1; dircol[SE] = 1
dircol[S] = 0; dircol[SW] = -1; dircol[W] = -1; dircol[NW] = -1
let file = vector 1..NUMROWS of sym
file[1] = "a"; file[2] = "b"; file[3] = "c"; file[4] = "d"
file[5] = "e"; file[6] = "f"; file[7] = "g"; file[8] = "h"
send (status) to console
send () to console
if (status = "king in check") then
  if (nummoves > 0) then
    send ("Move king to escape check") to console
    reply = "not checkmate"
    i = 1
    [  if (i ≤ nummoves) then
         send (piecename[color*KING], file[col[color*KING]+dircol[direction[i]]],
                                      row[color*KING]+dirrow[direction[i]]) to console
         i = i+1; repeat
       fi
    ]
  else
    send ("King cannot be moved to avoid check") to console
```

```
            fi
            send () to console
            if (numblockers > 0) then
               send ("Block attacking piece") to console
               reply = "not checkmate"
               i = 1
               [  if (i ≤ numblockers) then
                     send (piecename[blockers[i]], file[bdestcol[i]], bdestrow[i]) to console
                     i = i+1; repeat
                  fi
               ]
            else
               send ("Attacking piece(s) cannot be blocked") to console
            fi
            send () to console
            if (numcapturers > 0) then
               send ("Capture attacking piece") to console
               reply = "not checkmate"
               i = 1
               [  if (i ≤ numcapturers) then
                     send (piecename[capturers[i]], ":", file[cdestcol[i]], cdestrow[i]) to console
                     i = i+1; repeat
                  fi
               ]
            else
               send ("Attacking piece(s) cannot be captured") to console
            fi
            send () to console
         fi
         send (reply) to console
         send () to console
      ]
   ]
```

Bibliography

[1] W.C. Athas and C.L. Seitz, *The Cantor User Report, Version 2.0*, Dept. of Computer Science, California Institute of Technology, Technical Report 5232, Jan. 1987.

[2] W.C. Athas, *Fine Grain Concurrent Computations*, Dept. of Computer Science, California Institute of Technology, Technical Report 5242, May 1987.

[3] W.C. Athas and C.L. Seitz, "Multicomputers: Message-Passing Concurrent Computers," *IEEE Computer*, Aug. 1988.

[4] A. Cayley, "On the Mathematical Theory of Isomers," *Philosophical Magazine*, Vol. 47, 1874, pp. 444-446.

[5] K. Mani Chandy and Jay Misra, "Distributed Computation on Graphs: Shortest Path Algorithms," *Communications of the ACM*, Nov. 1982.

[6] W.J. Dally, *A VLSI Architecture for Concurrent Data Structures*, Dept. of Computer Science, California Institute of Technology, Technical Report 5209, March 1986.

[7] E.W. Dijkstra and C.S. Scholten, "Termination Detection for Diffusing Computations," *Information Processing Letters*, Aug. 1980.

[8] E.W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Information Processing Letters*, Aug. 1980.

[9] R.W. FLoyd, "Algorithm 97: Shortest Path," *Communications of the ACM*, June 1962.

[10] L.J. Guibas, H.T. Kung, and C.D. Thompson, "Direct VLSI Implementation of Combinatorial Algorithms," *Proceedings, Caltech Conference on VLSI*, 1979.

[11] C.A.R. Hoare,"Quicksort," *Computer Journal*, Vol. 5, 1962.

[12] D. Knuth, *The Art of Computer Programming*, Vol. 1, (second edition) Addison-Wesley, 1973, pp. 386-388.

[13] F. Preparata and Michael Shamos, *Computational Geometry*, Springer-Verlag, 1985.

[14] C.L. Seitz, "The Cosmic Cube," *Communications of the ACM*, Vol. 28, No. 1, pp. 22-33, January 1985.

[15] J. Seizovic, *The Reactive Kernel*, Dept. of Computer Science, California Institute of Technology, Technical Report, Oct. 1988.

[16] Jan L.A. van de Snepscheut, "A Derivation of a Distributed Implementation of Warshall's Algorithm," *Science of Computer Programming*, July 1986.

[17] S. Warshall, "A Theorem on Boolean Matrices," *Journal of the ACM*, Jan. 1962.