Fine Grain Concurrent Computations

Thesis by

William C. Athas

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California

1987

(Submitted May 1, 1987)

Acknowledgments

Thanks, Chuck

Abstract

This thesis develops a computational model, a programming notation, and a set of programming principles to further and to demonstrate the practicality of programming fine grain concurrent computers.

Programs are expressed in the computational model as a collection of *definitions* of autonomous computing agents called *objects*. In the execution of a program, the objects communicate data and synchronize their actions exclusively by message-passing. An object executes its definition only in response to receiving a message, and its actions may include sending messages, creating new objects, and modifying its own internal state. The number of actions that occur in response to a message is finite; Turing computability is achieved not within a single object, but through the interaction of objects.

A new concurrent programming notation *Cantor* is used to demonstrate the cognitive process of writing programs using the object model. Programs for numerical sieves, sorting, the eight queens problem, and Gaussian elimination are fully described. Each of these programs involve up to thousands of objects in their exectuion. The general programming strategy is to first partition objects by their overall behavior and then to program the behaviors to be self-organizing.

The semantics of Cantor are made precise through the definition of a formal semantics for Cantor and the object model. Objects are modelled as finite automata. The formal semantics is useful for proving program properties and for building frameworks to capture specific properties of object programs. The mathematical frameworks are constructed for building object graphs independently of program execution and for systematically removing objects that are irrelevant to program execution (garbage collection).

The formal semantics are complemented by experiments that allow one to study the dynamics of the execution of Cantor programs on fine grain concurrent computers. The clean semantics of Cantor suggests simple metrics for evaluating the execution of concurrent programs for an ideal, abstract implementation. Program performance is also evaluated for environments where computing resources are limited. From the results of these experiments, hardware and software architectures for organizing fine grain message-passing computations is proposed, including support for fault tolerance and for garbage collection.

Contents

List of Figures

Chapter 1

Introduction

Concurrent computers are able to exploit concurrent behavior in computer programs over a wide range of size, or *granularity*, of the concurrent parts. Concurrency occurring at the sub-part level is difficult to exploit except for concurrency intrinsic to the computing elements, e.g., bit-wise parallelism. Therefore, from the perspective of exploiting concurrency, there is a strong impetus to keep the size of the program parts small.

There are also no fundamental difficulties in building fine grain concurrent computers. The high-tech world of the last quarter of the 20th century abounds with small computers, i.e., instruction processors with only small amounts of storage. These small computers occur in such diverse places as wristwatches, automotive engines, microwave ovens, and childrens' dolls. By and large, these computers do not have any direct interaction with other computers; at best, these small computers are *loosely* coupled. Due to recent developments in high performance, message-passing interconnection networks, automata-based routing mechanisms have been designed whose whose performance in delivering message packets between computers rivals the rate at which the instruction processors inside the small computers can execute instructions [12].

The fundamental problem with fine grain concurrent computers is the gap in the body of knowledge about how to organize computations for these machines. Programming techniques for organizing sequential programs at the level of storage locations and logical/arithmetic operations are copious. Programming techniques for organizing concurrent programs as collections of processes where each process is a sequential program are also well understood, though not to the same degree of maturity as the sequential program. In both cases the infinite, or "unbounded" character of the computation is separately encapsulated into the program space of each process. For example, with recursive functions, the depth of the recursion is not a specification of the function because the depth may be data dependent. The assumption is that the recursion depth is infinite.

Organizing programs as collections of processes where one or more processes reside on each small computer or *processing node* is flawed because the illusion of an infinite program space per process cannot be realized. The storage space per node is quite small. However, because the nodes are small, the number of nodes available is large. The solution is obvious: shift the unbounded character of computations from the internal behavior of each process to between the processes. In other words, have each process occupy finite space and require only a finite amount of time to process a message, but let an unbounded number of processes be applied to a computation.

1.1 Objects

Processes that have these properties will be called objects. The only actions permitted of an object are: changing the state of the object, sending messages, and creating new objects. Objects are message-driven in that they are normally at rest, waiting for a message to arrive. An object can process only one message at a time, and the number of messages sent and number of objects created in response to receiving a single message is finite.

The action of modifying the state of an object has the conventional interpretation found in programming languages: namely, executing assignment statements on the variables private to the object, performing logical/arithmetic operations on the private variables, and making simple control flow decisions. Neither indefinite iteration nor unbounded recursion is allowed inside an object because of the requirement that an object processes a message in a bounded amount of time.

Because communication is not instantaneous, the action of sending a message from an object is only a promise that the message will be sent. The message is guaranteed to be delivered eventually to its destination, and messages sent between two objects in direct communication will have message order preserved.

The action of creating a new object is like the send action in that it is only a promise to eventually create a new object. Completion of the *new* action immediately yields a *reference* to the new object, although the instance of the new object may not yet exist. The reference value, however, may be computed on and be shared with other objects.

The three types of actions: *send*, *new*, and update state, are sufficient to express any program, not with a single object, of course, but by a collection of interacting objects. For example, a recursive function can be expressed as a collection of objects that behave as a stack, thereby providing a mechanism to implement the recursion.

This object model is related to the actor model of computation. The fundamental difference between the two is that actors [6] [10] [38] [2] are recursive constructions; hence, an individual actor can exhibit infinite internal behavior. The object model used here is comprised only of objects with finite internal behavior. The behavior of objects is a proper subset of the behavior of actors. The primary advantage of the actor model over the object model is its power of abstraction. On the other hand, the object model has a simpler analytical formulation.

1.2 Object Computations

The internal behavior of objects is constrained to be sequential because the internal state can be modified over time. When an object processes a message, the object is guaranteed to generate only a finite number of messages, to produce a finite number of new objects, and then to prepare itself to receive another message. Because the response to a single message is finite, the interval between receiving the message and readying to receive another message is considered indivisible. To illustrate, Figure 1.1 depicts the first three *states* of a sample object computation. The picture of Figure 1.1(a) shows a message travelling from object O_1 to object O_2. The objects are depicted as circles and the message as a triangle. The arrows denote that the object or message at the tail of the arrow has a reference to the object at the head of the arrow. A reference allows one object to send another object a message. Each message contains a special reference to its destination object. This special reference is drawn on one vertex of the triangle.

The response of O_2 after receiving M_1 is shown in Figure 1.1(b). O_2 produces two new objects O_3 and O_4, and sends messages to both of these new objects. The interval between O_2 receiving message M_1 and readying itself for another message cannot be observed. The state of the computation is well defined only before and after the message is received, not during. The object graph of Figure 1.1(b) shows concurrency since both O_3 and O_4 can process their messages concurrently.

Figure 1.1(c) depicts the computation after M_2 is received by O_4. Notice that one of the state modifying operations inside of O_4 resulted in the reference to O_2 being removed. There is no possible way for O_2 to receive a message unless some other object were able to *forge* a reference to O_2. The possibility of forging references in effect allows any object to send any other object a message, in contrast to having the set of acquaintances of an object limited to the transitive closure of the set of objects to which it can send messages. Forging references is not allowed in the object model because it makes the set of objects completely interconnected.

The object model has the required characteristic of placing the burden of expanding the computation at the level of interaction between objects instead of inside of objects. From the viewpoint of a fine grain computer, the problem of dynamic growth during a computation becomes one of prudently assigning objects to the nodes of the concurrent computer. The assignment task is complicated by myriad details about the fine grain computer, i.e., interconnection topology and message channel characteristics. Evidence is presented in this thesis that programmers of fine grain computers do not need to be aware of the assignment task since automatic mechanisms work virtually as well.

An important feature of the object model is the treatment of futures, where the word "futures" is used in the commodities trading sense. The use of futures has been proposed [30] as a mechanism to moderate the concurrency derived from a program by explicitly qualifying the futures as either "lazy" or "eager" when a program is written. This approach to futures places premature restrictions on the use of futures. The full potential of futures is to view them from the perspective of creating new objects, as described earlier. In the object model, a reference to an object can exist before the instance of the corresponding objects exists. The only way to be sure that an object has been created is to receive a message from it. References without objects are futures. The runtime system supporting the object model has a great deal of latitude in the assignment process since only the instance of an object is necessarily assigned to a processing node. The future can be an independent quantity. The manipulation of futures independently of the computation is called *future flow* to distinguish it from the computation or *program flow*.

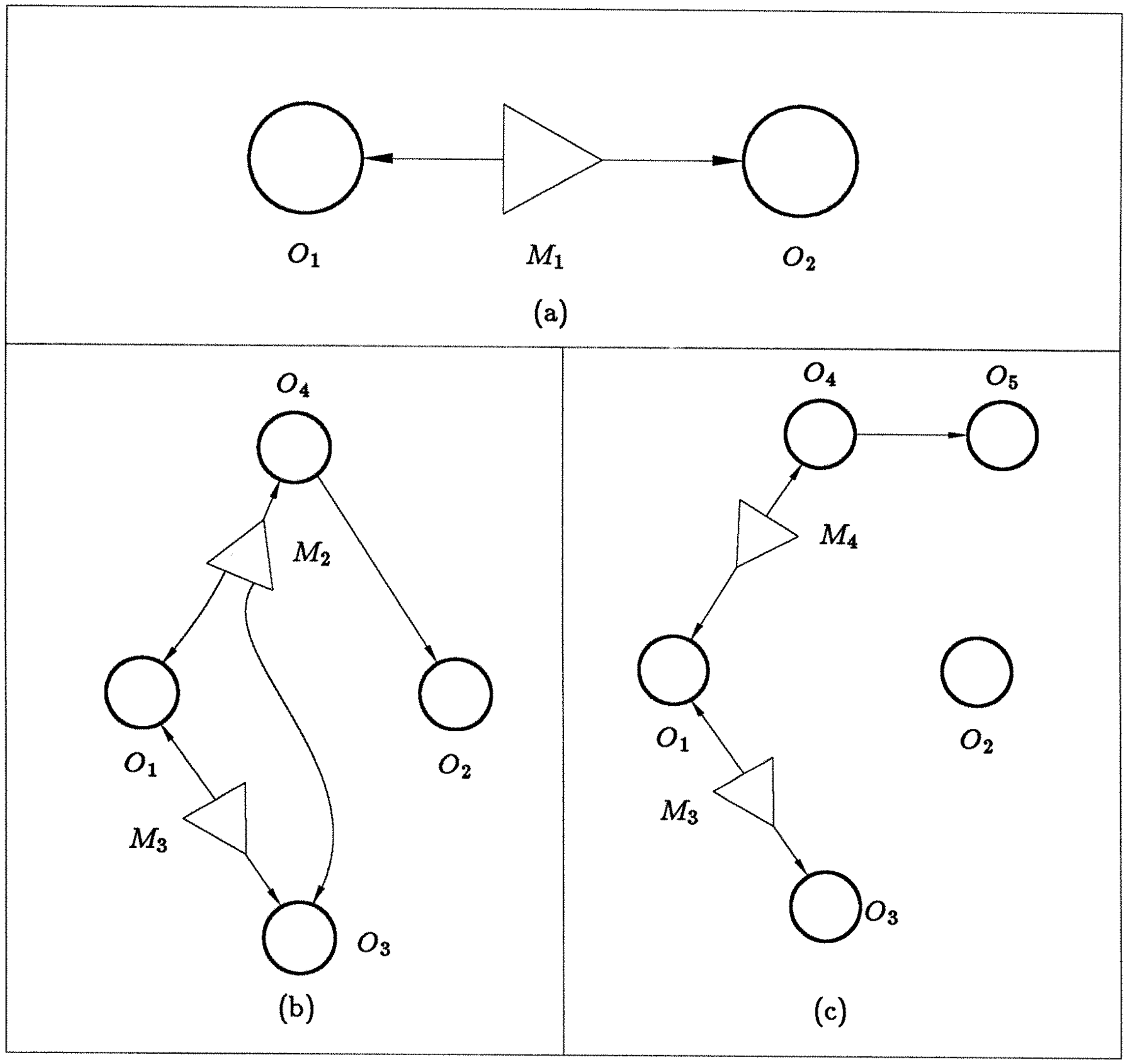


Figure 1.1: Snapshots of an Object Computation

1.3 Thesis Flow

As might be expected, writing programs in the object model is substantially different from writing conventional programs for sequential computers. One approach to the object model is to view it as a compilation target in which high level concurrent programming notations are compiled into a set of object definitions that are then used to build up the object computation. A second approach is to program directly at the level of the object model. This second approach is the one taken in this thesis, since all of the capabilities and limitations of the object model can then be explored.

A compact programming notation called *Cantor* is presented in Chapter 2. The semantics of Cantor are discussed informally, and several small programs are presented to demonstrate the various syntactic constructs. The feasibility of writing programs directly at the abstraction level of the object model is demonstrated by presenting Cantor object definition to implement conventional data structures such as queues and arrays. Six complete medium-size programs are then discussed in detail to demonstrate the cognitive process of writing Cantor programs starting with only a definition of the program task.

The semantics presented informally in Chapter 2 are made precise in Chapter 3 by introducing a formal model for the semantics of the object model. Objects are modelled as finite automata whose state graphs and transition rules have special properties. A primitive notation for expressing object programs is defined in terms of the formal model. Interpretation rules expressed in the primitive notation are then defined for each of the syntactic constructs of Cantor. The objects of Cantor are therefore given a precise meaning by first translating the Cantor object into the primitive notation and then expanding each of the constructs of the primitive notation into an equivalent finite automaton representation.

The formal definition for Cantor provides a framework for program flow that is useful for proving program properties. A framework for future flow is derived from the program flow framework and is used to compute futures at compile time. The basis for future flow at the compilation stage is shown to be a special case of the task of constant propagation found in optimizing compilers. The results of future flow can be applied to the assignment task to make a speculative placement of future objects to processing nodes.

The complement to future flow is *after flow* or "garbage collection." The task of after flow is to determine the set of objects that have become irrelevant to the computation proper without disturbing the computation. The resources held by these objects may be recycled without affecting program semantics. The after flow framework is a simplification of the program flow framework. The only operation relevant to the after flow framework is transitive closure over object references. The execution rules for Cantor objects combined with the after flow framework permit uncomplicated garbage collection algorithms to be used effectively.

The analytical approach to object program behavior in Chapter 3 is useful for understanding the internal behavior of objects and also for inferring general properties of object programs. The behavior for complex programs consisting of thousands of objects is beyond the scope of the analytical approach. Chapter 4 defines an idealized, abstract implementation for Cantor programs to quantitatively measure the behavior of the programs introduced in Chapter 2. The program quantities measured include the amount of concurrency and the amount of message traffic extant in the programs under ideal conditions. The ideal conditions are then incrementally removed to experiment with the effects of limiting computing resources and applying different automatic assignment strategies. The results from Chapter 4 show that simple, automated assignment strategies perform virtually as well as the ideal strategies.

From the abstract implementations developed in Chapter 4, software and hardware architectures for Cantor and the object model are proposed in Chapter 5. Organizations for node computers suitable for ensemble machine architectures [35] [33] [34] are explored. These organizations include hosting multiple objects per processing node and also placing multiple Cantor engines on each node. The multiple Cantor engine per node organization is effectively a shared memory organization. The possibility of multiple nodes and multiple Cantor engines suggests methods for allowing object computations to proceed in the presence of faults. Hardware and software mechanisms for handling faults are discussed. Many of these mechanisms are also suitable for the garbage collection algorithms of Chapter 3.

Chapter 2

Programming with Concurrent Objects

Computations expressed in the object model require new modes of expression. To represent these new modes in existing programming notations is awkward because of the substantial difference between the object model and the programming models used for sequential computers. The existing programming models implicitly assume that program behavior is sequential and any provisions for concurrency must be explicitly stated. The object model assumes that program behavior is inherently concurrent; to express sequential behavior requires deliberate effort.

This chapter introduces the programming notation Cantor, a concrete realization of a programming model based upon the object model. Cantor and its semantics are the basis for the program experiments and analysis performed in this thesis. Essentially, the properties of the object model become the semantics of Cantor's syntactic constructs. This chapter details the differences between the Cantor programming model and the object model, provides a cursory overview of Cantor's syntax, and then uses Cantor to express some object programs.

2.1 The Programming Model

The Cantor programming model closely follows the object model but also imposes additional constraints that do not contradict the object model, but rather limit the range of expression of the computations. The limitations arise when a framework is chosen to bridge the gap between the representation of programs, i.e., syntactic constructs and program semantics.

The overall structure for a program is a collection of object *definitions*. The Cantor definition is akin to the class concept found in object-oriented languages except that there is no inheritance of attributes between object definitions. Each definition has a unique name and a template that is used to build new objects. The template specifies a set of private variables for new object. These variables are assigned values when the object is created. The template also specifies a set of one or more descriptions that define the response of an object to a message. The description contains a set of message variables that define the *expected* contents of the next message to be received and a list of actions to be performed in response to the message.

When an object is created, the object is assigned an initial set of values for its private variables and an initial description to process the first message that will be received for the object. After the first message is received, the object can send messages, create new objects, and change its internal state. The internal state change consists of modifying the private variables and possibly switching to a different description to use for processing the next message. The set of descriptions that an object can switch is determined when the object definition is compiled.

The object model does not address the genesis of object computations. The programming model and its implementation assume that in the beginning there is only a single message and a single primordial object from which all other messages and objects are descendant. The destination of the first message is the primordial object. This first message imports the initial set of external references for the set of external objects that will be used by the object program to communicate with the outside world. If the computation is to expand, then the primordial object must create new objects. If the computation is to become concurrent, then multiple messages must be sent to different destinations. The termination condition for a program occurs when there are no messages left to be processed anywhere. Programs that exhibit cycles of object references with messages perpetually flowing around the cycle never terminate.

2.2 Cantor

This section provides a cursory explanation of the syntax and semantics of Cantor, the object language that has been designed and implemented for the investigations reported in this thesis. The expository style is to progress gradually through the BNF description of Cantor depicted in Figure 2.1. For a more thorough and gentler introduction, the reader is encouraged to study Chapter 2 of the Cantor User Report [5].

A program is textually described by a set of definitions. This structure is captured in the first three rules of the BNF specification:

$$\begin{aligned}
&\langle \textit{program} \rangle && \Longrightarrow && \langle \textit{object definition} \rangle^{*}\ \langle \textit{description} \rangle \\
&\langle \textit{object definition} \rangle && \Longrightarrow && \langle \textit{object name} \rangle\ \langle \textit{persistent list} \rangle\ ::\ \langle \textit{description} \rangle \\
&\langle \textit{description} \rangle && \Longrightarrow && (\ {*}[\ \mid\ [\ )\ \langle \textit{body} \rangle\ ]
\end{aligned}$$

For the ⟨*program*⟩ rule, the single ⟨*description*⟩ following the list of ⟨*object definition*⟩s specifies the behavior for the primordial object. This object is by convention called the main object. A simple but complete Cantor program that uses only this primordial object is:

[(console) <u>send</u> ("Hello World") <u>to</u> console]

The effect of running this program is to print "Hello World" on the user's output device. The matching pair of square brackets denotes a description. To understand the meaning of the description, the BNF rule for ⟨*body*⟩ needs to be expanded:

$$\begin{aligned}
&\langle \textit{body} \rangle && \Longrightarrow && \langle \textit{sequence} \rangle \mid \langle \textit{case} \rangle \mid \langle \textit{description} \rangle^{+} \\
&\langle \textit{sequence} \rangle && \Longrightarrow && \langle \textit{message list} \rangle\ \langle \textit{statement} \rangle^{*} \\
&\langle \textit{case} \rangle && \Longrightarrow && \underline{\text{case}}\ (\ \langle \textit{name} \rangle\ )\ \underline{\text{of}}\ \langle \textit{casebody} \rangle^{+}
\end{aligned}$$

```
⟨program⟩            ⟹  ⟨object definition⟩* ⟨description⟩
⟨object definition⟩  ⟹  ⟨object name⟩ ⟨persistent list⟩ : : ⟨description⟩
⟨description⟩        ⟹  ( *[ | [ ) ⟨body⟩ ]
⟨body⟩               ⟹  ⟨sequence⟩ | ⟨case⟩ | ⟨description⟩+
⟨sequence⟩           ⟹  ⟨message list⟩ ⟨statement⟩*
⟨case⟩               ⟹  case ( ⟨variable name⟩ ) of ⟨case entry⟩+
⟨case entry⟩         ⟹  ⟨selector⟩ : ⟨sequence⟩
⟨statement⟩          ⟹  ⟨if⟩ | ⟨let⟩ | ⟨send⟩ | ⟨assign⟩ | ⟨control⟩ | ⟨description⟩
⟨if⟩                 ⟹  if ⟨expression⟩ then ⟨statement⟩+ { else ⟨statement⟩+ } fi
⟨let⟩                ⟹  let ⟨variable name⟩ = ⟨expression⟩
⟨send⟩               ⟹  send ⟨list⟩ to ⟨expression⟩
⟨assign⟩             ⟹  ⟨name⟩ := ⟨expression⟩
⟨control⟩            ⟹  exit | repeat | become ⟨expression⟩
⟨persistent list⟩    ⟹  ⟨name list⟩
⟨message list⟩       ⟹  ⟨name list⟩
⟨name list⟩          ⟹  () | ( ⟨name⟩ { , ⟨name⟩ }* )
⟨expression⟩         ⟹  ⟨expr1⟩ { (or | xor ) ⟨expr1⟩ }*
⟨expr1⟩              ⟹  ⟨expr2⟩ { and ⟨expr2⟩ }*
⟨expr2⟩              ⟹  ⟨expr3⟩ { ( = | <> | < | > | <= | >= ) ⟨expr3⟩ }*
⟨expr3⟩              ⟹  ⟨expr4⟩ { ( + | - ) ⟨expr4⟩ }*
⟨expr4⟩              ⟹  ⟨primitive⟩ { ( * | / | mod ) ⟨primitive⟩ }*
⟨primitive⟩          ⟹  ⟨variable name⟩ | ⟨selector⟩ | ⟨reference⟩ | ⟨real⟩ |
                         abs ⟨primitive⟩ | not ⟨primitive⟩ | ( ⟨expression⟩ )
⟨selector⟩           ⟹  ⟨integer⟩ | ⟨logical⟩ | ⟨symbol⟩
⟨reference⟩          ⟹  self | ⟨object name⟩ ⟨list⟩
⟨list⟩               ⟹  () | ( ⟨expression⟩ { , ⟨expression⟩ }* )
⟨symbol⟩             ⟹  " ⟨char⟩* "
⟨logical⟩            ⟹  true | false
⟨name⟩               ⟹  ⟨ident⟩
```

Figure 2.1: BNF Description for Cantor

The body of an object definition is either a message list followed by a sequence of zero or more statements, a <u>case</u> construct, or a series of nested descriptions. For the **main** object of the "Hello World" program, the body is an instance of the ⟨*sequence*⟩ rule: comprised of a message list containing a single component named **console** followed by a <u>send</u> statement. The first message sent to the primordial object **main** is expected to contain a reference to an external object capable of printing messages.

The general interpretation of a ⟨*description*⟩ is this: Whenever an executing object encounters an open square bracket followed by message list, viz. a new description, execution of the object stops until a new message arrives for it. Initially an object starts executing at the outermost left bracket, and is therefore initially waiting for a message. When a message arrives, it is copied into the message list of the body of the description. The object then executes statements until the outermost close square bracket is encountered, or until another description is reached.

For the **main** object, the message list contains references to external objects that are passed into the Cantor program by the runtime system hosting the program. For the "Hello World" program, the message from the outside contains a reference to an object internally named **console**. A message sent to the reference contained in **console** will cause the message contents to be displayed on some output device.

The statements of Cantor correspond to the list of actions that an object can perform. A statement can change the value of an internal variable, alter the control flow within the object, or cause an external effect by sending a message or creating a new object. The entire set of possible statements is given by the BNF for a ⟨*statement*⟩:

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

The **main** object of the "Hello World" program contains only a single statement, a <u>send</u> statement. The syntax for the <u>send</u> statement is the following:

⟨*send*⟩ ⟹ <u>send</u> ⟨*list*⟩ <u>to</u> ⟨*expression*⟩
⟨*expression*⟩ ⟹ ... ⟨*primitive*⟩ ...
⟨*primitive*⟩ ⟹ ⟨*variable name*⟩ | ⟨*reference*⟩ ...
⟨*reference*⟩ ⟹ <u>self</u> | ⟨*object name*⟩ ⟨*list*⟩

The meaning of the <u>send</u> statement is to copy the contents of ⟨*list*⟩ into a message and then send the message to the object whose reference is the value of ⟨*expression*⟩.

Lists have the following syntax:

⟨*list*⟩ ⟹ () | (⟨*expression*⟩ { , ⟨*expression*⟩ }*)

Expressions in Cantor can take on a variety of forms as directed by the precedence structure of the arithmetic and logical operators built into Cantor. However, all expressions must evaluate to one of the five primitive data types that are built into Cantor. The five types of primitive values are:

- reference, the address of another object.

- integer, e.g., 42.
- real number, e.g., 3.14159265.
- symbol, e.g., "Hello World".
- logical (boolean), viz. true or false

All Cantor values are tagged with type information. When the contents of a list are copied into a message, each expression is evaluated to yield a value, and these values, each tagged with its type, make up the contents of the message.

For the ⟨*send*⟩ rule, the destination of the message is denoted by an ⟨*expression*⟩ that must evaluate to a reference value. Any other value type is a programming error. For the "Hello World" program, console is a reference, "Hello World" is a literal symbol, and the effect of the single send statement is to dispatch this message to console. The message is interpreted by the external object referred to as console as a list containing one symbol, and this symbol is printed.

Referring again to the BNF, a reference value can in general be the value associated with a variable name, e.g., a component of the persistent list for the object, or a component of the message list for the current description, or can be introduced by the rule for ⟨*reference*⟩. This rule states that a reference can be produced by the keyword self, which produces a reference value for the object where the keyword self occurs, or can be manufactured by composing the name of an object definition with a list. The effect of this composition is to create a new instance of the object and to "install" the list into the persistent list of the new object.

To illustrate the process of objects creating new objects, consider the following program to compute the factorial function:

```
factorial(reply, n)::
[ ()
  if n<2
    then send (1) to reply
    else send () to factorial(self , n-1)
         [ (m) send (n*m) to reply]
  fi
]

[ (console) send () to factorial(console, 6) ]
```

The main object receives the console message, creates a new object from the factorial definition, and then sends the new object an empty message. The expression:

factorial(console, 6)

yields a new object with two private variables; the value of console is assigned to reply and the value of 6 is assigned to n. Because the private variables contain all the necessary information, i.e., a value of n and a reply link, the empty message sent to the factorial object is used only to make the object perform the sequence of statements inside the description.

The definition of the factorial object follows almost directly from the recursive definition. If n is 0 or 1, the value 1 is replied immediately to the caller which contains the value of console. For values of n greater than 1, the recursive step is taken. A new factorial object is created with a reply link of self and a value of n - 1. The keyword self denotes the reference value of the current object. The object must then rendezvous with the reply from the new object. This rendezvous is performed by nesting a description inside of the current description. The BNF allows this recursive action because one of the possibilities for the ⟨*statement*⟩ rule is the ⟨*description*⟩ rule.

After the rendezvous, the received integer value and the value of n are multipled together and sent to reply. The object then reaches the outermost square bracket. At this point, because the object has no description to use next, the object self-destructs. An object can be directed to reuse a description by beginning the description with *[instead of [, or by using the repeat command. In a complementary fashion, the exit command is used to override a *[and causes the current description to be exited.

For nested descriptions, Cantor supports lexical scoping on variables. The factorial program could equally have been written:

```
factorial()::
[ (reply, n)
  if n<2
    then send (1) to reply
    else send (self, n-1) to factorial()
         [ (m) send (n*m) to reply ]
  fi
]


[ (console) send (console, 6) to factorial() ]
```

Cantor also allows local variables to be declared inside of descriptions via the ⟨*let*⟩ statement. The factorial program could also be written:

```
factorial() ::
[ (reply, n)
  if n<2
    then send (1) to reply
    else send (self , n-1) to factorial()
         [ (m) send (n*m) to reply ]
  fi
]

[ (console)
```

```
    let f = factorial()
    send (console, 6) to f
  ]
```

The statement:

```
                         let f = factorial()
```

allocates a new variable f and assigns to f the reference value of the new factorial object. The assignment of the reference value to f suggests a potential problem for the three factorial programs. Suppose that the main object is rewritten as:

```
[ (console)
  let f = factorial()
  send (console, 6) to f
  send (console, 7) to f
]
```

Message order is preserved for messages sent between two objects in direct communication. After the factorial object processes the first message, it then waits for the rendezvous from the recursion; however, the (console, 7) message may very well arrive next. The factorial object is expecting a message containing a single integer but receives a message containing a reference and an integer. A second problem is that the factorial object is expecting two messages but receives 3. After the second message is received, the object self-destructs because no replacement description is specified.

The problem can be solved by factoring out the nested description used for the rendezvous and multiply step within a separate object called mult. Consider the following factorial program:

```
multiply(reply, n) :: [ (m) send (n*m) to reply ]


factorial():: *[ (reply, n)
                 if n<2
                   then send (1) to reply
                   else let m = multiply(reply, n)
                        send (m, n-1) to self
                 fi
]


[ (console)
  let f = factorial()
  send (console, 6) to f
  send (console, 7) to f
]
```

This program uses only a single factorial object that is shared for all of the factorial requests generated by the two initial requests of 6 and 7. The *[form of description is used so that the same description is reused. For n greater than 1, the recursive step involves creating a new multiply to hold the value of reply and n. A message is then sent to self with a reply link of the new multiply object.

The final construct to be covered is the case construct. The case construct is an alternate form of the description. The chief reason for the inclusion of the case construct is to facilitate the tagging of messages. The use of the tag provides a mechanism to vary the interpretation of the contents of a message based upon the value of the first component of the message. The tag also serves the role of the "method" or "attribute" selector used in object-oriented programming. The tag of the message is used to invoke an attribute of an object by dispatching on the tag to one of the case entries. Each entry corresponds to a method of the object. The first variable of the message is the method selector. The ⟨*selector*⟩ is enclosed in parentheses and follows the case keyword. The value of ⟨*selector*⟩ must be either a symbol, integer, or logical. This value is used to select which of the case entries that will be executed in response to the message. If none of the selectors match the value of the first message variable, then the message is discarded. If a selector matches the value of the first message variable, then the message list and sequence of statements following the selector are executed. Thus the message list for the case construct is separated into two parts. The first message variable is enclosed in parentheses following the case keyword while the remainder of the message list follows the selector and has the same syntax as the ⟨*sequence*⟩.

A simple example that uses the case construct is the task of maintaining a sorted list of integers using a linked list of objects. For the sorted list there are two commands: insert integer and check if integer is in list. The test command requires that a value of true or false be sent to the caller. The insert command requires that the value be inserted. The case construct allows the first component of a message to be examined before the rest of the message list is interpreted. The object definition of Figure 2.2 uses the case construct for the sorted list.

The first component of a message received by a slist object is interpreted as cmd. The value of cmd must be integer, boolean, or a symbol. For the slist object, two symbols are used for the possible selectors: "insert" and "check". If value of the cmd selector is "insert", the second component of the message is interpreted as v, the integer value to be inserted. If the value of the cmd selector is "check", then the message is expected to have two more components: v and caller.

```
slist(n, next) ::
*[ case (cmd) of
   "insert" : (v)
              if v < n
                then next := list(n, next)
                     n := v
                else if v > n
                       then if next = "nil"
                              then next := slist(v,"nil")
                       else send ("insert",v) to next
                            fi
                     fi
              fi
   "check" : (v, caller)
              if v = n
                then send (true ) to caller
                else if v < n
                       then send (false ) to caller
                       else if next = "nil"
                              then send (false) to caller
                              else send ("check",v,caller) to next
                            fi
                     fi
              fi
]
```

Figure 2.2: List Object for Sorted List

2.3 Three Useful Library Objects

Although the internal behavior for an object is described by the object definition that was used to create the object, the external behavior of an object can be completely described by its input and output behavior. The input to an object is always a message and the output is the response of the object to the message, i.e., the messages sent and objects created. Defining the behavior of an object by its input and output behavior is an abstraction over the internal details of the object's implementation. With this abstraction, objects can be created and manipulated without knowing their internal workings. The object definition can thus be expressed in Cantor, or in a representation that is preferred for the implementation.

This flexibility allows the semantics of Cantor to be obviated within an object definition, but still requires that the semantics be observed at the level of interaction between objects. The motivation for the alternate representations, called *custom* objects, is special efficiency for particular runtime environments. For example, efficient concurrent data structures, such as the balanced cube [11] which exploit interconnection topology information of ensemble machine implementations, can be accessed from Cantor as custom objects. Thus, objects expressed in Cantor and custom objects can be freely interposed in the same computation.

Object definitions that can be cleanly and succinctly defined by their input and output behavior are excellent candidates for library objects. For example, the class of objects that always produce the same output for the same input message is functional or *history-insensitive* and they can be used freely as library objects. The application of history-insensitive objects to object libraries is straightforward. For history-sensitive objects, succinctly defining the external behavior is complicated by the object's capability to change its current behavior based upon the messages the object has received. This section documents three object definitions that result in history-sensitive objects. The changes to the internal state for these objects are, however, well-defined.

2.3.1 Stacks

The object definition shown in Figure 2.3 behaves as an element for a stack. Two assumptions about the stack object are that each object is created containing a value and, secondly, that there will never be more pops than pushes. The stack object has two "modes" of operation. The "full" mode occurs when the stack object contains a value for the stack. For each "push" message received in the full mode, a new stack object is created containing the current stack value and next link. The current stack value then becomes the argument of the "push" message.

When the first "pop" message is received, the stack object sends the value of content to caller and then enters the "empty" mode of operation. All "pop" messages received by a stack object that is in the empty mode are forwarded to next. If next contains the value "nil", then the symbol "empty" is replied to caller. The symbol "empty" will be sent only if more "pop" messages are sent to the stack than "push" messages.

When a "push" message is received by a stack object that is in the empty mode, the content of the message is assigned to content and the stack object returns to the full mode.

```
stack (content, next) ::
*[ case (cmd) of
   "push" : (val)
            if next = "nil"
               then next := stack(content, "nil")
               else send ("push",content) to next
            fi
            content := val
   "pop" : (caller)
           send (content) to caller
           *[ case (cmd) of
             "push" : (val)
                      content := val
                      exit
             "pop"  : (caller)
                      if next = "nil"
                         then send ("empty") to caller
                         else send ("pop", caller) to next
                      fi
            ]
  ]
```

Figure 2.3: Stack Object

This formulation for a stack has two attractive properties. Multiple messages can be sent to the stack without having to synchronize with reply messages. For example, a series of "push" messages followed by a series of "pop" messages will behave correctly, independent of message delays.

A second property is that an interleaved series of "pop" and "push" messages will only involve the "top of stack" object. The first "pop" message will put the top of stack object into the empty mode. The following "push" message will place the top of stack object back into the full mode. For this formulation, the number of objects involved in a push or pop operation is limited to the difference between the number of "push" and "pop" messages processed by the stack.

One possible refinement to the stack object is to have the objects self-destruct if their value is popped and the value of next is "nil". The difficulty with this refinement is that the top of stack object has to be treated differently; otherwise, the entire stack will self-destruct.

2.3.2 Queues

Figure 2.4 depicts one formulation for expressing a queue in Cantor. The two operations performed on the queue object are advance queue front which returns the value at the head of the queue, and insert a new value at the tail of the queue. The stack object requires only a single reference to the "top of stack" object, but queues require two references, one for the head of the queue and a second for the tail of the queue. To properly maintain the two references, the queue is partitioned into two different kinds of objects: queue_cell objects and queue_master objects. The queue_cell objects are used to form a linked list of the values stored in the queue. The queue_master object is used to control access to the linked list of queue_cell objects. The queue of Figure 2.4 permits concurrent writes into the queue, but only a single reader.

An enqueue operation is accomplished by sending the queue_master an "insert" message with a single argument of the value to be inserted. No reply message is sent in response to the "insert" message. A dequeue operation is accomplished by sending the queue_master object an "advance" message. The argument to the "advance" message is a reference value designating the object that is to receive the value stored at the front of the queue.

Multiple "insert" messages may be concurrently sent, but before an "advance" message is sent, the reply value for the previous "advance" must have been sent, excluding of course the very first "advance" message. The set of objects that communicate with the queue_master object must conform to this domain constraint.

The behavior of the queue_master object is divided into three modes: an empty queue, a queue with one value, and a queue with multiple values. The outer <u>case</u> construct reflects the mode where the queue is completely empty. If an "advance" message arrives first, then by the domain constraint, the next message must be an "insert" message. For this case, the queue_master object waits for the "insert" message and then replies the value of the "insert" message to the value of caller of the previous "advance" message

Once the number of "insert" messages received exceeds the number of "advance" messages received, a queue_cell object is allocated and assigned to the local variables hd and tl. The second <u>case</u> construct is then used to process subsequent messages. The receipt of an "insert" message results in a new queue_cell object. This new object is to be added at the tail of the current queue. A "link" message containing the value of the new tail object is sent to the old tail object, and the variable tl is assigned the reference value of the new queue_cell object.

When a "advance" message is received, care has to be taken to check if the queue contains only a single value or not. To test if the queue contains only a single value, hd is compared with tl. If the two variables have equal values, then the queue has reference to only a single queue_cell object. The action for this case is to send the queue_cell object an "fget" message. The queue_master object then returns to the empty mode, i.e., the outermost <u>case</u> construct, because the "fget" message will cause the queue_cell object to reply directly to the object that sent the "advance" message.

If the queue contains more than one value, the front of the queue must be retrieved and the front of the queue advanced. Fetching the value at the front of the queue and the value for the new queue front is accomplished by sending a "get" message to hd.

From this point, two types of messages can be received: additional "insert" messages and the reply from hd. A third case construct is used to handle the two types of possible messages. The case for "insert" performs as before. The case for "hd reply" sends the value at the queue front to the object that sent the "advance" message and assigns the new queue front value to hd. The receipt of the "hd reply" messages returns control to the second case construct. If the value of the head of the queue is "nil", then the queue is empty, and control returns to the outer case construct.

```
queue_cell(value, next) ::
*[ case (cmd) of
 "link" : (new_next) next := new_next
 "get"  : (caller) send ("hd reply",next,value) to caller
                   exit
 "fget" : (caller) send (value) to caller
 ]

queue_master() ::
*[ case (cmd) of
 "advance" : (caller) % wait for "insert" message
             [ (discard, v) send (v) to caller ]
 "insert"  : (v)
             let hd = queue_cell(v,"nil")
             let tl = hd
             *[ case (cmd) of
              "insert"  : (v)
                          let nt = queue_cell(v,"nil") % new tail
                          send ("link",nt) to tl
                          tl := nt
              "advance" : (caller)
                          if hd = tl then send ("fget",caller) to hd
                                          exit
                          fi
                          send ("get",self) to hd
                          [ case (cmd) of
                          "insert"   : (v)
                                       let nt = queue_cell(v,"nil")
                                       send ("link",nt) to tl
                                       tl := nt
                                       repeat
                          "hd reply" : (new_head,v)
                                       send (v) to caller
                                       hd := new_head
                          ]
                          if hd = "nil" then exit fi
             ]
 ]
```

Figure 2.4: Queue Object

2.3.3 Vectors

To manipulate vector data in Cantor, the data must be placed into objects. The objects are then organized into some form of linked data structure that is capable of mimicking the indexing operations. The cost of this mimicking is that the indexing operation of a true vector is a unit time operation, but in Cantor must be performed as a series of message-passing operations. Thus, the vector object is a prime candidate to be made into a custom object for implementations where large amounts of contiguous storage are available. To offset this advantage, the Cantor representations are concurrently accessible and because of their organization, amenable to sparseness, i.e., missing elements.

The least complicated formulation is to organize the vector as a list of objects. The object definition of Figure 2.5 implements a sparse representation for a vector. The vector object processes "put" and "get" messages using the case construct. The "put" message contains two arguments, the vector index i and the value to be stored at index i. If the value of i is equal to the value of index for the object, the value of the "put" message is stored inside the object. If i is greater than index, the "put" message is forwarded to the next vector object in the list. If the current object is the last object in the list, a new vector object is created whose index and value are the arguments of the "put" message. If the value i is less than the current index, the current contents of the object are copied into a new object which is linked in as next and the contents of the "put" message become the contents of the object.

The processing of "get" messages is similar to the processing of "put" messages. If the value of i equals the index of the object, the value stored as the content of the object is sent to caller. If i is greater than the local index and a next object exists, the "get" message is forwarded the next link. If there is no next object or i is less than the current index, a value of zero is replied. Because this object definition implements a sparse representation, the default value for elements of the vector that are not instantiated as objects is integer zero.

For a vector of length N, if the indices sent to the vector are random, the average number of messages sent per indexing operation is $\frac{N}{2}$. This number is excessive unless the vector can be used in a pipelined fashion, which effectively defeats the purpose of the random access mode. A second formulation for the sparse vector object is shown in Figure 2.6. The elements of the vector are organized as a binary tree instead of a linear list. The maximum number of messages to access an element of the vector is $\log_2 N$. This worst case number is a tremendous improvement over the average case number for the linear list.

The initialization of the tree vector object is different from the linear list vector because the tree needs to be kept balanced. The initial message sent to the vector object contains the minimum and maximum index for the the vector. The object calculates the mid-point between the minimum and maximum and then enters the case construct which discriminates between the incoming "put" and "get" messages, as was done for the linear list version.

The formats of the "put" and "get" messages are identical to the formats for the linear list version. When the object receives a "put" message, the index of the "put"

```
vector (index, content, next) ::
*[ case (cmd) of
   "put" : (i, val)
           if index = i
              then content := val
              else if i > index
                      then if next <> "nil"
                              then send ("put",i,val) to next
                              else next := vector(i,val,"nil")
                           fi
                      else next := vector(index,content,"nil")
                           index := i
                           content := val
                   fi
            fi
    "get" : (i, caller)
            if index = i
               then send (content) to caller
               else if i > index
                       then if next <> "nil"
                               then send ("get",i,caller) to next
                               else send (0) to caller
                            fi
                       else send (0) to caller
                    fi
            fi
 ]
```

Figure 2.5: Linear Sparse Vector Object

message is compared to the index of the object. If the two indices are equal, the value of the message is stored as the local value. If the object index is greater than the message index, the message is forwarded to the left sub-tree; otherwise, the message is forwarded to the right sub-tree. Before the "put" message is forwarded, the appropriate sub-tree is checked to make sure it exists. If not, a new vector object is allocated and sent an initialization message followed by the forwarded message.

The response of the vector object to a "get" message is similar to its response to "put" messages. If the message index matches the object index, the local value is replied to caller; otherwise the "put" message is forwarded to one of the two sub-trees. If the appropriate sub-tree does not exist, the object corresponding to the index value of the "get" message is not a member of the tree. Because the vector is sparse, all vector elements not represented by objects are assumed to be of value zero. Integer zero is therefore sent to caller.

```
vector (content) ::
[ (low,high)
  let index  = low + (high-low+1) / 2
  let left = "nil"
  let right = "nil"
  *[ case (cmd) of
     "put" : (i, val)
             if i < index
                then if left = "nil"
                        then left := vector(content)
                             send (low,index-1) to left
                     fi
                     send ("put",i, val) to left
                else if i > index
                        then if right = "nil"
                                then right := vector(content)
                                     send (index+1,high) to right
                             fi
                             send ("put",i, val) to right
                        else content := val
                     fi
             fi
     "get" : (i, caller)
             if i < index
                then if left <> "nil"
                        then send ("get",i,caller) to left
                        else send (0) to caller
                     fi
                else if i > index
                        then if right <> "nil"
                                then send ("get",i,caller) to right
                                else send (0) to caller
                             fi
                        else send (content) to caller
                     fi
             fi
   ]
]
```

Figure 2.6: Tree Sparse Vector Object

2.4 Some Interesting Programs

To explore the cognitive process of turning English descriptions of programs into Cantor programs, six complete programs are presented in this section. The programs were chosen not for their cleverness nor execution efficiency *per se*, but rather for their clarity in demonstrating useful paradigms for writing Cantor programs.

2.4.1 Perfect Numbers

Perfect numbers are defined as positive integers whose divisors when summed equal the number. For example, the numbers 6 and 28 are perfect:

divisors of 6 are	1,2,3	and	$1+2+3=6$
divisors of 28 are	1,2,4,7,14	and	$1+2+4+7+14=28$

To test a number for perfectness the divisors are computed and then summed. For a test number N, the simplest procedure is to enumerate the integers between 1 and $\sqrt{N}$ and keep a running total of the divisors plus their quotients whose remainders are zero. For the division and summing to be performed concurrently, the test divisors must be enumerated concurrently. A simple strategy is to build a binary tree in which for test divisor k, the sub-trees have test divisors of $2k$ and $2k+1$. The program of Figure 2.7 contains an object definition called divisor which uses this algorithm. The object definition is used in the program to find all perfect number less than or equal to 10,000. The persistent variables of divisor are the test number N and the object that is to receive the result, caller. The test divisor is received as the message variable k. If the test divisor is greater than $\sqrt{N}$, then zero is replied to caller. If the test divisor is less than or equal to $\sqrt{N}$ and the remainder of the dividend N and the test divisor is zero, then the quotient and test divisor are summed and stored in a new add1 object. The value of caller is also stored in the add1 object, and the reference value of the add1 object becomes the value of caller.

Providing that the test divisor is not greater than $\sqrt{N}$, the next step is to concurrently perform the testing of $2k$ and $2k+1$. Two new divisor objects are created and sent messages containing $2k$ and $2k+1$. An adder object is also created to synchronize the two replies from the two new divisor objects. After the adder object synchronizes with the two replies, the sum of the two replies is sent to caller.

The object definition iter performs the iteration to test all integers between 1 and 10,000 inclusively. Each iter object initially waits for a test number and then initiates a test for perfectness on the received number by sending the number to a new divisor object. The iter object then creates a new iter object and sends the new object the current test number plus one, provided the test number is not greater than 10,000. After the iter object initiates the checking of the next test number, it waits for a reply to the message it sent to the divisor object. The reply contains the sum of the divisors. If the reply equals the test number, then the test number tagged with the symbol "perfect" is sent to caller which always is set to the value of console obtained from the main object.

Concurrency for this program is twofold. The search for divisors for a *single* test number is applied concurrently, and the summation of the test divisors for a test number is also performed concurrently. The second source of concurrency is in the iter object

```
add1 (caller, ans1) :: [(ans2) send (ans1+ans2) to caller ]

adder(caller) :: [ (ans1) [ (ans2) send (ans1+ans2) to caller ] ]

divisor (N, caller) ::
[ (k)
  if k*k > N
     then send (0) to caller
     else if (N mod k) = 0 then caller := add1(caller, k + N/k) fi
          let m = adder(caller)
          send (2*k)   to divisor(N, m)
          send (2*k+1) to divisor(N, m)
  fi
]

iter(MAX,output) ::
[ (i) send (1) to divisor(i, add1(self,-i))
      if i < MAX then send (i+1) to iter(MAX,output) fi
      [ (reply) if i = reply then send ("perfect",reply) to output fi ]

]

[ (console) send (1) to iter(10000,console) ]
```

Figure 2.7: Perfect Numbers Program

where the testing of numbers is decoupled from the generation of divisors. Each iter object initiates the test for perfectness on its test number and also initiates the test for perfectness on the next test number.

2.4.2 Prime Sieve

The second form of concurrency in the program for finding perfect numbers was effectively to unroll the "iteration" by instantiating the objects sequentially, but not requiring the first object to terminate before the second object becomes active. The generalization to this paradigm is to allow the objects to persist indefinitely so that objects can be reused in a pipelined fashion. A simple demonstration of this technique is the construction of a prime sieve to generate all the prime numbers up to a predefined limit. The task of the sieve is to filter out all non-prime numbers. The technique is devoid of tantalizing number-theoretic tricks but uses Cantor objects to form a simple but elegant pipeline. The program is partitioned into two parts: the sieve and the number generator that feeds the sieve. The requirements of the number generator are that it must emit a stream of integers such that the stream contains all the prime numbers and that the integers are emitted in strictly increasing order. The simplest such stream would be the natural numbers starting with 2.

The sieve can be described by a linear chain of objects with each object containing a single prime number. Each sieve object receives potential primes numbers to test for divisibility. If the test number divides evenly, it is discarded; otherwise the test number is relatively prime and is sent to the next sieve element. Whenever a test number reaches the end of the sieve, the number must be a prime and is made into a new sieve object at the end of the chain.

The number generator could be a simple counter and emit the natural numbers starting with 2. Naturally this would be wasteful, for all even numbers except for 2 are not prime. A first improvement would be to omit all multiples of two, i.e., emit only the odd numbers. Further improvements would be to omit multiples of 2,3,5,7,11, etc.. A number generator of this type is called a "wheel" [19]. The wheel is composed of an "addendum" and "spokes". The addendum is the product of the prime numbers from 2 up to some finite limit. The spokes are the integers relatively prime to the addendum. The wheel operates by maintaining an accumulator that is a multiple of the addendum. The accumulator is added to each spoke and then sent to the sieve for primality testing. After each spoke has sent a number to the sieve, the accumulator is advanced to the next multiple of the addendum.

For example, consider the addendum of: $1 \cdot 2 \cdot 3 = 6$. This addendum would result in a two spoke wheel of 1 and 5, and would generate the sequence:

$$0+1, 0+5, 6+1, 6+5, 12+1, 12+5, 18+1, 18+5, ... = 1, 5, 7, 11, 13, 17, 19, 23...$$

Every integer in this sequence is relatively prime to 2 and 3, and the first non-prime generated is 25. The number of spokes for a wheel is determined by calculating Euler's Function for the addendum.

The program of Figure 2.9 uses the combination of a predetermined object structure and a computed object structure. The wheel is the predetermined object structure, consisting of 8 spokes initialized with the values of 1,7,11,13,17,19,23, and 29. The addendum for the wheel has the value: $1 \cdot 2 \cdot 3 \cdot 5 = 30$. The computed object structure is the sieve. A snapshot of the object graph during the execution of this program is shown in Figure 2.8. In this graph, circles represent objects and the edges represent the references between objects. The program of Figure 2.9 requires only three types of objects.

The sieve objects make up the linear chain of objects that comprise the sieve. The wheel is comprised of 8 spoke objects and a single idler object. The 8 spoke objects and the idler object are linked together inside of the main object as a ring. Initially, a single message is inserted into the ring at spoke3. Thereafter, a message will circulate through the 9 objects that comprise the wheel. For each complete revolution, each spoke object will insert a test number into the sieve. The idler object will then advance the current value of the accumulator by the value of the addendum. This cycling action will continue until the accumulator reaches 10,000, at which point the wheel is shut down, followed by the sieve emptying out.

The sieve object performs as described previously. Each sieve object independently sends prime number messages to console, so it is quite possible that prime numbers will arrive at the external object in non-ascending order.

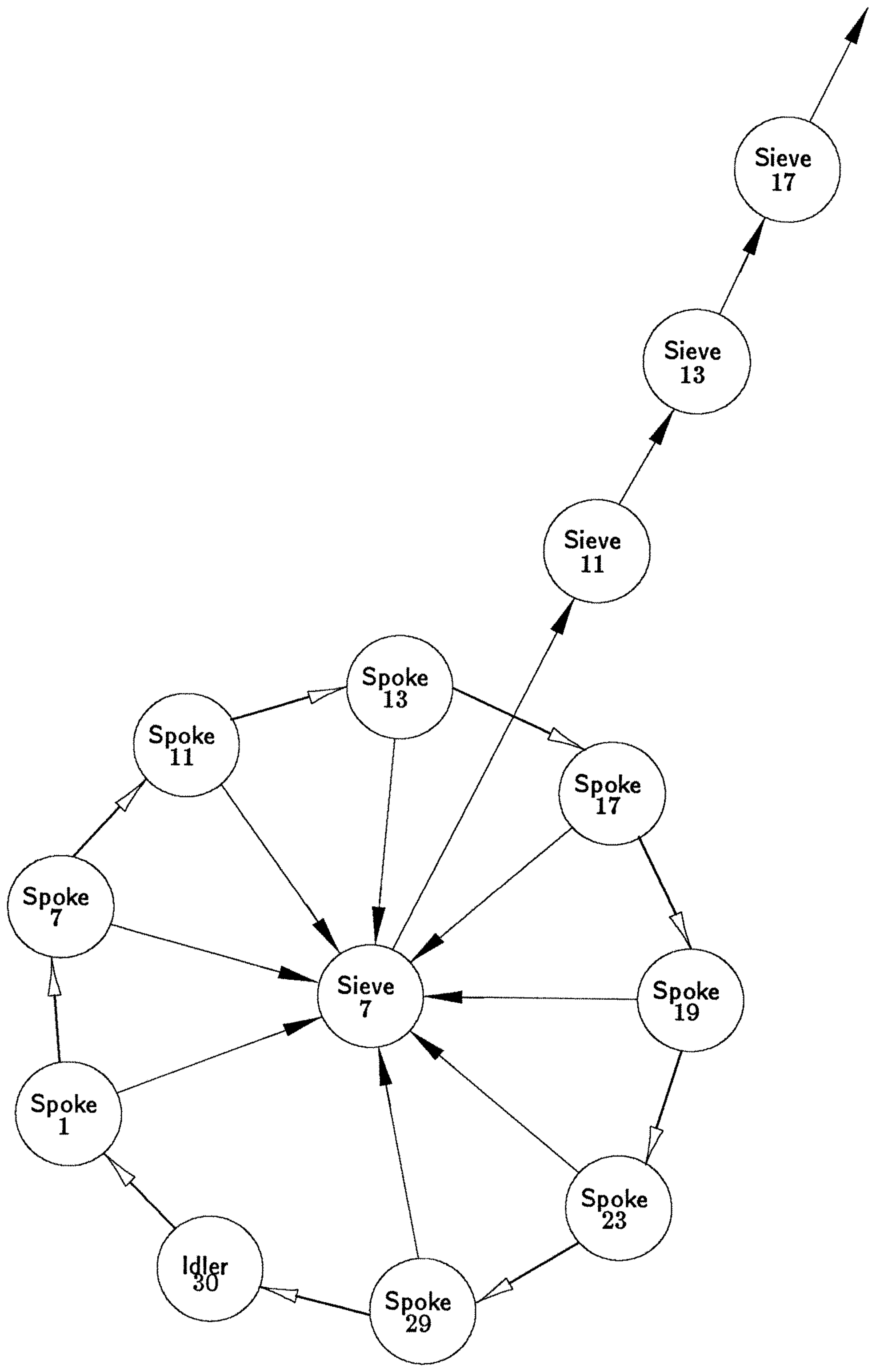


Figure 2.8: Object Graph for Prime Sieve

```
spoke (v,hub,next) ::
*[ (k) send (v+k) to hub
       send (k) to next ]

sieve (v, next, console) ::
*[ (p) if ((p mod v) <> 0)
         then if next = "nil"
                 then send (p) to console
                      next := sieve(p, "nil", console)
                 else send (p) to next
              fi
      fi
]

idler (delta, next) ::
*[ (k) if k < 10000 then send (delta+k) to next fi ]

[ (console)
  let hub = sieve (7,"nil",console)

  let spoke1 = spoke( 1,hub,spoke2)   let spoke2 = spoke( 7,hub,spoke3)
  let spoke3 = spoke(11,hub,spoke4)   let spoke4 = spoke(13,hub,spoke5)
  let spoke5 = spoke(17,hub,spoke6)   let spoke6 = spoke(19,hub,spoke7)
  let spoke7 = spoke(23,hub,spoke8)   let spoke8 = spoke(29,hub,idler1)
  let idler1 = idler(30,spoke1)

  send (0) to spoke3
]
```

Figure 2.9: Wheel Driven Prime Sieve

A major problem with the program of Figure 2.9 is that the production of messages flowing into the sieve is not regulated to the consumption rate of the sieve. Therefore, a message bottleneck may form between the spokes of the wheel and hub. A solution to the bottleneck between the sieve and the wheel that does not change the basic character of the algorithm is to alter the message production so that the "supply" of test numbers never exceeds the "demand" for test numbers. Stated slightly differently, prime numbers will be "pulled" out from the end of the sieve instead of having test numbers "pushed" into the first stage of the sieve.

A program that uses the new strategy is shown in Figure 2.10. Each sieve object employs the case construct to distinguish between two types of incoming messages, "request" and "answer". The sieve objects are created containing a prime number p and are initially waiting for a number N to test for primality. For each "answer" message received, the sieve object will immediately request another test number and then test the number received for primality. If the number is relatively prime, the number is sent to next which is initially set to the value of the external object console. A new sieve object is then created and assigned to the next acquaintance variable.

After the next acquaintance variable contains a reference value to a new sieve object, the original sieve object can be in one of two modes; either the object contains a test number that is relatively prime and is waiting for a "request" message from the next sieve element, or it has received a "request" message from the next sieve element and is waiting for a test number from the previous stage of the sieve. Notice that the sieve object sends an initial "request" message to self to compensate for the missing "request" message from the new sieve object.

The overall strategy for the sieve objects is to internally "buffer" one number that is relatively prime to the number stored inside the sieve object. When a "request" message is received, the sieve object will repeatedly send "request" messages to the previous sieve element until a relatively prime number is received. The number is then sent onto the next stage of the sieve. If the "answer" message is received first and the number N received is relatively prime, the object waits for the "request" message and then sends the saved value of N to the next stage of the sieve.

The first sieve object is connected to a new object called axle which serves as the interface between the wheel and the sieve. Each "request" message received by the axle object will cause the wheel to advance one notch. Thus the speed at which the wheel "rotates" is directly controlled by how fast the first sieve object can accept test numbers, and indirectly by the rate at which prime numbers are stored into new sieve objects at the end of the sieve.

```
sieve (next, back, p) ::
*[ (cmd, N)
   send ("request") to back
   if N mod p = 0
      then send ("request") to back
      else send (N) to next
           next := sieve(next, self, N)
           send ("request") to self
           *[ case (cmd) of
              "request" : () *[ (cmd, N)
                                send ("request") to back
                                if N mod p <> 0
                                   then send ("answer", N) to next
                                   exit
                                fi ]
                "answer" : (N)
                           if N mod p = 0
                              then send ("request") to back
                              else let N_save = N
                                   [ (cmd) send ("answer", N_save) to next ]
                                   send ("request") to back
                           fi ]
   fi ]

axle (next_spoke, start, acc) ::
*[ (discard)
   send (acc) to next_spoke
   [ (v, k, ns) send ("answer", v + k) to start
                acc := k
                next_spoke := ns ] ]

spoke (v, hub, next) :: *[ (k) send (v, k, next) to hub ]

idler (delta, next) ::
*[ (k) if (k < 10000) then send (delta+k) to next fi ]

[ (console)
  let hub    = axle(spoke3, sieve(console, hub, 7), 0)
  let spoke1 = spoke( 1,hub,spoke2)   let spoke2 = spoke( 7,hub,spoke3)
  let spoke3 = spoke(11,hub,spoke4)   let spoke4 = spoke(13,hub,spoke5)
  let spoke5 = spoke(17,hub,spoke6)   let spoke6 = spoke(19,hub,spoke7)
  let spoke7 = spoke(23,hub,spoke8)   let spoke8 = spoke(29,hub,idler1)
  let idler1 = idler(30,spoke1)
  send ("request") to hub ]
```

Figure 2.10: Demand Driven Prime Sieve

2.4.3 Merge Sort

The merge sort algorithm for sorting values by pairwise comparison is both an elegant recursive algorithm and also an inherently concurrent algorithm. The unconditioned input data is represented as a linked list of values. The list is recursively halved until the resulting lists consists of one or two values. For lists of two values, the values are compared and then interchanged if necessary. The next step is to merge the results by combining the lists of one or two elements into progressively larger lists until the original list has been completely reassembled.

The program of Figures 2.11 and 2.12 builds a list of 1,000 integers in descending order and then sorts the integers *in situ* into ascending order. The object definition make_list builds the initial list and sends a reference to the first object of the list to main. The merge sort algorithm starts by sending the mergesort object a message containing a reference to the first element of the list and the length of the list. The mergesort object is merely a front end for the msort object. The length of the list is examined by msort. If the length is 1, then the list is sent a "get3" message which returns three values: a reference to the first element of the list, the first value stored in the list, and a reference to the second element of the list. If the list length is greater than 1, the list is halved by sending the list a "split" message. The msort object waits for the reply from the "split" message. The reply contains the reference to the second half of the list. The list is now split, and the concurrent recursive step is applied by creating two new msort objects and sending the two new objects onehalf of the original list. The return from the recursion is handled by creating a join object which merges the two sorted list halves. The join object then replies the merged list to the caller of msort.

The join object expects messages of the "get3" form, that is, references to the first and second elements of the list and the value stored in the first list element. After the join object receives two such messages, the two separate lists are merged. The value of the first element of the first list is compared to the value of the first element of the second list. If the first is less than the second, then the second list is merged into the first list; otherwise, the second list is merged into the first list. The merge step is performed by sending the list object a "merge" message. The reply from the "message" indicates that the merge operation has completed. The join object then sends the merged list to caller, which is the caller of msort.

Most of the interesting behavior for the merge sort program occurs in the list object. The "split", "get", "get3", and "print" cases are straightforward. The "merge" case, however, is subtle because it uses one level of look ahead. The "merge" message does not merge the local value, but rather merges the value of the next list element. The argument of the "merge" message is of the "get3" form with the addition of caller which is a reference value. The list contained in the message is to be merged into the tail of the current list. If the tail is "nil", then the list received becomes the new tail, and caller is acknowledged that the merge step is finished. If the tail of the current list is a reference to a list object, then a "get" message is sent to tl. The reply from the "get2" message contains the value stored in the next object and the value of tl for the next object. The list object now holds the value stored in the first element of the other list and the value stored in the tail of the current list. These two values are compared. If the value of the other list is smaller, then the other list is sent a "merge" message containing the tail of the current list and the other list becomes the tail of the current list. If the value of the other list is equal to or greater than the value stored in the tail of the current list, then no change is made to tl and the "merge" message is forwarded to tl.

Assuming that all of the concurrency can be exploited in this program, the number of steps required to sort a list of length n is proportional to n. The halving of the list requires $\frac{n}{2}$ for the first split, $\frac{n}{4}$ for the second split; thus, the splitting time is proportional to:

$$\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ldots < n$$

The merge step begins with lists of length 1 and progressively combines the lists until a list of length n is reached. The growth pattern is the complement of the halving step, however; to split a list of length n can be done in $\frac{n}{2}$ steps, but to combine two lists of length $\frac{n}{2}$ into a list of length n requires at most n steps. Therefore the time complexity is still linear in n, but the constant factor will be larger. The time to completion for this concurrent program is $O(n)$ in comparison to $O(n \log n)$ for sequential merge sort [3].

The space requirements for this program are quite small. The linked list requires n list objects, and the number of join objects used to merge the single list elements into large lists will be $n - 1$. The number of join objects can be deduced by viewing the list objects as the leaf nodes of a binary tree and the join objects as the branch nodes of the tree. For a complete binary tree with n leaves, there are $n - 1$ branch nodes.

```
mergesort(caller) ::
[ (l,n) send (l,n) to msort(self) [ (l1, a, x) send (l1) to caller ] ]

msort (caller) ::
[ (l,n)
  if n = 1 then send ("get3",l,caller) to l
           else send ("split",self,n/2) to l
                [ (l2)
                  let j = join(caller)
                  send (l, n/2) to msort(j)
                  send (l2, n/2 + n mod 2) to msort(j)
                ]
  fi
]

join (caller) ::
[ (l1,a,x)
  [ (l2,b,y)
    if a < b then send ("merge",l2, b, y, self) to l1
                  [ () send ("get3", l1, caller) to l1 ]
             else send ("merge",l1,a,x,self) to l2
                  [ () send ("get3", l2, caller) to l2 ]
    fi
  ]
]

serializer (output) ::
*[ (v, caller) send (v) to output
               send () to caller ]

make_list(n, caller, l) ::
[ (i) if i = n then send (list(i, l)) to caller
               else send (i+1) to make_list(n, caller, list(i, l))
      fi ]

[ (console)
  let n = 1000
  send (1) to make_list(n,self,"nil")
  [ (l)  send (l,n) to mergesort(self)
         [ (sl)  send ("print", serializer(console)) to sl
                 send ("done") to console
         ]
  ]
]
```

Figure 2.11: Initialization and Recursion for Merge Sort

```
list (hd, tl) ::
*[ case (cmd) of
  "merge" : (l, a, x, caller)
            if tl = "nil"
               then tl := l
                    send () to caller
               else send ("get2", self) to tl
                    [ (b, y)
                      if a <= b then send ("merge",tl,b,y,caller) to l
                                     tl := l
                                else send  ("merge",l,a,x,caller) to tl
                      fi
                    ]
             fi
   "split" : (caller, n)
             if n = 1 then send (tl) to caller
                           tl := "nil"
                      else send ("split", caller, n-1) to tl
             fi
   "get2"   : (caller) send (hd, tl) to caller
   "get3"   : (l,caller) send (l,hd, tl) to caller
   "print"  : (output)
              send (hd,self) to output
              [ ()
                if tl <> "nil" then send ("print", output) to tl fi ]
              exit
]
```

Figure 2.12: List Object for Merge Sort

2.4.4 Eight Queens

The eight queens problem is a showcase problem often cited in the literature of computer programming. The task is to place eight queen pieces on an 8 by 8 chessboard so that no queen is in jeopardy of capture. The queen game piece captures any other piece that lies along the same row, column, or diagonal. There are 92 solutions to the eight queens problem [9]. Figure 2.13 shows one solution that was generated from the program of Figures 2.14 and 2.15.

The search for solutions to the eight queens problem is massively concurrent. From the capture rule, there will be only one queen per row and column. A concurrent search can therefore be organized either by row or by column. Assuming a column by column search, a queen is placed in a row of column 1. The rows of column 2 are then searched to find a safe position for the second queen. After a safe position has been found, the rows of column 3 are searched to find a safe position for the third queen. Each subsequent column search involves checking the previous row and column pairs to be sure that the new position is safe. For the case where no safe row can be found, the partial solution is discarded.

A sequential search would involve backtracking once an impossible configuration is exposed. The backtracking step systematically removes the current emplacements of queens and then continues with a new placement. For the concurrent search, backtracking is not necessary because all solutions are equally pursued. The only action taken in response to a detected impossible configuration is to discard the configuration.

The program of Figure 2.14 begins with the make_q object definition which create eight queen objects each with a separate qlist object. The qlist objects are initialized with the same column number, but each qlist object is assigned a different row number. The qlist objects are used to keep track of the partial chessboard configuration. Each queen object is sent a message containing the first row number to be tested for the second column.

The queen object receives test row numbers and then sends a "check" message containing its column number and the test row number to the first element of its qlist to see if the test position results in a capture or not. When the qlist object receives the "check" message, the object checks the test position for capture with the row and diagonal of the qlist object. If the coordinates of the qlist object captures the coordinates of the "check" message, then the boolean value false is sent to the requesting queen object. If the test coordinates are safe from capture, the "check" message is forwarded to the next qlist object unless the end of the list has been reached. When a "check" message reaches the end of the list of positions, the test position is safe for the partial list. The boolean value true is replied to the requesting queen object.

After sending a "check" message, the queen object waits for the reply from the qlist object. A reply of true will cause the partial list to be copied and augmented with the new pair of coordinates. The list is copied by sending a "copy" message to the first element of the list. The reply from the "copy" request is a reference to the first element of the new list. A new qlist object is created to hold the new coordinates and is linked onto the front of the new list. If the column number is less than 8 a new queen object is created to test the new qlist object against the rows of the next column. If the column number is 8, then a complete solution has been found. The reference to the first qlist object of the complete solution is sent to a queue_master object. The queue_master is a library object and is described in Section 2.3.2.

After processing the reply from the "check" message, the queen object sends itself a new row number to test providing the current test row number is less than 8.

The queue of Section 2.3.2 is used to assemble the lists of solutions from the breadth-first search. The printer object of Figure 2.15 is used as an exchanger between the qlist objects and the queue_master object. Initially the main object creates a printer object which contains reference to a queue_master object and to chessboard. The main object for this program expects two external reference values to be sent to the program, a reference to the console object as before, and a second special object called chessboard. This second external object is used to generate the chessboard display of Figure 2.13. The print queue is primed by sending the queue_master object an "advance" message. The first qlist solution that is sent to queue_master from a queen object will be immediately forwarded to the printer object.

The printer object waits for a message containing a reference to a qlist object. The printer object then sends each row and column pair to the chessboard object. After the solution has been serially sent to the chessboard object, the printer object sends an "advance" message to the queue_master object to request the next solution.

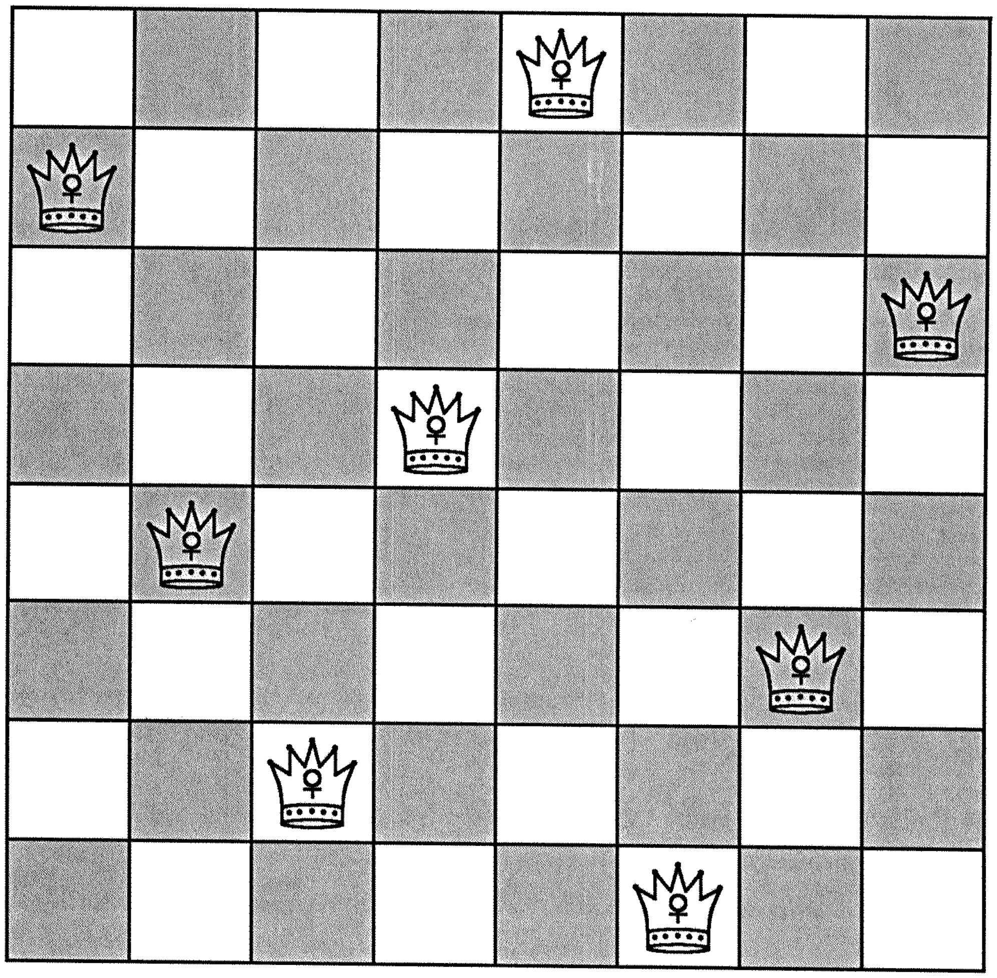


Figure 2.13: Solution to the Eight Queens Program

```
qlist (row,col,next) ::
*[ case (cmd) of
   "check" : (rn,cn,caller)
             if rn = row or (cn - col) = abs (rn -row)
                then send (false) to caller
                else if next = "nil"
                        then send (true) to caller
                        else send ("check",rn,cn,caller) to next
                     fi
             fi
   "copy" : (caller)
            if next = "nil"
               then send (qlist(row,col,"nil")) to caller
               else send ("copy",self) to next
                    [ (l) send (qlist(row,col,l)) to caller ]
            fi
   "get" : (caller) send (row,col,next) to caller
 ]

queen (ql,cn,out) ::
% ql --- list of valid queen positions
% cn --- column number
*[ (rn)
  send ("check",rn,cn,self) to ql
  [ (reply) if reply
              then send ("copy",self) to ql
                   [ (nql)
                     nql := qlist(rn,cn,nql)
                     if cn = 8
                        then send ("insert",nql) to out
                        else send (1) to queen(nql, cn+1, out)
                     fi ]
            fi
   ]
   if rn < 8 then send (rn+1) to self else exit fi
]

make_q(out) ::
*[ (i) send (1) to queen(qlist(i,1,"nil"),2,out)
       if i < 8 then send (i+1) to self else exit fi ]
```

Figure 2.14: Concurrent Eight Queens

```
printer (output, print_queue) ::
*[ (l) send ("get", self) to l
       *[ (r,c,l)
          send (c,r) to output
          if l <> "nil" then send ("get",self) to l
                        else exit
          fi
        ]
   send () to output
   send ("advance",self) to print_queue
 ]

[ (console, chessboard)
  let qm = queue_master()
  let ptr = printer(chessboard,qm)
  send ("advance",ptr) to qm
  send (1) to make_q(qm)
]
```

Figure 2.15: Printer and Main Objects for Queue Object

2.4.5 Gaussian Elimination

Numerical analysis problems present a special challenge when programmed in Cantor. Programs used in numerical analysis are often formulated in terms of matrices and described as sequences of actions performed over the indexed elements of the matrices. These representations provide an efficient compromise between the domain of mathematical objects used by numerical analysts and the representation of matrices as segments of contiguous storage in sequential computers. Cantor does not provide for contiguous arrays of storage elements as one of the basic types of values. Instead, matrices are built out of individual objects that behave collectively as a matrix.

An algorithm that uses a matrix representation extensively is Gaussian elimination. For concurrent applications the algorithm has been expressed in a myriad of ways including a functional version written in LISP [26] and also in a concurrent extension to the object-oriented language Simula [25]. The algorithm is used to solve sets of m linear equations with n unknowns by representing the equations as an m by n matrix. To solve for the unknowns, the matrix is placed into reduced row-echelon form and then back substitution is used to output the computed values for the unknowns. The program shown in Figures 2.17 through 2.20 performs the row reduction phase, with pivoting included to minimize round-off error. The procedure consists of the following steps, with j ranging from 1 to m:

1. Find pivot row for column j.
2. Move pivot row to the top of the matrix.
3. Adjust pivot row so that value in column j is 1.
4. Add multiples of the top row to the rows beneath it so column j is zero.
5. Cover top row, increment j and repeat.

For a sequential application of this procedure, the number of arithmetic operations is $O(m^2n)$. For concurrent applications, step 4 can be performed concurrently, reducing the time to completion by a constant factor of m. Thus, the total time to completion would be $O(mn)$.

Crucial to the formulation of Gaussian elimination in Cantor is the choice of the data structure to represent the matrix. Step 2 of the algorithm requires that two rows of the matrix be interchanged. This requirement suggests that the matrix may best be formulated in row major form, that is, each row of the matrix be represented as a linked list and accessed by a row header object. This representation is efficient for interchanging two rows because only the reference values for the first two elements of each row have to be swapped. The object graph of Figure 2.16 depicts each row as a linked list of list objects. The first element of each row is referenced by a list_head. The list_head objects in turn are organized as a linked list. To access an element of the matrix at random, the list_head objects must be traversed to reach the proper row. The list objects for the row are then traversed to reach the proper element. This organization is inefficient for randomly accessing elements of the matrix. The organization is, however, efficient for the access patterns required by the Gaussian elimination algorithm.

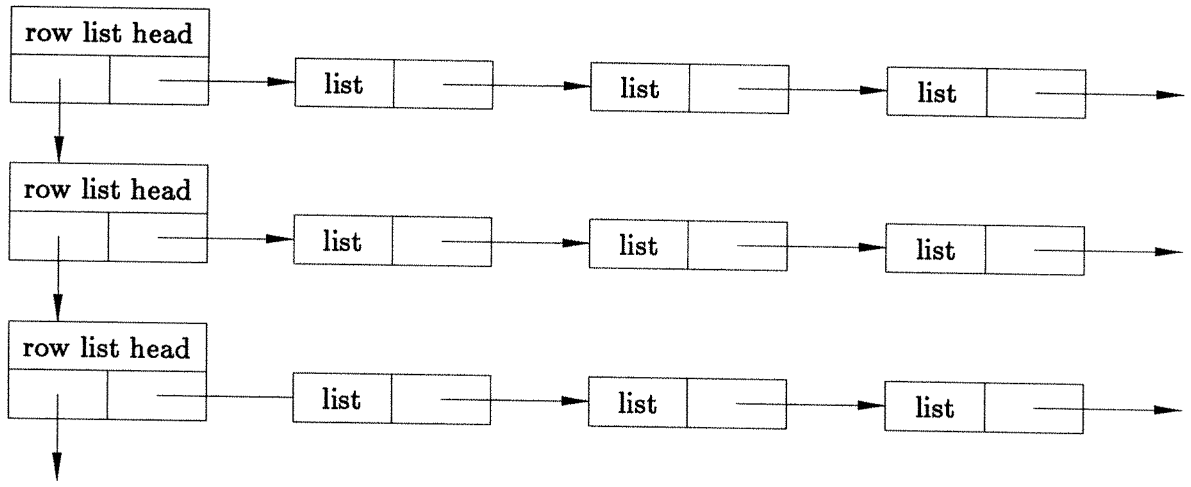


Figure 2.16: Matrix Data Structure for Gaussian Elimination

The first step is to build the matrix. The two objects definitions used to build the matrix are shown in Figure 2.20 along with the main object. Construction of the matrix begins with a message sent from main to a build_matrix object. The build_matrix object begins construction of the linked list of list objects for the list head and also initiates construction of the next row, providing the row count has not been reached. The row list is built by sending a message to a build_list object. The build_object object will generate a linked list of n list objects with each object initialized to $a_{ij} = 1.0/(i - j + 0.5)$. After the linked list is built, a "list" message containing a reference to the first element of the list is sent to the build_matrix object that requested the row to be built.

The build_matrix waits for completion messages from the row list and next list_head object that it initiated building. Because the replies can arrive in arbitrary order, a case construct is used to process the two replies. Once the two replies are received, the list_head object is created and a reference to the new object is replied to the object that created the build_matrix object. The arrival of the reply message at the main object indicates that the entire matrix has been constructed. The next step is to create a gauss object and send the reference to the first list_head object to the new object.

After the gauss object receives a reference to a list_head object, it checks the size of the matrix to see if the current matrix contains no equations or no unknowns. For either case, the original matrix is output by sending a "print" message to the first list_head object of the original matrix. If the matrix contains one or more equations or unknowns, then a compare object is created. A reference to the new object is sent to the the first list_head object of the current matrix as the argument of a "find piv" message. The "find piv" message will result in the current matrix being reduced by one row and one column, i.e., the top row and first column. The size of the matrix is reduced accordingly by decrementing m and n.

Most of the interesting behavior for the Gaussian elimination program occurs in the list and list_head object definitions. The list_head object initially waits for either a "print" or "find piv" message. The "print" message should only occur before or after the matrix has been reduced. The "find piv" message is used to select the pivot row of step 1. The response of the "list_head object to the "find piv" is to forward the message to the next list_head object in the linked list and also to get the value stored in the first element of the row list. This value is relayed to the compare object, along with a reference to the list_head object for the row.

The compare object accumulates the replies from the list_head objects and keeps track of the row with the maximum absolute value. After all of the replies have arrived, the row with the maximum value is interchanged with the top row, by sending the top row of the matrix a "swap" message with a reference to the list_head object containing the maximum value.

After the list_head objects send their row values to the compare object, they wait for a message from either the compare object or another list_head object. If the list_head object receives a "swap" message, then it sends a "set low" message to the list_head object containing its row list. If the list_head object receives a "set_low" message, then it is the pivot row. A reference value for its row list is sent to the top row list_head object, and the list object reference value contained in the message is assigned as the reference value for the local row list.

After the top row list_head object receives the reference value for the pivot row list, the reference value is stored as the local row list value, and the compare object is sent a message containing a reference to the row list and a reference to the next list_head object in the linked list.

When the compare object receives the reply from the top row list_head object, step 2 has been completed. Step 3 is performed by sending the pivot row list a "divide" message with an argument of the value stored in the first element of the pivot row list. The processing of the "divide" message by the list object is straightforward. When the list object completes the divide operation, an empty message is replied to the compare object.

When the compare object receives the empty message from the pivot row list, step 3 has been completed. step 4 is performed concurrently by sending the second list_head object in the linked list of list_head objects a "set mf" message containing a reference to the pivot row list. When the list_head object receives the "set mf" message, it will forward the message to the next list_head object in the linked list. The list_head object also sends a "mpy" message to the top row list, containing a reference to its own row list. The list_head object then advances the front of the row list by one to reduce the number of columns in the current matrix by one.

The list objects accomplish step 4 using a sequence of "mpy" and "sub" messages that are sent between the top row list and each of the lists beneath the top row. The sequence is started by the list_head object, which sends a "mpy" message to the top row list containing a reference to its own row list and the value stored in the first list object of its row list. The first list object of the top row then sends a "sub" message to the first list object of the lower row list containing the value stored in the top row and the next link for the top row object. When the lower row list receives the "sub" message, it multiplies the value stored in the top row element by the value stored in the first object of its row and then subtracts this value from the value stored locally. The operation is then repeated by sending a "mpy" message to the next link of the top row list. This cross-stitching pattern of message-passing continues until the end of the two lists is reached. An empty message is then sent to the compare object.

The compare object counts the number of completion messages received from the list objects. When the count equals the number of rows in the current matrix, Step 4 has been completed. The compare object then sends a reference value to the list_head object of the reduced matrix, i.e., the matrix that is one row and one column smaller than the current matrix, to the gauss matrix.

```
list (hd, tl) ::
*[ case (cmd) of
   "mpy"    : (mf, add_col, done)
              send ("sub", mf, hd, tl, done) to add_col

   "sub"    : (mf, val, ntc, done)
              hd := hd - mf * val
              if ntc = "nil" then send () to done
                             else send ("mpy", mf, tl, done) to ntc fi

  "divide" : (val, reply_to)
              hd := hd / val
              if tl = "nil" then send () to reply_to
                            else send ("divide",val, reply_to) to tl fi

  "get pr" : (caller) send (hd, tl) to caller

  "print"  : (output, caller)
              send (hd) to output
              if tl = "nil" then send () to caller
                            else send ("print",output, caller) to tl fi
]
```

Figure 2.17: Row List Element for Gaussian Elimination

```
list_head (ll,l,next) ::
*[ case (cmd) of
   "print" : (output)
             send ("print", output, self) to l
             [ () send () to output
                 if next <> "nil" then send ("print",output) to next fi
             ]

   "find piv" : (caller)
                if next <> "nil" then send ("find piv",caller) to next fi
                send ("get pr",self) to l
                [ (hd, tl) send (hd, self) to caller

                [ case (cmd) of
                  "swap" : (max_row, done)
                           if max_row <> self
                              then send ("set low",hd,l,self) to max_row
                                   [ (new_l) l := new_l ]
                           fi
                           send (next,l) to done

               "set low" : (new_l, reply_to) send (l) to reply_to
                                             hd := new_hd
                                             l := new_l
                                             repeat

               "set mf"  : (done, tr)
                           if next <> "nil"
                              then send ("set mf", done, tr) to next fi
                           send ("mpy", hd, l, done) to tr
                           l := tl
                  ]
                  ]
]
```

Figure 2.18: List Head Object for Gaussian Elimination

```
compare(count, matrix, gauss) ::
[ (max_val, max_row)
  let rc = count
  if  rc > 0
      then  *[ (test_val, test_row)
                 rc := rc - 1
                 if abs max_val < abs test_val then max_val  := test_val
                                                    max_row  := test_row
                 fi
                 if rc = 0 then exit fi
             ]
  fi

  % exchange top row with max row
  send ("swap", max_row, self) to matrix

  % divide max_row by value in first column
  [ (next, ml)
    send ("divide", max_val, self) to ml
    [ ()
      if count > 0
         then % do multiply and subtract steps concurrently
              send ("set mf", self, ml) to next
              % count replies from multiply and subtract steps
              *[ () count := count - 1
                    if count = 0 then exit fi  ]
      fi
    ]
    % repeat on smaller matrix
    send (next) to gauss
  ]
]

gauss (orig_matrix, m, n, output) ::
*[ (matrix)
   if m = 0 or n = 1 then send ("print",output) to orig_matrix
                     else let cmp = compare(m-1, matrix, self)
                          send ("find piv",cmp) to matrix
                          m := m - 1
                          n := n - 1
   fi ]
```

Figure 2.19: Compare and Gauss Objects for Gaussian Elimination

```
build_list(i, l, caller) ::
*[ (j)
   l := list(1.0 / (i-j+0.5), l)
   if j = 1 then send ("list", l) to caller
                 exit
            else send (j-1) to self
   fi
 ]

build_matrix (m, n) ::
[ (i,caller)
  send (n) to build_list(i, "nil", self)
  if i = m
     then [ (discard, l) send ("matrix",list_head(l,1,"nil")) to caller ]
     else send (i+1, self) to build_matrix(m, n)
          [ case (reply) of
            "list"   : (l)
                       [ (discard, next)
                         send ("matrix",list_head(l,1,next)) to caller ]
            "matrix" : (next)
                       [ (discard, l)
                         send ("matrix",list_head(l,1,next)) to caller ]
          ]
  fi
]

[ (console)
  let m = 64    % number of rows
  let n = m+1   % number of columns
  send (1,self) to build_matrix(m, n)
  [ (discard, matrix) send (matrix) to gauss(matrix, m, n, console) ]
]
```

Figure 2.20: Initialization and Output for Gaussian Elimination

2.5 Summary

In contrasting the six Cantor programs to sequential programs that perform the same computational task, the difference in expression is remarkable. Each object used in these programs is, by itself, quite simple. The intricacies of the Cantor formulations are found entirely in the interactions between objects.

The complexity is therefore controlled by organizing the programs so that the dependencies between objects are well understood. For the most general case in which every object can communicate with every other object, the possible set of pairwise interactions would increase quadratically in the number of objects. However, the semantics of Cantor enforce a principle of locality in which the reference to an object can be acquired only by message-passing or by object creation. This locality of reference is used to limit the possible set of messages that an object can receive. The new message received is not always assured to be the message that would be most convenient for the object to receive next. Thus the object must be prepared either to receive messages in an unknown order, e.g., the case construct, or rely upon sequencing dependencies in the communication.

Locality of reference is introduced by the programming paradigm, e.g., the divide and conquer strategy of the perfect numbers and eight queens programs, or is introduced by careful choice of data structure, e.g., the Gaussian elimination program. As is the case for structured programming, proper choice of control flow and data structures is paramount. For programming in Cantor, the plan for message flow and the organization of the objects is crucial.

Chapter 3

Computational Model and the Frameworks for Flow Analysis

The necessity of providing formal definitions for programming notations has perhaps been best summarized by Wirth and Weber [41]. Two objectives defined by Wirth and Weber are to provide users of a notation or "language" with a precise meaning for the language, and secondly, to ensure that the meaning of programs is preserved when the language is implemented on a computer. The strategy proposed by Wirth and Weber to accomplish these objectives is to redefine the "high-level" language into a second language that has simpler structure than the first. This procedure is continually reapplied until a final, primitive language is reached in which further redefinitions would only produce a new language whose structure would be no simpler than the primitive language.

The semantics of Cantor are defined using this approach. First, the model of computation is formally described in terms of functions whose domains and ranges are either primitive values or finite automata. Next, a set of four execution rules are stated to express how computations are interpreted in the formalized object model.

A primitive notation consisting of ten operations, called "instructions," is presented. For each instruction of the primitive notation, a corresponding action is defined over the formal model. The primitive notation captures the basic components of a Cantor program. For each syntactic construct in Cantor, an interpretation rule is defined in terms of the primitive instructions.

Cantor programs are translated into the primitive notation, which in turn has an interpretation in the formal model. The expression of the program in terms of the formal model satisfies the two objectives for a formal definition proposed by Wirth and Weber. The formal model also provides a framework, called the program flow framework, for reasoning about the execution of programs in the object model. Two additional frameworks, called future flow and after flow, are developed by replacing the formal model with a simpler model to capture particular aspects of a computation. Interpreting Cantor programs using the future flow or after flow frameworks provides information that approximates the information contained in the actual computation. This partial information is used to manage objects before and during program execution.

3.1 The Computational Model

The state of a computation expressed in the object model can be concisely determined by two pieces of information: the state of each object that is taking part in the computation, and the messages that are queued for each of the objects. Both objects and messages are organized by reference values. Each object has a reference value that uniquely identifies the object, and each message has a reference value that uniquely identifies the destination object for the message. The set R denotes the set of countably infinite reference values. For example, the set of reference values could be represented as the natural numbers. The requirement that R be infinite in size is only necessary for certain non-terminating computations, e.g., the prime sieve programs of Chapter 2.

The domain of primitive values includes R and also includes *literals*, in which the value of a literal is the literal. Examples of literals include integers, real numbers, and the logical values true and false. Formally, the set of primitive values for Cantor is written:

$$V = R \uplus \text{integers} \uplus \text{reals} \uplus \text{boolean} \uplus \text{symbols}$$

The set operator $\uplus$ denotes the disjoint union of the component sets.

The state of a computation is represented by two functions that describe the state of the queues and the objects respectively:

$$\begin{array}{lclcl} \mu & : & R & \mapsto & K^* \text{ where } K = V^* \\ \pi & : & R & \mapsto & O \end{array}$$

The function $\mu \in \mathrm{M}$ maps the reference values into zero or more k-tuples of V, written as K. Each k-tuple from V represents the contents of a message. Thus $\mu(r)$ for an object with reference value r represents the contents of the queue for object r. The set M is the set of all possible queueing functions μ. The function $\pi \in \Pi$ maps reference values into O, the set of all possible objects. The set Π is the set of all possible mappings between reference values and objects. Initially for a computation, every reference value is mapped to the ***undefined object*** definition. Messages cannot be sent to objects that are undefined objects.

The representation of an object is partitioned into two parts called the ***reactive*** part and the ***persistent*** part. The persistent part defines the state preserving capabilities for the object. The reactive part defines the transformations that the object can perform upon message objects. The reactive part is formally described by a finite state machine. The persistent part is defined by a set of storage locations that the state machine has exclusive access to and by the current state for the state machine. Formally, an object $o \in O$ is described by an 8-tuple: $o = \langle S, I, s_0, L, C, \delta, \lambda_{\mathrm{M}}, \lambda_{\Pi} \rangle$ where:

$$\begin{array}{lcl} S & = & \text{set of states (finite)} \\ I & = & \text{set of input states, } I \subseteq S \\ s_0 & = & \text{initial state, } s_0 \in I \\ L & = & \text{set of locations (finite)} \\ C & = & \text{set of functions } c\text{, where } c : L \mapsto V \\ \delta & : & S \times C \times \mathrm{M} \mapsto S \times C \\ \lambda_{\mathrm{M}} & : & S \times C \times \mathrm{M} \mapsto \mathrm{M} \\ \lambda_{\Pi} & : & S \times \Pi \mapsto \Pi \end{array}$$

Modelling the persistent part of an object is accomplished using a finite set of locations L and a storage function c that maps locations into their contents. The set C is the set of all possible mappings between locations and their contents for a single object. Initially, every object starts with the same c_0 function that maps all locations into the *undefined* value. This approach for representing storage is used in the modelling of random accessed stored program machines (RASP) [18]. In the RASP model, the set L is often treated as infinite, and thus the set C is also possibly infinite. For the objects of O, the set of locations is finite. The set V is, however, possibly infinite, because V contains R. Therefore, C is infinite if the computation requires an infinite number of objects and finite otherwise. Since objects are always created with the same c_0 function, there are only a finite number of descriptions for new objects.

The reactive part of an object is divided between internal changes to the object and the external influence of the object upon other objects. Both internal change and external influence can only occur in response to a message. Both internal and external effects occur by transition functions over S, the finite set of states of the object. The subset I of S defines the set of states in which the object is capable of accepting or *inputing* a new message. The initial state s_0 is always an input state.

The internal change of an object is controlled by the state transition function δ. The function δ defines the next state and next store mapping for an object from the current state, the current mapping of locations to values, and the current message the object is working on.

The external influence of an object consists of two functions, λ_M and λ_Π, to express the sending of new messages and creation of new objects, respectively. These two *output* functions share the same domain as the state transition function δ.

The functions c, λ_M, and λ_Π, as defined above allow behaviors that are clearly impossible from the standpoint of object programs studied in this thesis. For example, the function λ_Π cannot change the mapping of any object, only that of objects that have not yet been defined. To impose additional structure on the sets, queueing functions and rules for object creation are defined. To ensure that local optimizations on objects are not precluded, the c functions are left unrestricted.

The types of queueing actions allowed follow the construction for queues used in the Karp–Miller program schema [22]. Three actions of insert k-tuple (ins), access front of queue (acc), and advance queue front (adv) are defined as:

for object with reference value $r \in R$,
let $\mu(r) = k_1, k_2, \ldots, k_n$, the contents of the message queue for object r,

$$\begin{aligned} \text{ins}(\mu(r), k) &= k_1, k_2, \ldots, k_n, k \\ \text{adv}(\mu(r)) &= k_2, k_3, \ldots, k_n \\ \text{acc}(\mu(r)) &= k_1 \end{aligned}$$

From these three queueing operations, the types of transitions capable of objects can be more clearly expressed:

For object with reference value $r \in R$,

$$\begin{array}{lcll} \delta & : & S \times C \times \mathrm{acc}(\mu(r)) & \mapsto \quad S \times C \\ \lambda_{\mathrm{M}} & : & S \times C \times \mathrm{acc}(\mu(r)) & \mapsto \quad \begin{cases} \mathrm{ins}(\mu(r'), k) & \text{or} \\ \mathrm{adv}(\mu(r)) & \end{cases} \\ \lambda_{\Pi} & : & S \times \Pi & \mapsto \quad (r', o') \quad \text{where} \quad \pi(r') = o' \end{array}$$

The specification of acc($\mu(r)$) for K denotes that the message k-tuple is not any k-tuple, but the k-tuple current at the front of the queue for the object. The λ_{M} function denotes that the only types of changes allowed on μ are to insert a single message into a queue for an object, or to remove the message currently at the front of the queue for the object. The second definition for λ_{Π} captures the property that changes to π occur one at a time, but does not describe how the reference values are selected from R. Clearly one requirement is that r' be an undefined object before the transition.

3.2 Execution Rules

The state of a computation is completely specified by the configuration pair (μ,π) where $\mu \in$ M and $\pi \in \Pi$. To complete the formal definition of the computational model, execution rules are required to describe how a computation progresses from one configuration to another.

1. Object r is active (can undergo a state change) only when the input queue for r is not empty.
$$\forall i \in I : |\mu(r)| > 0 \Longleftrightarrow \delta(i, c, \mathrm{acc}(\mu(r)) = (s', c')$$

2. The queue front for an object r is only advanced when the object enters an input state.
$$\delta(s, c, \mathrm{acc}(\mu(r)) = (i, c) \Longleftrightarrow \lambda_{\mathrm{M}}(s, c, \mathrm{acc}(\mu(r))) = \mathrm{adv}(\mu(r))$$

3. Every cycle in the state transition graph must contain an input state.
$$\begin{array}{l} \forall s_i, s_j \in S, \forall c_i, c_j, c_k \in C, \\ \delta(s_i, c_i,\mathrm{acc}(\mu(r)) \rightarrow \ldots \rightarrow \delta(s_j, c_j,\mathrm{acc}(\mu(r)) = (s_i, c_k) \Longrightarrow s_i \in I \end{array}$$

4. New reference values can be introduced only by creating a new object.
   (a) $\delta(s, c,\mathrm{acc}(\mu(r)) = (s', c')$
   $$\text{where } \forall v_r \in c'(L) \text{ s.t. } v_r \in R \Longrightarrow \begin{cases} v_r \in & c(L) \text{ or,} \\ v_r \in & \mathrm{acc}(\mu(r)) \text{ or,} \\ v_r = & r' \text{ where } \lambda_{\Pi}(s, c,\mathrm{acc}(\mu(r)) = (r', o') \end{cases}$$

   (b) $\lambda_{\mathrm{M}}(s, c,\mathrm{acc}(\mu(r)) = \mathrm{ins}(\mu(r'), k)$
   $$\text{where } \forall v_r \in k \text{ s.t. } v_r \in R \Longrightarrow v_r \in \begin{cases} c(L) \text{ or,} \\ \mathrm{acc}(\mu(r)) \end{cases}$$

The first rule partitions the objects used in a computation into the sets of active and inactive objects. The set of active objects are those objects that have one or more messages queued for them. If the object is active and is currently in an input state, then the object can transit from the input state and access the message at the front of its queue as the current message.

The second rule guarantees that the message at the front of the queue for an object is always discarded when the object enters an input state. The third rule guarantees that an object will never create an infinite number of new objects or send an infinite number of messages in response to a single message. In other words, the response of an object to a message is finite. Infinite behavior can only be achieved at the configuration or system level, not at the object level.

The last rule clarifies the semantics of creating a new object by imposing a locality constraint on references. The transition functions δ and λ_{M}, the next state and message functions, can only *preserve* reference. The function λ_{Π}, however, can introduce a new reference into a computation by introducing a new object into the computation. Part (a) of rule 4 shows that from a state transition, references contained in the new map of locations to values can only come from one of three sources: the reference was contained in the old map of locations to values, the reference is contained in the message at the front of the queue, or the reference was created by the λ_{Π} function.

Whereas part (a) controls how reference is preserved inside an object, part (b) controls how reference is propagated between objects. Part (b) shows that the contents of a message added to a queue for an object can only contain references that were either contained in the message at the front of the queue of the sending object, or were the contents of some location in the store of the sending object.

3.3 A Primitive Notation

Each syntactic construct of Cantor could be defined as instances of the functions of c, δ, λ_{M}, and λ_{Π}. However, this approach would produce an extremely complicated interpretation scheme. To control the complication, the constructs of Cantor programs can be related to more elementary constructs. Each action of the elementary notation requires only a simple interpretation scheme achieved by associating with each state of S a primitive action and by expressing the primitive actions in terms of the functions c, δ, λ_{M}, and λ_{Π}.

3.3.1 Assignment

The action of assignment is the mechanism for modifying the persistent part of an object. Each location $l \in L$ denotes a variable whose content can change on a state transition. In the primitive notation assignment is written:

$$s_{\mathrm{asg}} : \mathrm{var} \leftarrow E$$

where $s_{\mathrm{asg}} \in S$ is the state where the assignment occurs. The variable "var" corresponds to a location $\mathrm{l}_{\mathrm{var}} \in L$. The letter E denotes an expression whose value is a member of V. The effect of the assignment is modelled by the δ function:

$$\delta(s_{\text{asg}}, c, k) = (s, c', k) \quad \text{where} \quad c'(l) = \begin{cases} c(l) & l \neq l_{var} \\ E & l = l_{var} \end{cases}$$

3.3.2 Expressions

An expression E as used in the assignment statement can be one of the following:

- a literal
- the content of a variable $(E \in c(l))$
- the component of a message $(E \in \text{acc}(\mu(r))$
- a simple expression
- a list expression
- a new expression

Arithmetic and logical operators such as addition and equality testing are introduced into the primitive notation by simple expressions. A simple expression consists of a result variable, an operator, and two operands. The operands of a simple expression can be literals, variables, or the components of a message. The effect of the simple expression is to perform the operation on the operands and store the result in the destination variable. Thus, a simple expression consists of a functional operation followed by an assignment action.

A list expression is written:

$$\text{t} \leftarrow \underline{\text{list}}\ n$$

and returns an ordered set of n unused locations from L. Lists are used exclusively for sending messages, since k-tuples of V are always inserted into the queue of an object. The k-tuples are allocated as a list, and the contents of the list are indexed by square brackets, e.g., t[1] $\leftarrow E$. The size and number of lists used in an object is fixed when the definition for the object is compiled; thus, lists do not require L to be of arbitrary size.

A new expression is written:

$$s_{\text{new}} : \text{t} \leftarrow \underline{\text{new}}\ obj$$

and returns a reference value r' for the new object obj. The reference value is assigned into t with the corresponding λ_Π transition of:

$$\lambda_\Pi(s_{\text{new}}, c, k) = (r', obj)$$

The new instruction is the only way new reference values can be introduced into a computation. Whereas the requirements for lists can be satisfied when the definition for the object is compiled, the requirements for the new instruction cannot always be precomputed.

3.3.3 Sequence

For assignment, and for the new and list instructions, the state where the action occurred was labelled by an element of S. For syntactic convenience, the default next state for an instruction is the instruction that lexically follows the current instruction. For cases in which the control flow changes from the next instruction to some other instruction, a *label* is used to designate the state corresponding to the other instruction. Thus, labels are used to identify a subset of the states for which the number of predecessor states is greater than 1.

Two instructions are included in the primitive notation to alter the control flow of a sequence of instructions. The instruction that changes the control flow unconditionally is:

$$s_{\mathrm{b}} : \underline{\text{branch}}\ \mathit{label}$$

where *label* designates the destination state. The effect of the branch instruction is modelled by the δ function:

$$\delta(s_{\mathrm{b}}, c, k) = (\mathit{label}, c, k)$$

The conditional branch instruction is written:

$$s_{\mathrm{if}} : \underline{\text{iffalse}}\ E\ \mathit{label}$$

where E is an expression that is evaluated to produce a logical (boolean) value. If the value is false, then the branch is taken. Otherwise the natural successor of s_{if}, written s_{nat}, is the next statement executed. The effect of the conditional branch instruction is modelled by the δ function:

$$\delta(s_{\mathrm{if}}, c, k) = \begin{cases} (s_{\mathrm{nat}}, c, k) & \text{if } E \\ (\mathit{label}, c, k) & \text{if } \neg E \end{cases}$$

3.3.4 Message Passing

The external influence of an object is captured in part by the new instruction and by the send instruction. The send instruction is written:

$$s_{snd} : \underline{\text{send}}\ E_l\ E_d$$

where E_l must be a list expression and E_d must evaluate to a reference value. The effect of the send action is modelled by the λ_{M} function:

$$\lambda_{\mathrm{M}}(s_{\mathrm{snd}}, c, k) = \mathrm{ins}(\mu(E_d), E_l)$$

The complement of the send instruction is the switch instruction, which advances the message queue for an object and places the object in an input state. The switch instruction is written:

$$s_{\mathrm{acc}} : \underline{\text{switch}}\ \mathit{label}$$

The branch and switch instruction have the same syntax and also have the same behavior except that *label* for switch must be an input state, and that the λ_{M} function for the s_{acc} state is defined as:

$$\lambda_{\mathrm{M}}(s_{\mathrm{acc}}, c,\ \mathrm{acc}(\mu(r)) = \mathrm{adv}(\mu(r))$$

3.4 Cantor $\mapsto$ Primitive Notation

Each syntactic construct of Cantor presented in the BNF description of Chapter 2 has a corresponding interpretation rule composed of instructions from the primitive notation. A reduction technique is used to define constructs as compositions of elementary instructions. The reduction technique often involves recursively defining both the statements and expressions found in Cantor programs.

The mapping between program variables used inside the definition of a Cantor object and the set of locations L for an object of O is defined by the *environment* of the object. The environment for an object is comprised of the persistent variables, the message variables, temporary variables introduced by let statements, and temporary locations introduced by the list instruction.

The list of persistent variables defines the *base* set of L which is augmented by the use of lists and let statements. The number of additional locations needed for list instructions and let statements is fixed when the definition is compiled. The management of these temporary locations is managed by the compiler.

The list of message variables corresponds to the contents of $\text{acc}(\mu(r))$, which is by construction a k-tuple of values. By considering the locations used for let variables and list instructions as part of the persistent list, only two lists are needed to identify a variables name with the corresponding location in L or the corresponding component of a message k-tuple. The two lists are called the *persistent list* and *message list*, respectively.

3.4.1 Message Flow

In Cantor programs, the pair of brackets: "[ ]" always denotes a state graph where the first state is an input state. Object descriptions of the form:

[(message list) S]

where S is a grouping of one or more statements translated into the primitive notation as:

```
label $L_1$
*statement(persistent list, message list,* S*)*
switch $L_0$
```

The state denoted by L_1 is an input state. The label L_0 denotes a special input state that has no successor states. A message sent to an object in the L_0 state is a programming error.

The inclusion of asterisk in front of an open bracket:

*[(message list) S]

is translated into the primitive notation as:

```
label $L_1$
*statement(persistent list, message list,* S*)*
switch $L_1$
```

Once the statements in S have been executed, the object will use the same sequence of statements to process the next message.

The interpretation of the keywords exit and repeat are straightforward:

$$
\begin{aligned}
[\ (\textit{message list}\ )\ \mathrm{S}\ \underline{\mathrm{repeat}}\ ] &= *[\ (\text{message list})\ \mathrm{S}\ ] \\
*[\ (\textit{message list}\ )\ \mathrm{S}\ \underline{\mathrm{exit}}\ ] &= [\ (\text{message list})\ \mathrm{S}\ ]
\end{aligned}
$$

The nesting of descriptions is handled by changing the *message list* of the environment. For example, a program with a nested structure of:

$$*[\ (\textit{message list 1})\ S_1\ [\ (\textit{message list 2})\ S_2\ ]\ S_3\ ]$$

is interpreted as:

```
label $L_1$
statement(persistent list, message list 1, $S_1$)
switch $L_2$
label $L_2$
statement(persistent list, message list 2, $S_2$)
statement(persistent list, message list 1, $S_3$)
switch $L_1$
```

The scoping rules of Cantor treat nested descriptions in the same manner that nested blocks are treated in block structured languages. The statements of S_2 are evaluated in the context of both message lists. If the same variable name appears in both the inner and outer description, then the message variable of the inner description will be the variable accessed. The statements of S_3 are evaluated in the context of the first message list since the inner message list was discarded once the close bracket for the inner description was reached.

3.4.2 Control Flow

Aside from nesting descriptions, there are only two mechanisms for controlling the execution of statements inside an object. The first is sequential execution. Statements are sequenced left to right, top to bottom, and are compiled in the order they occur. For example, two statements such as:

$$S_1\ S_2$$

are compiled from left to right:

```
statement(persistent list, message list, $S_1$)
statement(persistent list, message list, $S_2$)
```

The second mechanism is the if statement, which allows the sequencing of statements to be altered based upon the logical value of an expression. An if statement of the form:

$$\underline{\mathrm{if}}\ E\ \underline{\mathrm{then}}\ S_1\ \underline{\mathrm{else}}\ S_2\ \underline{\mathrm{fi}}$$

is interpreted using the iffalse instruction:

```
iffalse E $L_2$
statement(persistent list, message list, $S_1$)
branch$L_1$
label $L_2$
statement(persistent list, message list, $S_2$)
label $L_1$
```

3.4.3 Assignment

The interpretation of assignment has a one-to-one correspondence with the assignment instruction of the primitive notation. The interpretation of let statements is also a one-to-one correspondence with a new location added to the set L.

3.4.4 Expressions

The expressions recognized by Cantor are the same as the expressions of the primitive notation, except that simple expressions are replaced with more general compound expressions. A compound expression may consist of multiple operators and multiple subexpressions that are evaluated using the precedence structure presented in the BNF description of Chapter 2. The construction of simple expressions from a compound expression often requires introducing temporary variables. For example, an expression of the form:

```
var := x * 7 + y * 4
```

is interpreted by introducing two temporary variables:

```
$T_1 \leftarrow x * 7$
$T_2 \leftarrow y * 4$
var $\leftarrow T_1 + T_2$
```

Temporary variables are treated like let variables, except that the variables are introduced by the compiler instead of the programmer.

List expressions occur in conjunction with sending a message or creating a new object. New expressions occur only when a new object is created. A send statement is of the form:

send $(e_1, e_2, \ldots, e_n)$ to E

The expressions e_1 through e_n must evaluate to a literal or a reference value. The expression E must evaluate to a reference value. The interpretation for the send statement is:

```
$T_1$      $\leftarrow$   list $n$
$T_1[1]$   $\leftarrow$   $e_1$
$T_1[2]$   $\leftarrow$   $e_2$
              ⋮
$T_1[n]$   $\leftarrow$   $e_n$
send    $T_1$ $E$
```

New expressions occur when an object is created. In Cantor, creating an object requires allocating the reference using a new instruction and sending the new object an initial message containing the initial values for the persistent variables of the object. For example, a declaration for a new instance of an object of the form:

$$\text{var} := \text{f}\ (e_1, e_2, \ldots, e_n)$$

is interpreted as:

T_1	$\leftarrow$	list n
$T_1[1]$	$\leftarrow$	e_1
$T_1[2]$	$\leftarrow$	e_2
	$\vdots$	
$T_1[n]$	$\leftarrow$	e_n
var	$\leftarrow$	new f
send	T_1 var	

The initial persistent message sent to an object must be distinguished from ordinary messages sent by a send instruction. The primitive notation does not include a mechanism to distinguish the initial persistent message, since there are many ways to define an interpretation rule to distinguish a message. For example, the initial persistent message is sent first and explicitly acknowledged before allowing additional messages to be sent to the object. Another interpretation would be to add a component to every message k-tuple to *tag* the message as either an initial persistent list or an ordinary message.

3.5 Flow Analysis of Cantor Programs

Cantor is a dynamically typed notation where, in the most general case, any variable may become bound to any value. The computing system that supports a Cantor implementation must provide type checking wherever necessary. For example, applying values to arithmetic and logical operators requires checking that the operator is defined over the types of the operand values. Another example is checking that the destination of a send command is of type reference.

In addition to type checking, because the destination of a send command can potentially be any reference value, the computing system must also provide message delivery from any object to any other object.

If the data type for a variable could be inferred for the program points where the variable is accessed, then the type check could be performed when the object definition is compiled, and the type check at runtime eliminated. Likewise, if all the reference values used by an object could be inferred at runtime, the object would not need the capability to send a message to any other object, only a subset of the objects.

The task of inferring data type is an instance of the more general problem of propagating constants through a program. Each value in a Cantor program can be thought of as comprised of two or three parts:

tag	item

or

ref	def-type	instance #

The tag field denotes the data type. The data type identifies the set from which the value is taken. For example, the type boolean is a set comprised of only two elements: the literals true and false. All values denotable in Cantor can be represented by these two fields. The item field for type reference is, however, further partitioned into two more fields: the *def-type* and *instance number*. The def-type is the object definition that was used to create the object via the new instruction. The instance number identifies the reference value as the *n*th instance of the object definition.

This representation for values is only one of many possible schemes. For example, the distinction between reference and def-type could be replaced by representing each object definition as a new data type. The above representation was selected because it fits well with the organization for the program flow analysis.

The flow analysis is first organized by object definition. The first layer is to propagate constant tags through each variable for each program point of the object definition. The benefit of Layer 1 analysis is the removal of unnecessary type checking during program execution. Layer 2 flow analysis augments Layer 1 by propagating constant def-types for variables whose tags are references. The results of Layer 2 analysis can be used to refine the results of the Layer 1 analysis and are potentially useful for the dynamic placement of new objects. Layer 3 analysis augments Layer 1 and 2 analysis by propagating constant values;, i.e., both tag and item are constant, through the variables for each object definition. By identifying constant reference values prior to program execution, the assignment of objects to processing nodes can be performed in part, possibly completely, before the program is started, thereby simplifying or eliminating the task of placing objects during program execution.

The propagation of constant values where the tags are integer, real, boolean, or symbol is straightforward. For example, consider the assignment statement:

```
x := y + 2
```

where y is inferred to be the constant 40. The add operation can then be performed at compile time and x is replaced by the constant 42. Reference values can also be propagated at compile time if the new instructions can be performed at compile time. For example:

```
y := 40
```

generates the integer 40 at compile time and binds it to the variable y. Likewise, the statement:

```
z := f(x,y)
```

generates an instance of the object f and binds it to the variable z. Whether f(x,y) can be treated as a constant value depends upon how f is defined. The crucial property of all the built-in data types, e.g., integers and booleans, is that they are history-insensitive. An integer value of 40 generated in one object is identical in behavior to an integer value of 40 generated elsewhere. The meaning of a program is preserved whether a history-insensitive object exists as a shareable single object, or as multiple instances of the same def-type.

Object definitions that are history-insensitive and independent of any data dependencies can be instantiated at compile time. The compiler or program performing flow analysis has a great deal of latitude in managing these objects. For example, one instance of each history-insensitive object def-type could be created, and each new instruction that creates an object of the same def-type would be replaced by the single constant. Such an approach is attractive for sequential implementations since some objects can be statically allocated, but is unsuitable for concurrent implementations since the single instance of the object is both a potential bottleneck with respect to message traffic, and an unnecessary serializer with respect to concurrent execution. For concurrent implementations, a first improvement would be to generate constant, history-insensitive objects on a per definition basis rather than a per program basis.

For objects which persist only for the duration of processing a single message and for which only one instance will ever exist, *all* new objects created by the object can be generated at compile time and propagated as constant values. For this case, the flow analysis *equates* the single instance characteristic of the object with the single definition for the object. The generalization is to propagate constants on a per object instance basis instead of an object definition basis. Flow analysis based upon propagating constants over object instances, called *future flow*, is tantamount to symbolically executing the program.

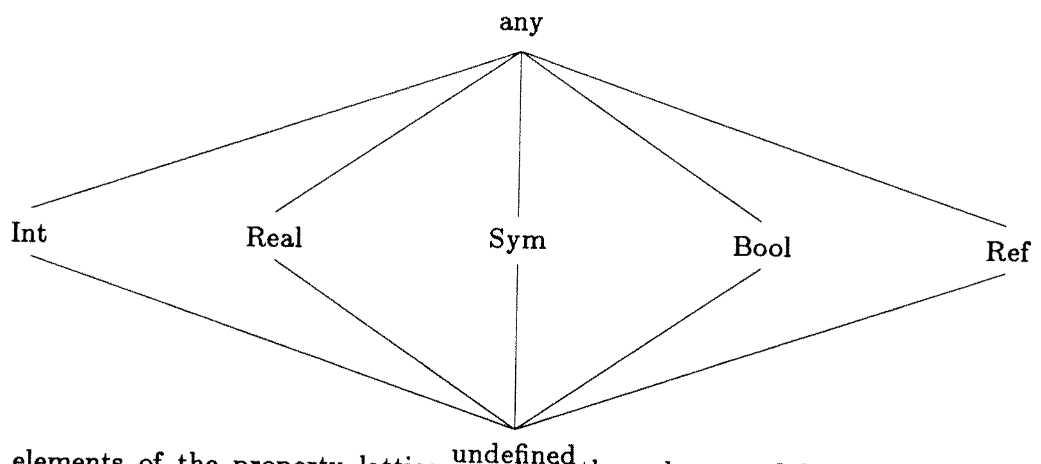
The advantages and limitations of future flow can possibly be best understood by explaining the procedure for propagating constants on a per definition basis and then generalizing to the per instance basis. The sources for values are:

- Message Variables
- Persistent Variables
- Let Variables
- Temporary Variables
- Expressions

Values received from external objects as a message by the main object are presumed to be known when the program is flow analyzed.

3.5.1 Layer 1 Flow Analysis

The goal of constant propagation is to establish a set of *properties* for each program point where a variable is used. A simple property lattice [29] corresponding to a Layer 1 analysis is the following:



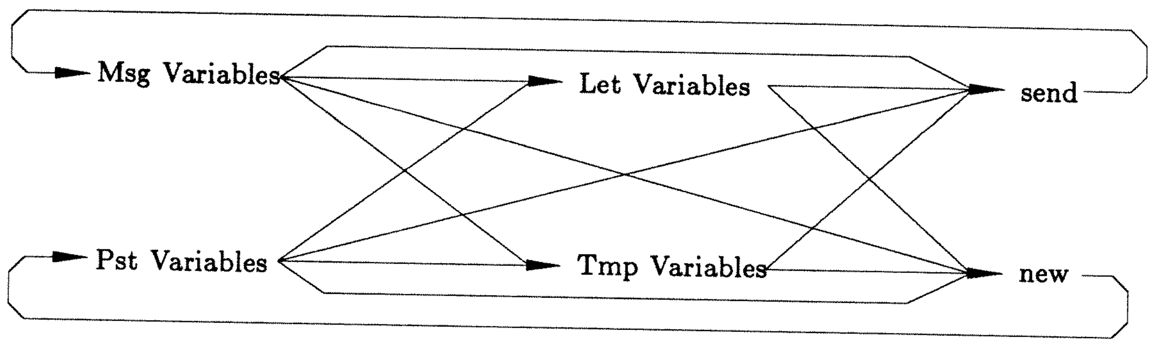
The elements of the property lattice supplant the values used by Cantor programs. Initially, every variable at every program point is assigned the undefined property. For each instruction of the primitive notation, a transformation over the property set is applied. The transformations on the elements of the property set are defined in terms of commutative *meet* (|) operations:

1. undefined | tag $\mapsto$ tag where tag is Int, Real, Sym, Bool, or Ref.
2. tag | tag $\mapsto$ tag where tag is Int, Real, Sym, Bool, or Ref.
3. tag_1 | tag_2 $\mapsto$ any where $\text{tag}_1 \neq \text{tag}_2$.
4. any | tag $\mapsto$ any where tag is Int, Real, Sym, Bool, Ref, any, or undefined.

In a sense, each object definition is *executed* over the simplified domain of the property set. Each primitive instruction needs to be modified at most twice. The number of times the properties of the instruction are examined depends upon how many different instruction paths merge upon the instruction. Since values are only propagated *within* a definition, all of the persistent and message variables must be treated as possessing property any.

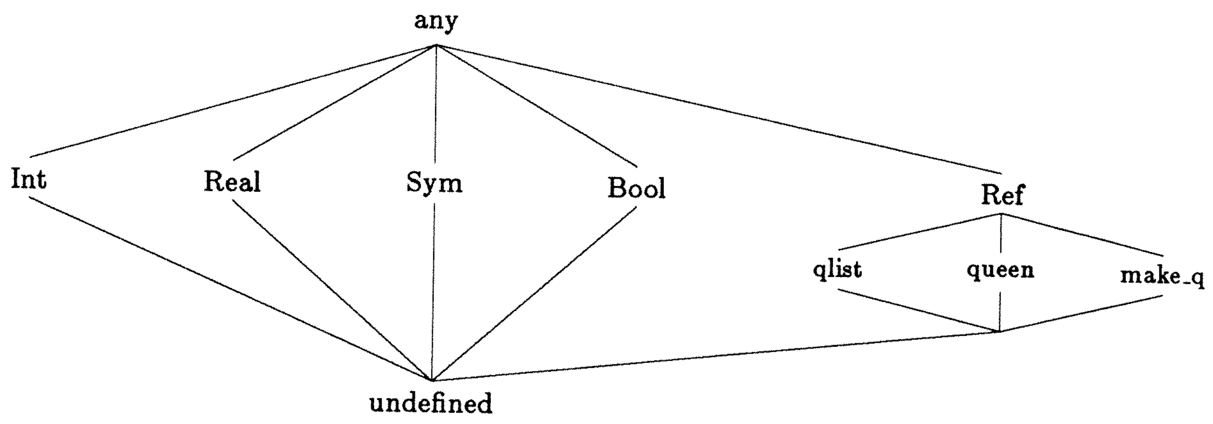
3.5.2 Layer 2 Flow Analysis

By performing a Layer 2 analysis, the types of persistent and message variables can be inferred by examining the interaction *between* object definitions. The overall flow of values through variables between object types is shown by the following picture:



Value types for persistent variables are generated from new instructions, and the types for message variables are generated from send instructions. The types generated by these instructions in turn feed back to define the types for the other kinds of variables used inside of the object definitions.

For the Layer 2 analysis, an additional layer needs to be added to the property lattice. The flow analysis needs to keep track of the def-types as an added property of the type reference. The exact shape of the lattice is now program dependent, since the number of def-types is determined by the number of object definitions inside of the program. As an example, a partial property lattice for the eight queens program is:



The goal of the Layer 2 inference analysis is to determine on a definition by definition basis what objects can create other objects: the new graph, and what objects can send to other objects: the send graph. The new graph is the easier of the two to build since the new instruction has the def-type embedded in the instruction. The type of each persistent variable for an object type is determined by *meeting* each component of the persistent list for the definition type with all the def-types that can create the object type.

The send graph is more difficult to build since the def-type is embedded in the reference value and not the send instruction. Also, the definition for an object may have many different message lists specified for it, e.g., the case construct, whereas there is only one persistent list. For the case of the multiple message lists for a definition, the message lists are combined into a single, composite list.

The construction of the send graph is coincident with the inference procedure. As each destination reference value for a send command is inferred, an edge is added to the send graph, and the *meeting* operation described for the new graph is applied. For the Layer 2 analysis, meet rules for def-types must be included:

- Ref | def-type $\mapsto$ Ref.
- def-type | def-type $\mapsto$ def-type.
- def-type$_1$ | def-type$_2$ $\mapsto$ Ref where def-type$_1$ $\neq$ def-type$_2$.

For each definition, a modified def-use (MDU) graph [28] is built to express the dependencies between variables and also between def-types. Each vertex of an MDU graph denotes either a segment of straight line code (SLC) from the primitive notation or an object definition. For each object definition, there is a single MDU graph. A directed edge from one SLC vertex to a second SLC vertex denotes that one of the variables defined by the vertex at the tail of the edge is used by the vertex at the head of the edge. An SLC vertex may also have an edge pointing to an object definition vertex. These edges are tagged as either new or send and indicate that the SLC vertex can create a new object of the def-type pointed to, or can send a message to the def-type pointed to, respectively.

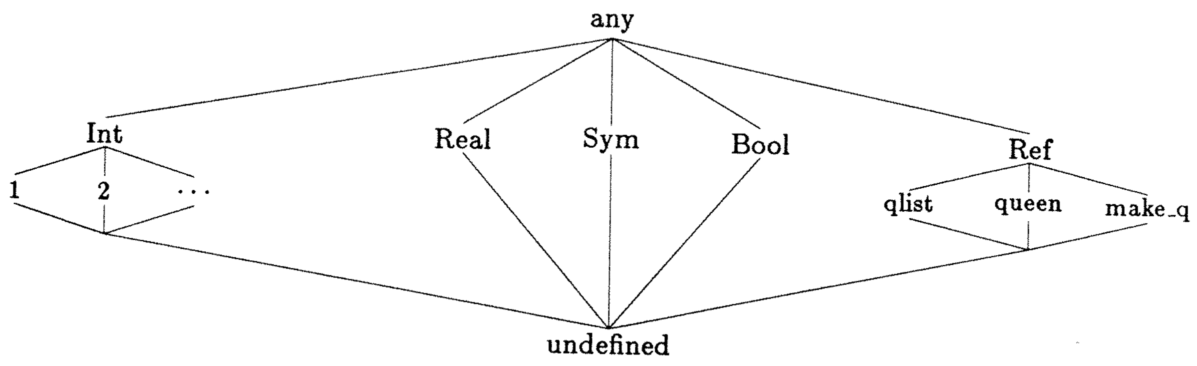
The inference algorithm propagates properties through the MDU graphs until no variables change properties. Each object definition vertex is represented by a *template* comprised of an MDU graph, a list for the properties of the persistent list, and a list for the properties of the message list. Each component of each list is initially set to undefined, except possibly for the message list of the main template. The main template is the only object definition initially scheduled for propagation. Starting with the persistent and message list, the properties are propagated through the MDU graph. For each SLC vertex containing a new instruction, the persistent list of the template for the new object is *meeted* with the old persistent list of the template. If the result of applying the component-wise meet operation changes any of the properties of the template, then the template is scheduled for propagation.

For each SLC vertex containing a send instruction, if the destination is of known def-type, then the message list of the send instruction is meeted with the message list of the destination template. Otherwise, all object definition templates are meeted with the message list. If any of the properties of a template changes as a result of the meet, then the template is scheduled for propagation.

The algorithm is guaranteed to terminate, because the total number of vertices for all of the MDU graphs is finite, and because the property set is a finite lattice. If all of the variables contained in the union of the MDU graphs are considered to be elements of one large set V, then for each complete propagation of types through all of the MDU graphs, at least one variable must change, else a steady state has been reached. Each variable defined can be changed at most three times, once for def-type, then ref, then any. Thus, the number of complete propagations is bounded by V^3.

3.5.3 Layer 3 Flow Analysis

For the Layer 3 analysis, an additional layer in the lattice is added for the types integer, real, symbol, and boolean. This new layer contains value instances of the type. For example, consider the property lattice for the 8-queens program where type integer has been augmented with instances of integers:



The meet rules are once again expanded to include:

- (tag,item) | (tag,item) $\mapsto$ (tag,item).
- (tag,item$_1$) | (tag,item$_2$) $\mapsto$ tag if item$_1 \neq$ item$_2$.

If the tag and item match exactly, then both are propagated as a constant value, and all calculations that exclusively use constants can be performed when the program is compiled and replaced with a constant. If the tags match but not the items, then the type can be propagated, and all calculations that exclusively use values of constant type can be type checked when the program is compiled, with the resulting type known. As an aside, if for example the operation is equality testing and the tags do not match, then the calculation for the resulting value can also be done at compile time.

To propagate instances of def-type as constant values, the meet rule for the Layer 3 analysis is refined to include the possibility of equal def-types, but not equal instance numbers:

- (ref,def-type,instance) | (ref, def-type,instance) $\mapsto$ (ref,def-type,instance)
- (ref,def-type,instance$_1$) | (ref,def-type,instance$_2$) $\mapsto$ (ref,def-type) where instance$_1 \neq$ instance$_2$

The value of propagating object constants, a.k.a. futures, by this technique is of limited value, because at most one future per program point can be assigned at compile time. If two different futures meet at a program point, then the property of the variable at the program point becomes either (ref,def-type) or simply ref if the futures were not generated by the same template.

To remove the limitation of one future per program point, the flow analysis can be performed on a per instance, or per future, basis rather than a per object definition basis. With respect to the inference algorithm, the only change is to replace the templates with futures. The algorithm is no longer guaranteed to terminate, since an infinite number of futures may be generated. Thus, both the property lattices and MDU graphs can potentially be infinite. The advantage is that the objects and their connectivity can be inferred at compile time.

Under future flow analysis, at one extreme the property lattice contains all of the values of Cantor's value domain, and consequently, the meet rules *exactly* interpret the primitive instruction set. Under these conditions, futures are instances of objects, and the inference algorithm is executing the program *in toto*. At the other extreme, the property lattice contains only futures. The MDU graphs for future-only inference are quite small, consisting mostly of send and new instructions. Propagating futures over the MDU graph *concurrently* with executing the object may prove to be advantageous, since the runtime system can look ahead to find the connectivity of a new object with other objects before the corresponding new instruction is executed.

A myriad of property lattices can be defined between these two extremes. For examples, if some of the data types, e.g., reals and symbols, are excluded from the property lattice, then futures are partially instantiated objects. A framework excluding reals and symbols could be used to do the equivalent of *unrolling* DO loops as is performed in optimizing FORTRAN compilers [4].

3.6 Reclaiming Irrelevant Objects

After flow is the framework for recovering lost or irrelevant objects during the course of a computation. The framework is unique to Cantor and the object model, but the algorithms are refinements of the techniques for garbage collection done by Dijksta, Lamport, Martin, Scholten, Steffens [16], and by Hewitt and Baker [7]. An initial working definition for the set of irrelevant objects is those objects that can *never* receive a message. This set is calculated by first determining the set of objects that are capable of receiving messages, either directly or indirectly, and then removing this set of relevant objects from the total set of objects used by the computation. These calculations require that either the set of all possible references values be finite, or that there exist auxiliary mechanisms to keep track of all the reference values used in a computation. While the former is preferable for the equations that follow, the latter is presumed to be favored for implementation.

For the task of reclaiming objects, a computation is approximated by the pair (h, R_m) where:

R is the set of all reference values.
$R_m \subseteq R$
$h : R \rightarrow R^*$

The π function of the computational model's formal definition maps reference values into object tuples. Each object tuple contains a set of a locations L and a a mapping function C. The relation h is derived from π, L, and C by taking only those values from the range of C that are reference values. The set:

$$h(r) = \{\forall l \in L \text{ s.t. } C(l) \text{ is a reference value} \mid C(l)\}$$

defines the set of objects that can be sent messages from the object r. The set R_m contains all of the reference values that are contained in messages not yet delivered. Let the set R_d be the set of reference values that are the destinations for the set of messages not yet delivered. For each tuple of values in $\mu(r)$, all of the values that are reference values must be included as possible destinations for new messages. This is the *safe* approximation to be made, since the fate of the reference value cannot be determined without first delivering the message. In summary, the set R_m is the union of the set R_d with the set of reference values contained in the undelivered messages.

3.6.1 A Two Color Strategy

Using the pair (h, R_m), computing the set of reference values that can directly or indirectly receive messages is the union of the transitive closures (TC) of the h relation for each reference value in the set R_m, written $\mathrm{TC}_h(R_m)$. The transitive closure can be computed by message-passing and can be performed concurrently to the computation proper.

Computing the transitive closure by message-passing involves sending messages to mark the objects that belong to a closure. For the moment, consider only the problem of marking the relevant objects after the computation has been stopped. To differentiate marked objects from unmarked ones, marked objects are assigned the color black and unmarked ones are assigned the color white. Initially, all objects are white. When a message is delivered to a white object, the object clearly belongs to the transitive closure and is colored black. The next step is to propagate the marking phase to all of the objects reachable from the current object. The propagate messages are sent to all of the reference values contained in the message received and to all of the reference values that are contained in the persistent storage locations of the object. These additional messages contain no reference values other than their destination and are solely used to color the objects reachable from the current object.

A message received by a black object results in messages sent only to the reference values contained in the message, excluding the destination reference value.

This strategy will eventually mark all of the reachable objects. The marking phase terminates when all messages have been delivered. To prove termination, let the initial set of messages be assigned the color white and let the set of messages used for coloring the objects reachable from the current object be assigned the color gray. Gray messages contain no reference values other than their destination. Initially, all objects and all messages are white. The supply of white messages will monotonically decrease since all of the white messages are guaranteed to be eventually delivered, and because no white messages are ever emitted. Once all of the white messages have been delivered, the objects denoted by the set R_d will have been colored black.

Gray messages are generated when:

1. A white message is received by a white object.

2. A white message is received by a black object.

3. A gray message is received by a white object.

The number of gray messages generated by 1 and 2 is finite, because the supply of white messages is finite. For case 3, a gray message received by a white object will change the color of the object to black.

Once the supply of white messages is depleted, the system will contain only gray messages, and white and black objects. Gray messages delivered to black objects result in no new messages sent; thus, the count of gray messages decreases by 1.

Gray messages delivered to white objects can increase the number of gray messages since each reference value stored inside of the object will be sent a gray message. To prove that the quantity of gray messages will eventually be depleted, the set of gray messages is partitioned into two subsets, those destined for white objects and those destined for black objects. The set of gray messages destined for black objects will not affect the color of any object, nor will they cause any additional messages to be generated.

The set of gray messages destined for white objects can be reduced in size by replacing multiple messages destined for a common object with a single message. This transformation is valid, because the first message delivered will change the color of the object to black. The remaining set of messages then belongs to the set of gray messages sent to black objects. The size of the set once the transformation has been applied is bounded by the number of white objects.

When a gray message is delivered to a white object, the number of gray messages may increase, because gray messages are sent to the reference values stored in the contents of the storage locations private to the object. However, the color of the receiving object changes to black. By performing the partition and reduction steps, the maximum size of the set of gray messages sent to white objects is guaranteed to decrease by at least 1, because there is one less white object. Thus, the maximum size of the set of gray messages sent to white objects is guaranteed to monotonically decrease.

When the maximum size of the set of gray messages sent to white objects reaches zero, all of the remaining messages must be in the other subset. These messages are guaranteed to be consumed without producing any additional messages. Therefore, the supply of gray messages will eventually be depleted.

The depletion of gray messages implies that all of the objects that could receive messages, either directly or indirectly, have been colored as black. Any remaining white objects are unreachable and can be reclaimed as irrelevant.

To allow the computation proper to proceed concurrently with the computation of the transitive closure, the termination condition that all messages be delivered must be modified. If the computation proper is to proceed, new messages and new objects are to be generated. Producing new objects poses no difficulty, provided that the new objects are created as black objects. Producing new messages presents a difficulty, since basing termination upon waiting for all messages to be delivered requires that the computation proper terminate. This condition is too strong. The solution is to add a third color for messages. A message sent from a black object that pertains to the computation proper is a black message. The new termination condition is that all white and gray messages have been delivered.

When a white object receives a white message, gray messages are sent as before and the object is colored black. The object will then send black messages and create black objects in response to the computation proper.

A black message will never be sent to a white object if message order is preserved between objects in direct communication. The proof is by contradiction, assume a black message is received by a white object. The source of the black message was a black object. Before a white object is marked as a black object, all of the reference values contained in the message and contained in the object are sent gray messages. For a black message to be received by a white object, the black message must have overtaken the gray message. However, this violates message order preservation.

Since a black message is never received by a white message, and the supply of gray and white messages is guaranteed to eventually be depleted, then once the system contains exclusively black messages, no messages can ever be delivered to any remaining white objects. These white objects can be reclaimed.

An attractive feature of this algorithm is that once all white and gray messages have been delivered and all remaining white objects removed, the system contains only black references. The algorithm can now be executed again by exchanging the roles of white and black colors. Thus, the game remains the same but the roles are interchanged.

In summary, the algorithm for garbage collection presented in this section is simple for two reasons. First, garbage collection is performed only on objects and not on smaller atoms such as the CONS cells of a LISP system. Secondly, objects are marked either before or after assignment operations are performed, but not during. Thus, the granularity of interleaving the coloring of objects with the computation proper is performed at the level of processing messages as opposed to the tremendously more difficult level of processing instructions.

Chapter 4

The Behavior of Concurrent Programs

Program behavior is the observable interaction between a computer program and the environment where the program is interpreted or executed. The interaction between the program and its runtime environment consists primarily of demands for resources made by the program during its execution. For programming models in which most of the resources are allocated prior to running the program, e.g., FORTRAN and OCCAM, program behavior is relatively tame. For programming notations such as LISP and Cantor, the resources demanded by a program become known only during the course of the computation, and program behavior is therefore quite dynamic.

Cantor places three types of resource demands upon its host environment:

1. Message-passing
2. Message processing
3. Object creation

For message-passing ensemble machines, message-passing and object creation require that communication channels be allocated to transfer a message between two nodes. Message-passing and object creation also require that storage be allocated. Message processing requires that an instruction processor be dedicated to processing the contents of a message.

Assuming that each object is stationed to a single node, that messages travel from the sending object to the destination object, and that an object only processes messages on the node where it is stationed, then management of the three resources is controlled exclusively by the assignment of objects to nodes. The object of the assignment or placement task is to maximize progress. The efficiency of the placement strategy with respect to maximizing progress is measured in terms of load balancing and preserving locality. Load balancing is the task of placing objects so as to avoid excessive competition for shared resources in some nodes while other nodes remain idle. For programs where the level of concurrent behavior is less than the number of nodes, some of the nodes will always be idle. Preserving locality is the task of placing objects so that objects that communicate frequently are kept in a close proximity to each other. Locality can be maximized by placing all of the objects for a program on a single node. This placement strategy is the worst for load balancing unless the program is sequential. In general, load balancing and preserving locality are opposing constraints.

The task of assigning objects to nodes can be conducted at three separate levels:

- Programmer directed
- Compiler directed
- Runtime directed

Permitting the programmer to direct the assignment of objects to nodes is flawed for several reasons. First, the programmer may use library objects written by other programmers. Because the programmer does not necessarily know the internal workings of the library objects, the interaction between the placement of the programmer's objects with the placement of the library objects may be detrimental. Secondly, if the nodes are time-shared, i.e., multiple programs per node, then the interaction between the different programs may be detrimental. These two arguments are not new. The same arguments were made over a decade ago in support of virtual memory [14].

For the programmer to make decisions about placement, the decisions must be made when the program is written. Likewise, the compiler must make placement decisions when the program is compiled. The difficulty with making the assignment early is that the interconnection topology and number of nodes dedicated to the program may change before and during the execution of the program. The topology may change due to constraints such as fault tolerance, or space-sharing, in which the user is assigned an exclusive subset of nodes for a program. For either case, by presuming one topology and directing the assignment task accordingly, the resulting program may perform poorly if the realization is not close to the expectation.

Compiler directed placement has two advantages. The first advantage is that it can resolve the library object problem by examining the internal workings of all the objects. However, the compiler cannot for all programs construct the object graph envisioned by the programmer. For example, object graphs that are data dependent cannot be constructed without executing the program. Compiler directed placement has the advantage of access to global information about the interaction between the component objects, but is unable in the general case to construct the object graph that the programmer has planned for. The second advantage of compiler directed placement is that the compiler can access information about the interconnection topology of the target network. This information combined with the predicted object graph enables the compiler to make an initial placement of objects to nodes.

Runtime directed placement solves the above problems by making the assignment during program execution. Data dependencies, interconnection topology, and the number of nodes dedicated to the program are all known, as are the number of objects and messages resident on each node. Unfortunately, for ensemble machines at least, data about the state of the computation is localized to each node and possibly the set of neighboring nodes. Each node must make decisions about placing objects based on limited and perhaps stale information about the state of the other nodes in the ensemble. The amount of immediately available information is limited to the node itself and perhaps the set of neighboring nodes. Obtaining accurate global information about the state of the computation is not practical.

The interactions between the program and runtime system are therefore quite complex. There are two facets to load balancing and preserving locality: making prudent initial choices and migrating objects to improve load balancing and locality. The experiments presented in this chapter address only the first facet. The approach is to provide a case study of the six programs presented in Chapter 2. The programs are first examined under an ideal set of conditions to understand their dynamic characteristics, in particular, to establish a best case performance. The ideal conditions are then incrementally removed and replaced with more realistic conditions. Under these altered conditions, different load balancing and locality strategies are evaluated.

4.1 Program Behavior for Soft Computers

To establish a base case for program behavior where the effects of the hosting environment are minimized, a hypothetical idealized ensemble machine called a *soft computer* is defined. The soft computer consists of a variable number of computing *sites*. The number of computing sites adapts to the number of concurrent objects extant in the object program. Thus, there is a one-to-one mapping between the set of concurrent objects and the set of computing sites. Every opportunity for concurrency is exploited, and load balancing is optimal.

Message delivery between two computing sites in the soft computer model is instantaneous, and the bandwidth of the message-passing network is infinite; thus, locality is inconsequential. After a message is delivered to a computing site, the message is processed in a single time quantum. This time quantum is the same for all messages and objects. For purposes of exposition, the processing of messages are synchronized over all the computing sites. Computations are organized into sweeps. For each sweep, every computing site (object) that has one or more messages enqueued is allowed to process one message. All of the messages sent in response to processing a message are immediately delivered. Likewise, all new objects are immediately instantiated.

The execution of an object program consists of a number of sweeps. For each sweep the following data is tabulated:

- total number of messages
- total number of active objects
- total number of objects
- number of messages delivered this sweep

An active object is defined as an object that has one or messages queued for it. The total number of active objects is called the *concurrency index* (CI). The ratio of the total number of messages to the concurrency index is called the *message load* (ML). The message load value is useful when developing programs to determine how heavily the objects used in the program are overloaded. The optimal value for the message load is one, and all six test programs except for the perfect numbers program have constant message load values of one. Therefore, all ML graphs except the one for perfect numbers are omitted.

4.1.1 Perfect Numbers

The perfect numbers program is profiled in Figures 4.1 and 4.2. The program was truncated to find perfect numbers up to and including 3,000 instead of 10,000. Figure 4.1 shows the number of objects in use by the program as a function of the sweep count. Figure 4.2 shows the number of concurrent objects per sweep. The number of objects grows linearly until the 3,000 test number is reached. The iterative behavior of the iter object is terminated, and the number of objects rapidly drops to zero. The growth rate for the CI graph shows a parabolic tapering as the number of objects involved in the computation steadily increases. An attractive feature of both graphs is that the total sweep count is 3,026 for testing 3,000 numbers. The sweep count is indicative of a pipelined behavior inside the program.

The message load graph of Figure 4.3 displays the message load value fluctuating between 1.00 and 1.25. This fluctuation is due to the adder object that receives two messages, adds their contents, and then sends one messages. For the cases where the two input messages arrive during the same sweep, the message load value exceeds 1.00. The occurrence of these cases is dependent upon the distribution of divisors for a test number.

4.1.2 Prime Sieves

Figures 4.4 and 4.5 show the number of objects and concurrency index as a function of the sweep count for the wheel driven prime sieve of Section 2.4.2. The source program used to make the two graphs is identical to the program of Figure 2.9, except that the message send to output in the sieve object was removed. The number of sweeps necessary to compute all primes less than 10,000 is 4,241. This is the absolute minimum for this program, assuming that every opportunity for concurrency is exploited. The number of objects grows at a steady, monotonically increasing rate, reflecting the growth of the sieve.

The concurrency index also grows steadily, since concurrent behavior is achieved by the stages of the sieve working concurrently. The concurrency index continues to increase until the number generator that provides input to the sieve is shut down. The sieve then begins to empty out with the concurrency index monotonically decreasing.

The plots for the object count and concurrency index are shown in Figures 4.6 and 4.7. Like the plots for the wheel driven version, the message send to console inside of sieve was removed. The number of sweeps to completion is 12,269, which is about a factor of 3 greater than the wheel driven version. The overall shape of the two object graphs and concurrency index graphs are the same. The concurrency index graph for the demand driven prime sieve is "bumpier" than the wheel driven version. The factor of 3 difference in number of sweeps and the bumpiness is due to the handshaking between the stages of the sieve. For the wheel driven version, each transmission of an integer between sieve objects requires 1 message to be processed. For the demand driven version, 3 messages are processed: request integer, answer integer, and accept reply.

4.1.3 Merge Sort

The object count and concurrency index graphs for the merge sort program are shown in Figures 4.8 and 4.9. Because the interesting concurrent behavior for merge sort occurs over a narrow sweep range, the concurrency index graph is plotted using a logarithmic scale. The first 1,000 sweeps of the merge sort program are used exclusively by the make_list object to build the list of 1,000 integers. The program is sequential for these 1,000 sweeps, and the object count linearly increases to 1,000. The split objects then start halving the list, with the concurrency index doubling each time the list is halved. After the original list is separated into lists of single elements, the lists are merged. The concurrency index now keeps decreasing by a factor of two each time the length of the list is doubled. Because only half of each list is accessed to split a list, but to merge two lists often requires that every element of both lists be accessed, the number of sweeps to merge two lists is roughly twice the number of sweeps needed to halve the lists. The object count and concurrency index graphs show the symmetry of the two operations: splitting and merging; and also show that merging requires twice the number of sweeps of splitting.

4.1.4 Eight Queens

The object count and concurrency index graphs for the eight queens programs are shown in Figures 4.10 and 4.11. The output phase of the program was inhibited by commenting out the line: "send ("advance", console) to qm". The program exhibits a colossal amount of concurrency and finds all 92 solution in less than 300 sweeps, or roughly a solution every 3 sweeps. The cost for this performance is an immodest number of objects, peaking out at slightly over 5,000. Most of these objects however are purely transitory. The final object count is 830, which includes the 92 solutions requiring 9 objects per solution plus the printer and queue_master object.

4.1.5 Gaussian Elimination

The plots for the object count and concurrency index for the Gaussian elimination program are shown in Figures 4.12 and 4.13. The "send ("print",output) to orig_matrix" statement inside the compare object was commented out to defeat the 4,096 sequential print messages that are used to output the reduced the matrix at the end of the computation. The object count graph is not interesting; build_matrix and build_list objects cause the number of objects to grow to approximately 4096, i.e., a 64 by 64 array of list objects. The object count remains at this level for the duration of the program. The growth rate is greater than linear because the matrix is built concurrently.

The concurrency index is quite interesting. For the find pivot, move pivot row, and adjust pivot row steps, there is no concurrency. The initial build matrix phase and the the add multiples of top row to rows beneath top row step are concurrent. For the add multiples of top row step, the concurrency index equals the number of rows left in the matrix.

By the iterative nature of the program, as the matrix becomes one row and one column smaller, the concurrency index decreases by one. Likewise, the interval between the concurrent "phases" is also decreasing by 1. Therefore, as the computation progresses, the concurrency per phase monotonically decreases, and the interval between concurrent actions also decreases monotonically.

This type of concurrent behavior can be expected from a large class of concurrent programs. In contrast to the divide and conquer nature of the perfect numbers and eight queens, or the pipelined behavior of the prime sieves, the behavior for Gaussian elimination is a sequential search, i.e., find pivot row, followed by a concurrent application of the result of the search to the components of the data base, i.e., the elements of the matrix.

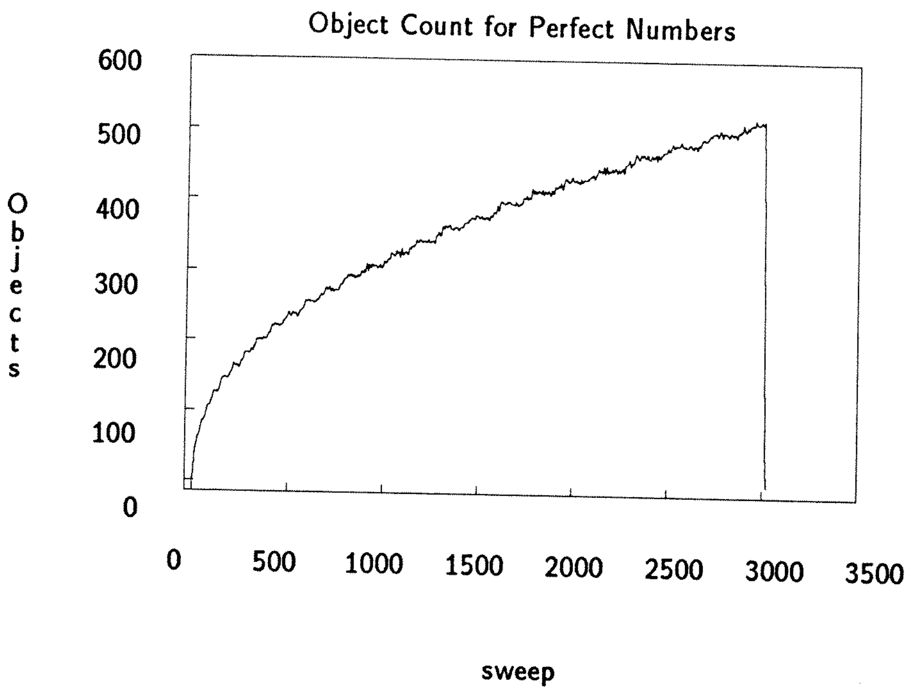


Figure 4.1: Object Count for Perfect Numbers 3,000 ($N = \infty$)

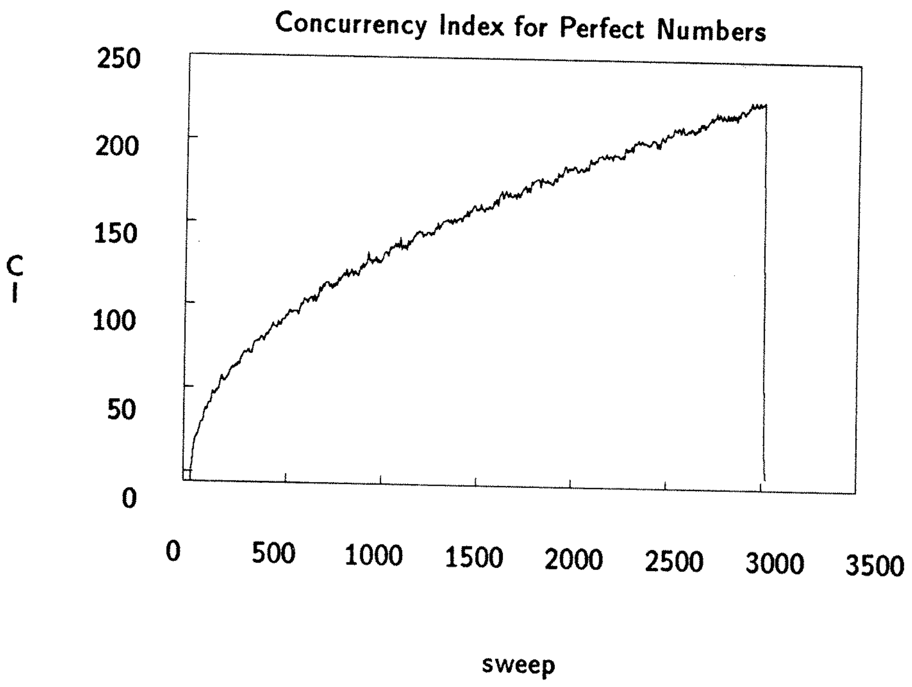


Figure 4.2: Concurrency Index for Perfect Numbers 3,000 ($N = \infty$)

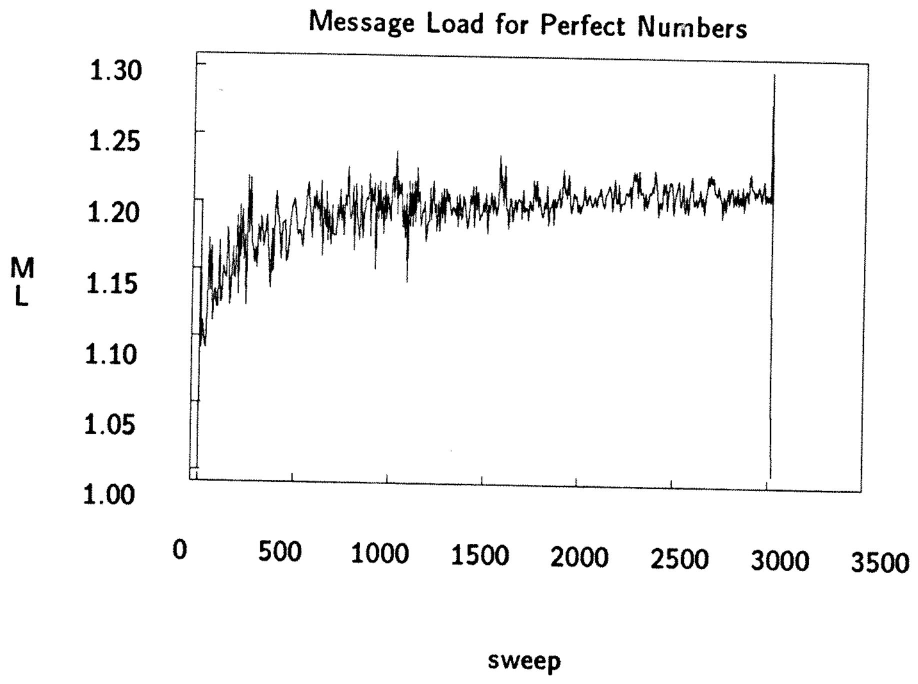


Figure 4.3: Message Load for Perfect Numbers 3,000 ($N = \infty$)

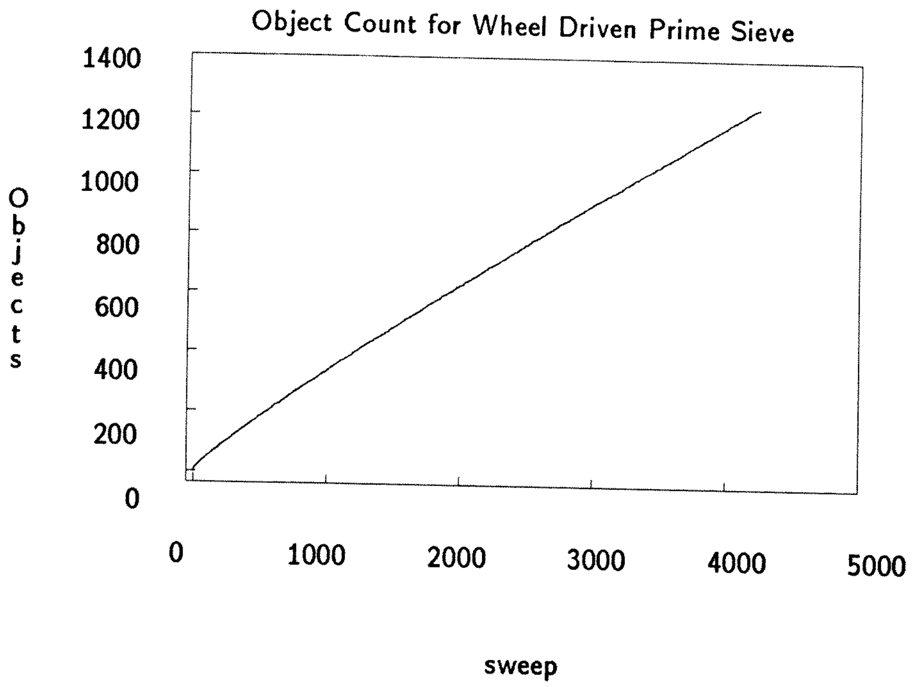


Figure 4.4: Object Count for Wheel Driven Prime Sieve ($N = \infty$)

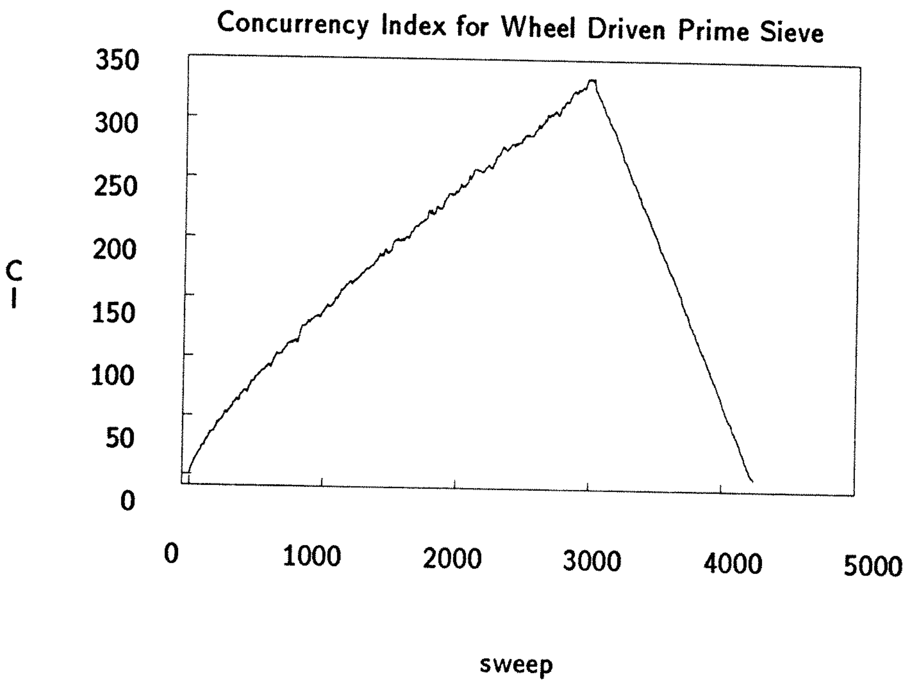


Figure 4.5: Concurrency Index for Wheel Driven Prime Sieve ($N = \infty$)

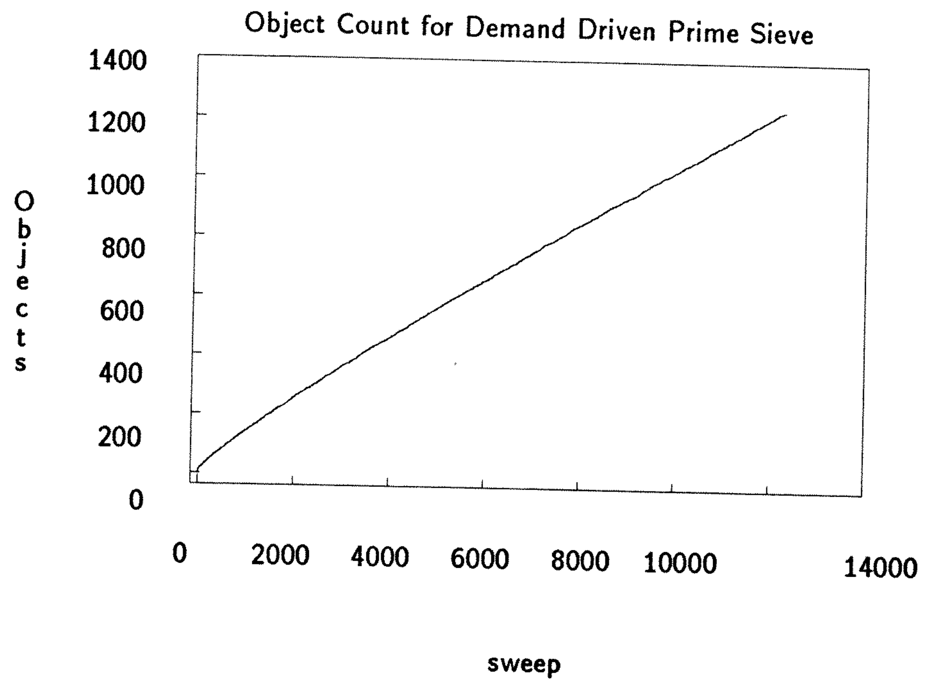


Figure 4.6: Object Count for Demand Driven Prime Sieve ($N = \infty$)

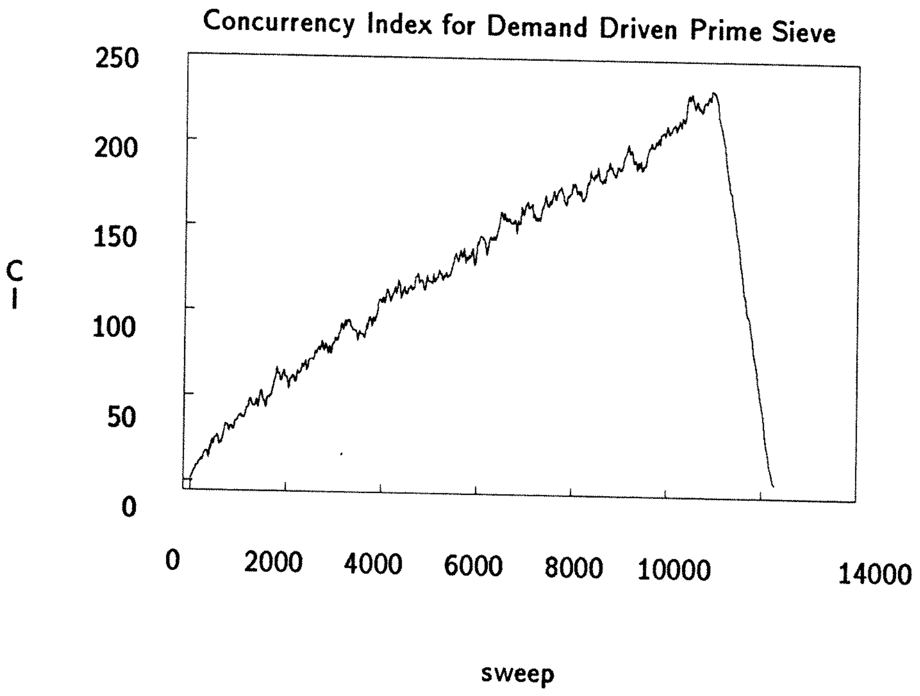


Figure 4.7: Concurrency Index for Demand Driven Prime Sieve ($N = \infty$)

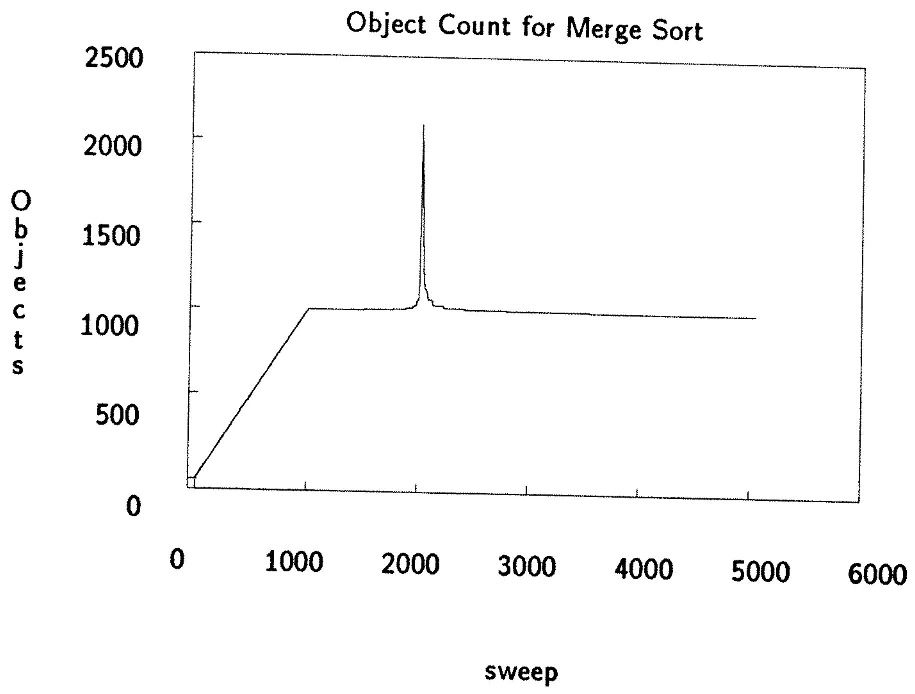


Figure 4.8: Object Count for Merge Sort ($N = \infty$)

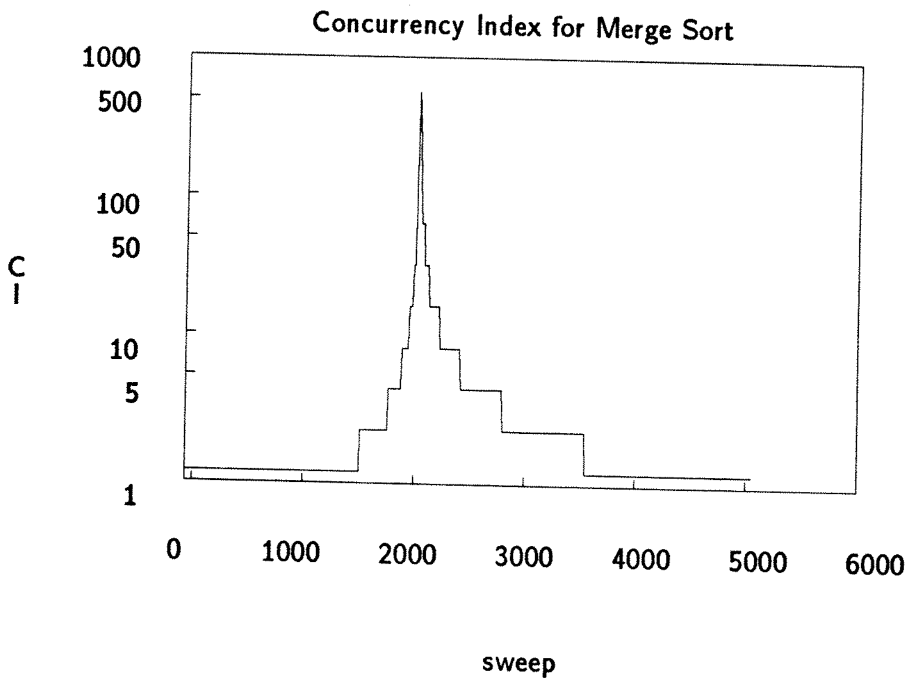


Figure 4.9: Concurrency Index for Merge Sort ($N = \infty$)

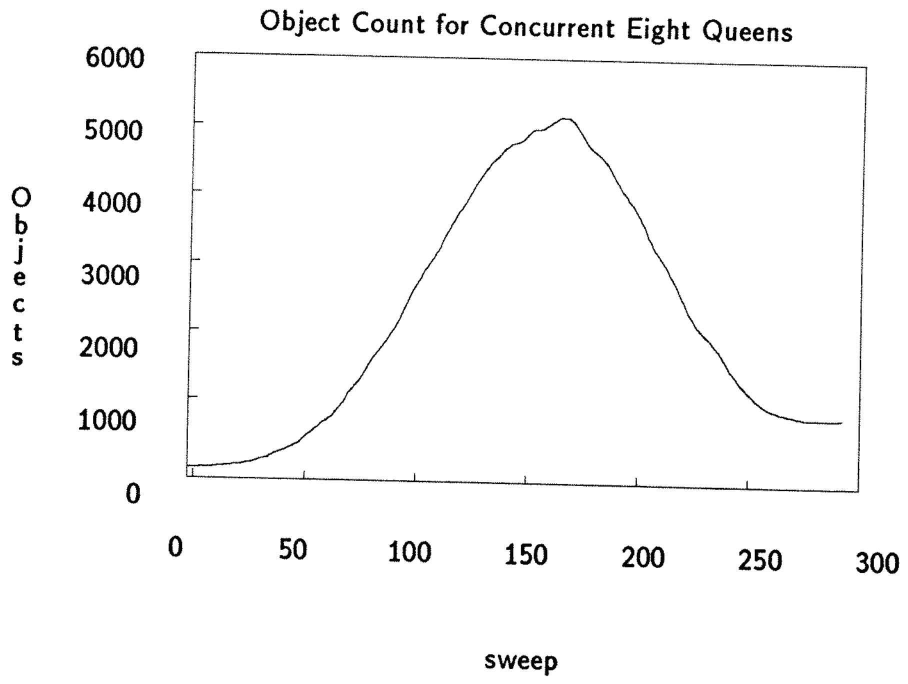


Figure 4.10: Object Count for Concurrent Eight Queens ($N = \infty$)

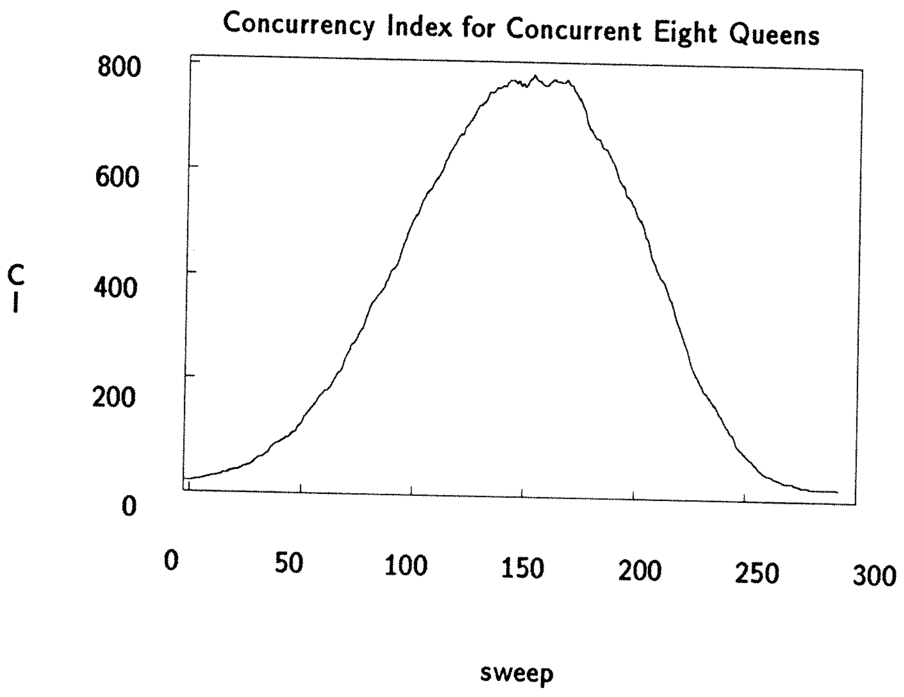


Figure 4.11: Concurrency Index for Concurrent Eight Queens ($N = \infty$)

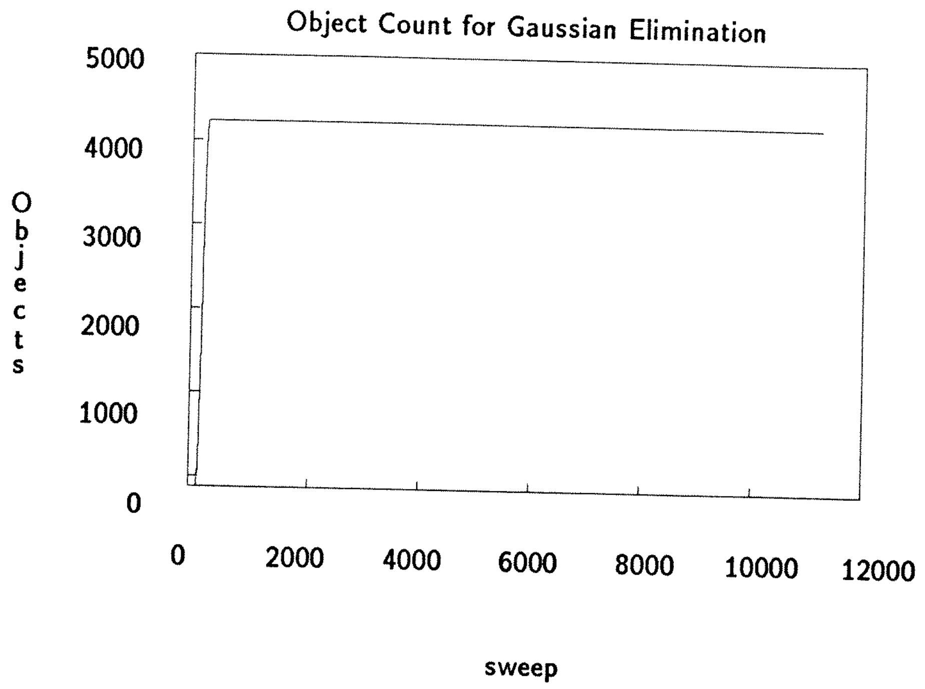


Figure 4.12: Object Count for Gaussian Elimination 64,64 ($N = \infty$)

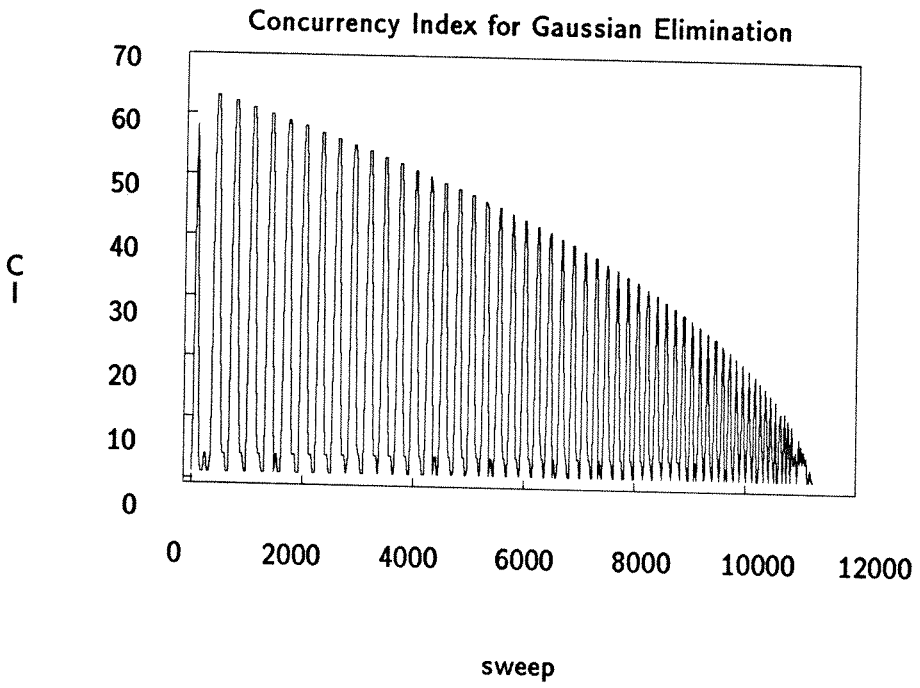


Figure 4.13: Concurrency Index for Gaussian Elimination 64,64 ($N = \infty$)

4.2 Program Behavior for Fixed Size Ensembles

The simulation results of the six test programs under the assumptions of the soft computer represent a best case for the execution of the suite of sample programs. All concurrency extant in the programs is exploited in execution. Although no proof exists, the conjecture is that programs which behave poorly under the soft computer conditions will perform poorly under any achievable set of real conditions. Assuming that the six test programs represent programs that perform well under the soft conditions, the next step is to challenge the soft assumptions. The first assumption to challenge is the amorphous computing sites of the soft computer. Clearly a physical ensemble machine will have a fixed number of processing nodes.

The introduction of a fixed processing surface mandates that an assignment procedure be used to assign objects to nodes. This requirement did not exist for the soft computer because the mapping is always one-to-one. The two factors to consider are speedup and node utilization. For the case where the number of nodes is greater than the number of objects, the best strategy, ignoring utilization and locality, is to place one object per node. For the case where the the number of objects is greater than the number of nodes, the best strategy for speedup is not clear unless the semantics of the program are known.

Two possible global strategies that do not use program semantics are to optimize object and message loads. For optimizing object load, the number of objects on each node is totalled, and the node with the lowest count is selected. Optimization of object load attempts to equalize the amount of storage used over all of the nodes. For optimizing message load, the number of messages not yet processed by each node is counted and the node with the lowest count is selected. Optimization of message load attempts to equalize the workload over all the nodes, because each node can process only process one message per sweep. A myriad of other optimizations are possible, including using a weighted average of message and object loads.

The effects of limiting the number of nodes to 64 and using the object load placement strategy are shown in Figures 4.14 through 4.17 for the merge sort and eight queens program respectively. Both programs exhibit clipping; that is, the concurrency index limits at the number of nodes. This clipping is the desired response, because it indicates that all of the nodes dedicated to the program are in use.

Although the peak concurrency index for merge sort is over 500 in Figure 4.9, the time to completion is not appreciably lengthened by limiting the number of nodes to 64. Peak concurrency is therefore not an accurate indicator for the amount of concurrency that can be exploited. The area under the concurrency index curve determines the amount of work to be done, and also the portion of the workload that can be performed concurrently. Because the merge sort program only exhibits high levels of concurrency for a short interval of sweeps, reducing the number of nodes does not appreciably impact the time to completion. In contrast, the eight queens program exhibits high levels of concurrency over many sweeps. By limiting the number of nodes to 64, the time to completion is increased by a factor of 6, indicating that the average level of concurrency is approximately 384 active objects per sweep.

The area under the concurrency index curve is not an absolute indicator, because of the possible non-determinism in message arrival. For some programs, the number of objects created and number of messages sent may vary if messages are processed in the less than the optimal order. This order may be affected by the assignment of objects to nodes.

Figures 4.18 through 4.23 are graphs of the sweep counts versus number of nodes for the six test programs. These graphs define the achievable *speedup* for a fixed number of nodes and placement strategy. The overall shape of the speedup curves can be calculated using Ware's formula [40]:

$$\text{sweep count} = m(k + \tfrac{1}{N}(1 - k))$$

where m is the total number of messages delivered, k is the fraction of the computation that must be performed sequentially because of message dependencies, and N is the number of nodes. The value of k can be calculated from the sweep counts for the soft computer experiments. For these experiments, every opportunity for concurrency is exploited, and the value of N is effectively infinity. With N equal to infinity, the sweep count is limited to the longest chain of messages that must be delivered in sequence.

Speedup graphs such as Figure 4.22 clearly show that the sweep count for a fixed size ensemble asymptotically approaches the sweep count for the soft computer. That latter sweep count is drawn as a dashed line across each of the graphs. For example, the sequential fraction of the eight queens computation is quite small ($k = 0.00293$), and the speedup is near-perfect until the node count reaches 256. For node counts between 1 and 256, the speedup increases close to a factor of 4 each time the number of nodes is quadrupled. Because the speedup graphs are log-log plots, the line between the sweep count for 1 node and 256 nodes is nearly a straight line. For node counts beyond 256, the sequential part of the computation begin to dominate Ware's formula, i.e., $1/N$ becomes comparable to k, and the return on applying additional nodes to the computation begins to diminish. For 4096 nodes, there is virtually no benefit for applying additional nodes to the computation.

Each speedup curve is compared with the sweep count of the corresponding soft experiment. Three placement strategies were used: optimal message load, optimal object load, and random placement. Random placement was implemented using the non-linear additive feedback pseudo-random number generator found in the the UNIX$^{\text{TM}}$ library[39]. The most striking aspect of the six curves is the performance of random placement compared to the two placement strategies that required global, instantaneous data from all of the other nodes in the ensemble.

The three divide and conquer programs of perfect numbers, merge sort and eight queens were quite insensitive to placement strategy. The speedup curves of Figures 4.18, 4.21, and 4.22 indicate near perfect halving of the number of sweeps to completion for each doubling of the number of nodes. The two pipelined sieves worked best under object load balancing and worst under message load balancing. Intuitively, message load should be a good indicator, because it defines the workload for each node, i.e., the number of messages that each node has to process. However, balancing the message load performs poorly for the prime sieves because initially the first few stages of the sieve are able to keep up with the influx of messages. The objects comprising the first few stages of the sieve are thus assigned to the same node. Placing the first few stages on the same node is a poor placement strategy because the stages cannot process messages concurrently.

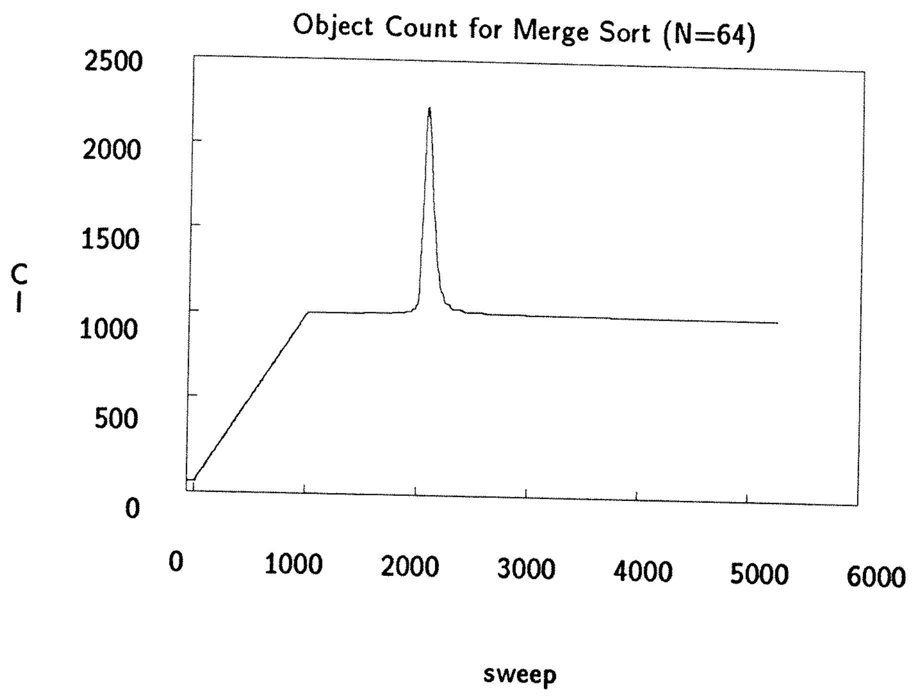


Figure 4.14: Object Count for Merge Sort ($N = 64$)

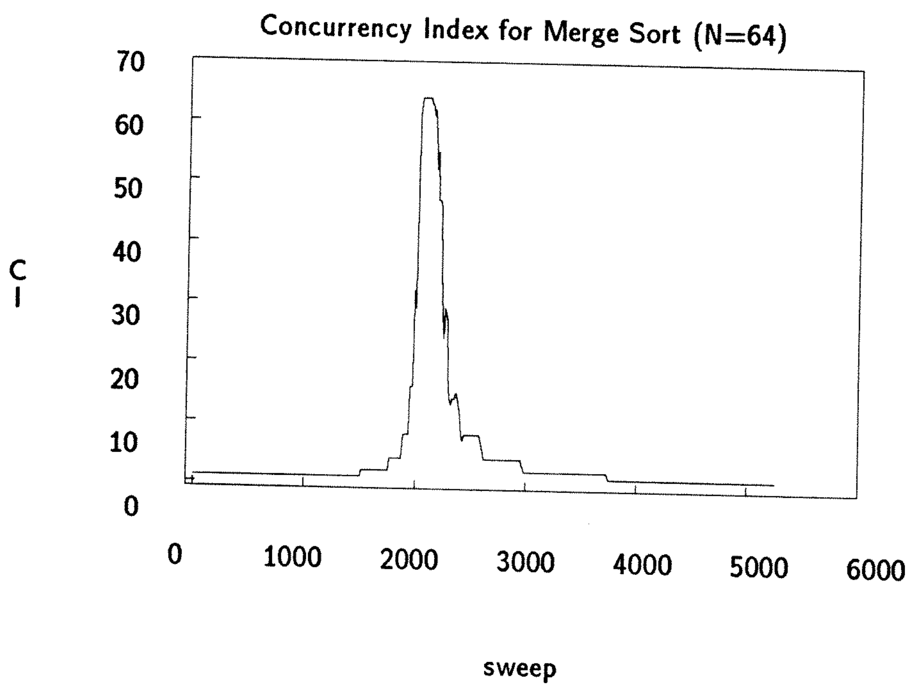


Figure 4.15: Concurrency Index for Merge Sort ($N = 64$)

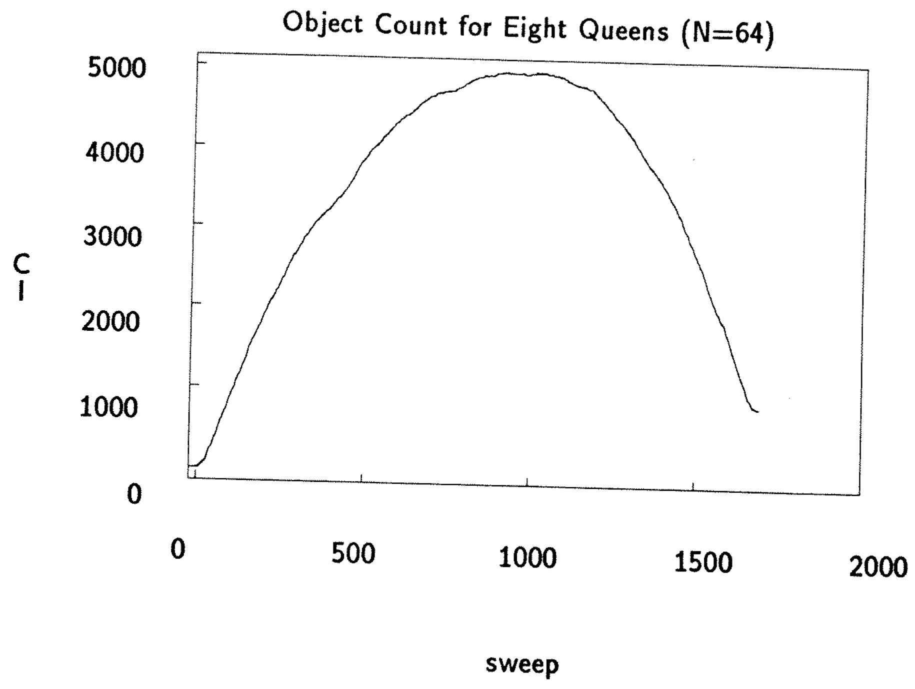


Figure 4.16: Object Count for Eight Queens ($N = 64$)

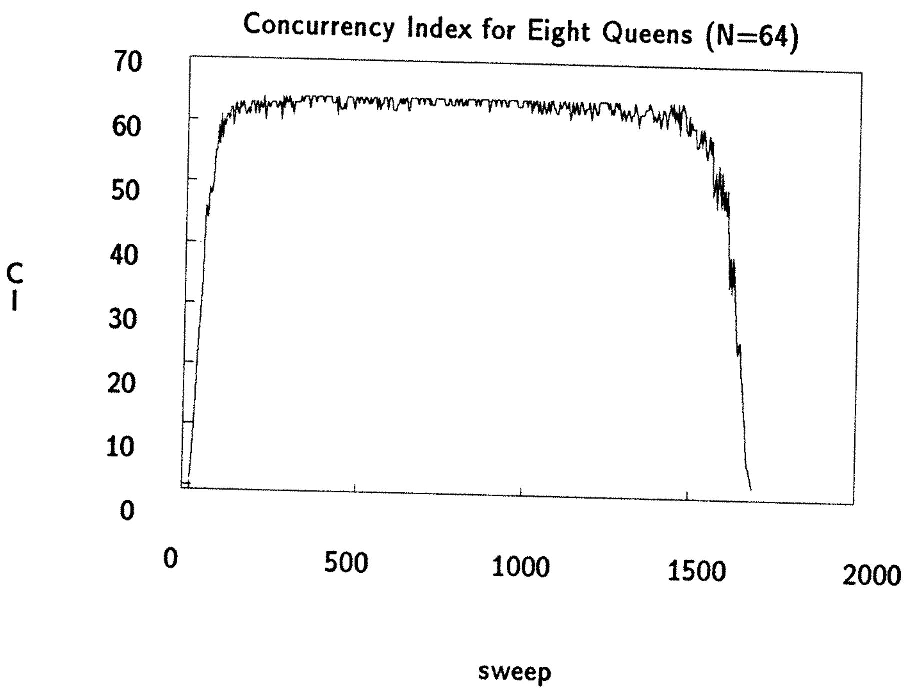


Figure 4.17: Concurrency Index for Eight Queens ($N = 64$)

These objects form a bottleneck and restrict the achievable concurrency in the later stages of the sieve. The bottleneck is especially pronounced for the wheel driven sieve. This program was previously noted as having a potential bottleneck between the output of the number generator and the input to the sieve, if the two sets of objects were not properly speed matched.

For the Gaussian elimination program, the performance of the object load and random placement were not substantially different, but object load was appreciably worse, over a factor of two slower for 16 nodes. The poor performance of the message load strategy is difficult to evaluate without examining in detail the assignment of the objects of the matrix to the nodes.

In conclusion, neither the message load or object load balancing strategies proved to be universally better for the six programs. In contrast, the random placement strategy was shown to be, in general, a good strategy for load balancing. Furthermore, there is no overhead for random placement, while message and object load balancing require global tables that must be instantaneously updated. The effectiveness of random placement can be understood by considering the two cases of many more active objects than processing nodes and many more processing nodes that active objects. For the case in which the number of active objects is much smaller than the number of nodes, the probability that two or more objects are assigned to the same node is small. For the case in which the number of active objects is much greater than the number of nodes, then by the weak law of large numbers, the probability that a node will be assigned an active object increases with the number of assignments made.

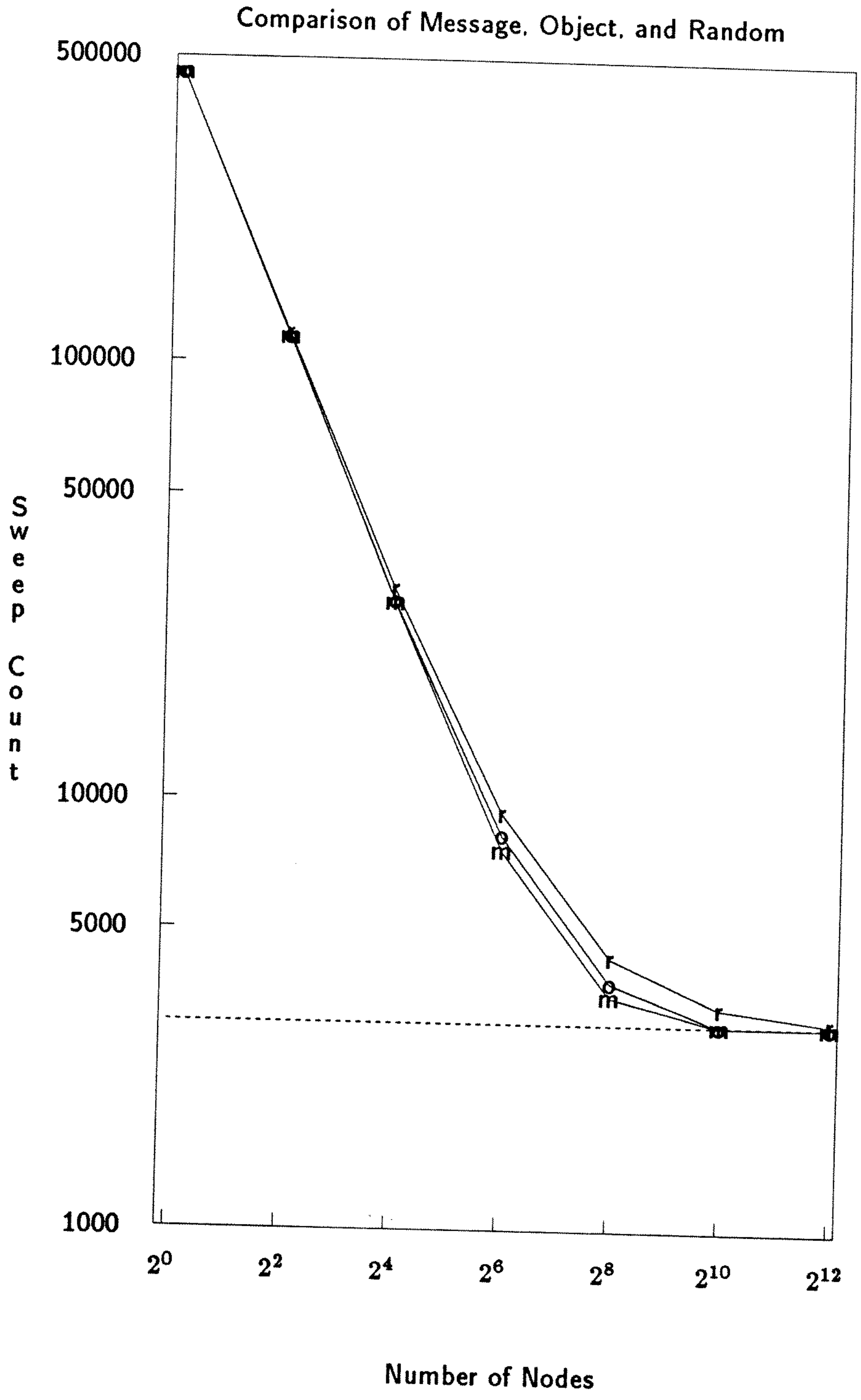


Figure 4.18: Speedup Comparison for Perfect Numbers

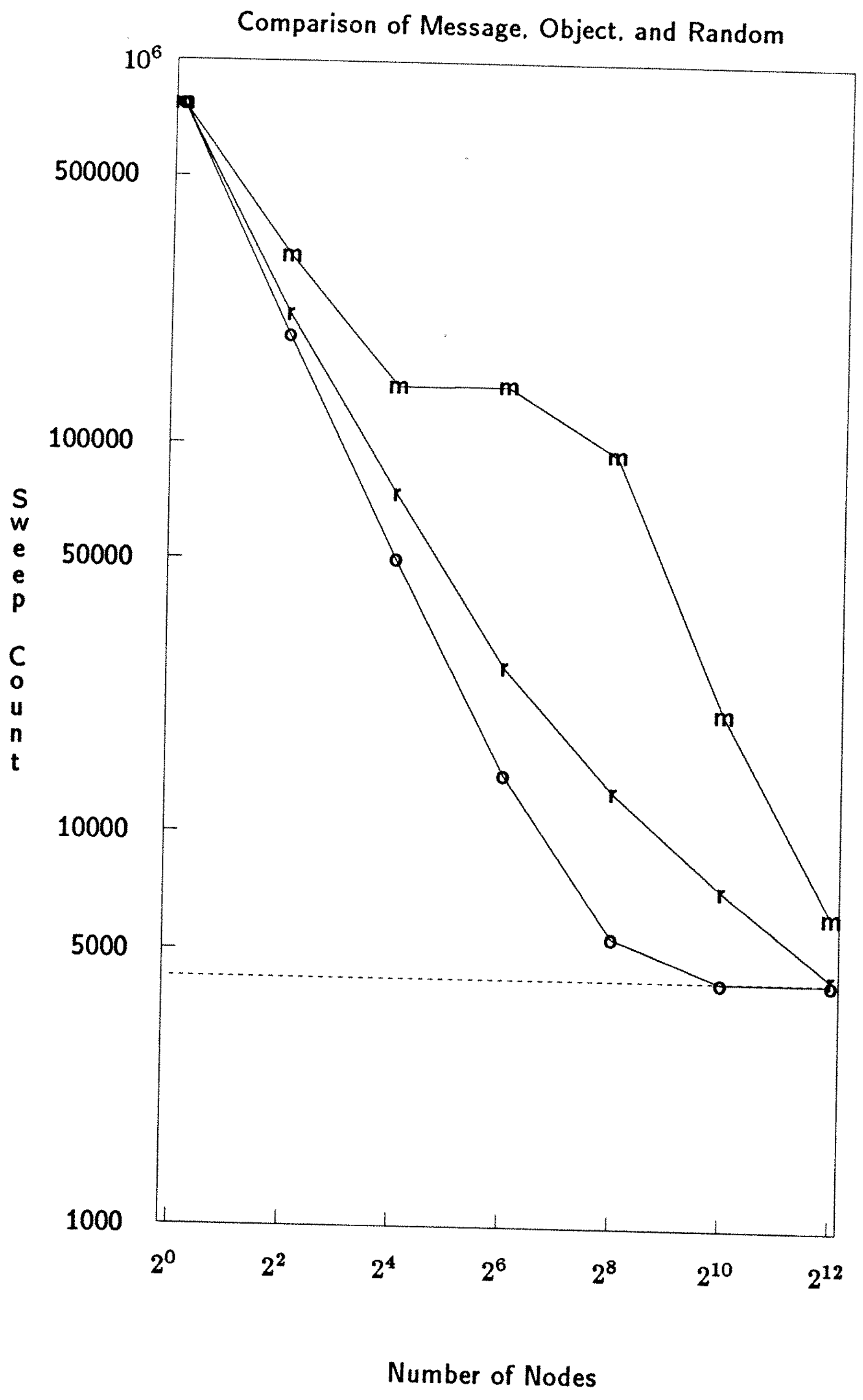


Figure 4.19: Speedup Comparison for Wheel Driven Prime Sieve

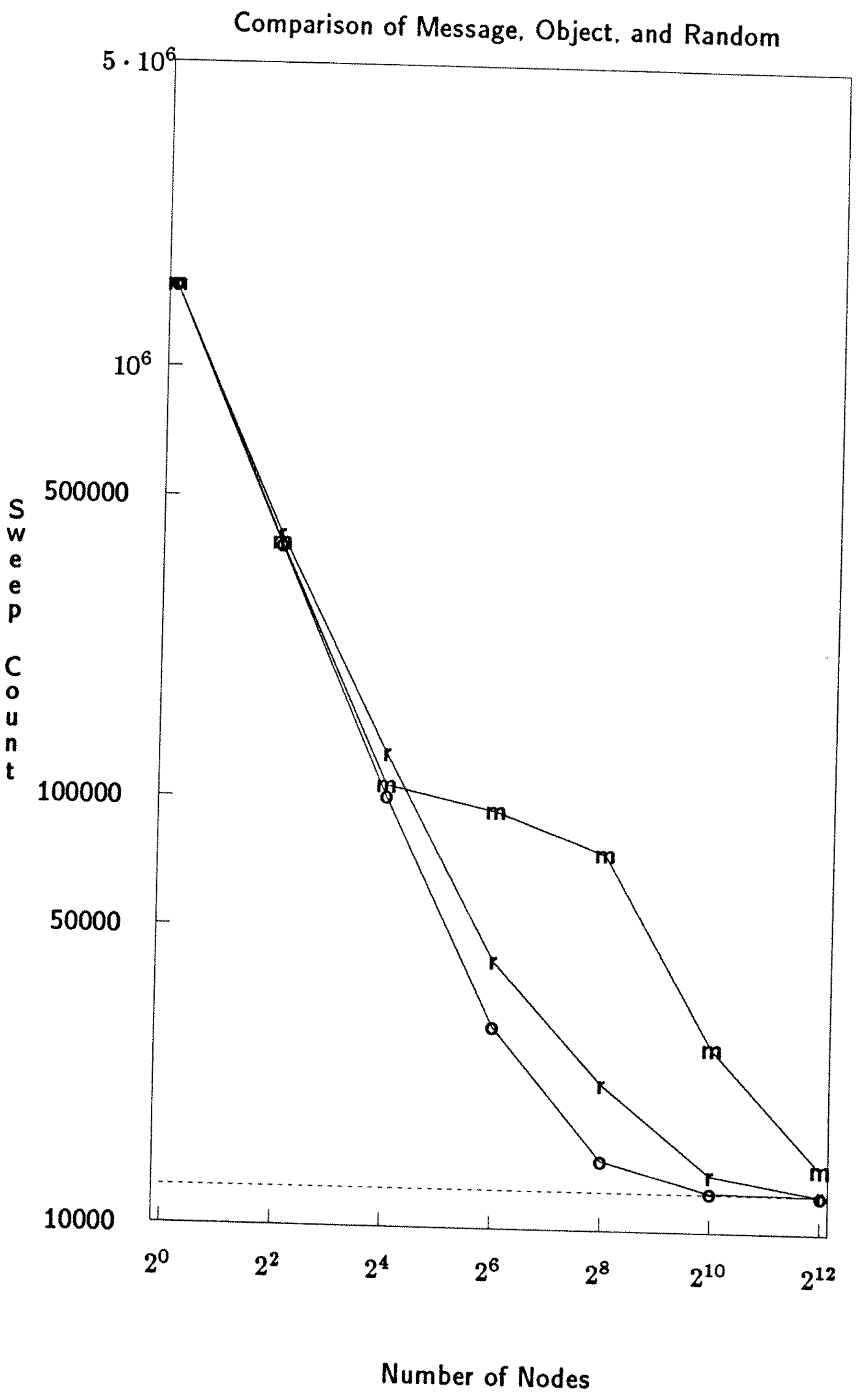


Figure 4.20: Speedup Comparison for Demand Driven Prime Sieve

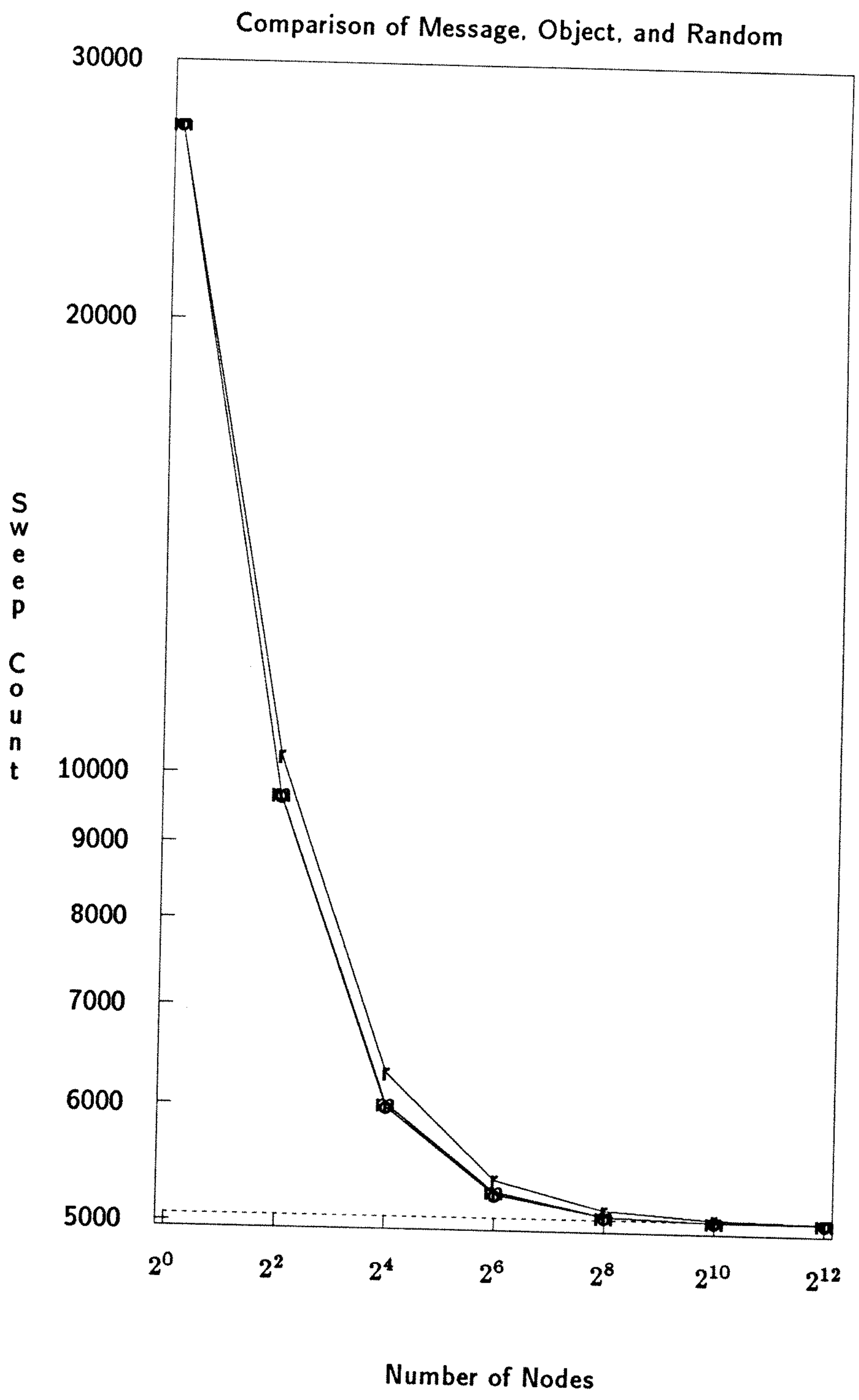


Figure 4.21: Speedup Comparison for Merge Sort

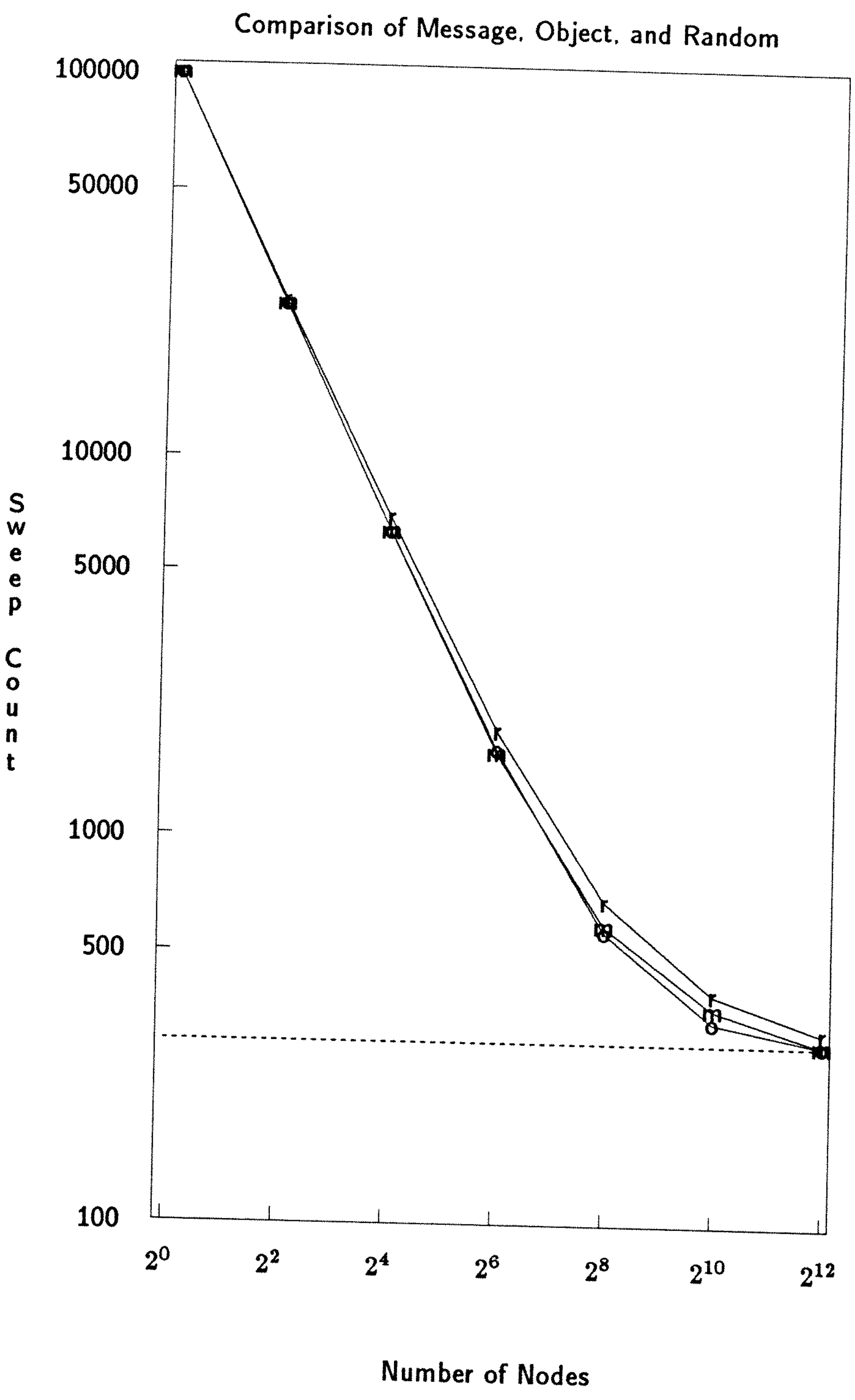


Figure 4.22: Speedup Comparison for Eight Queens

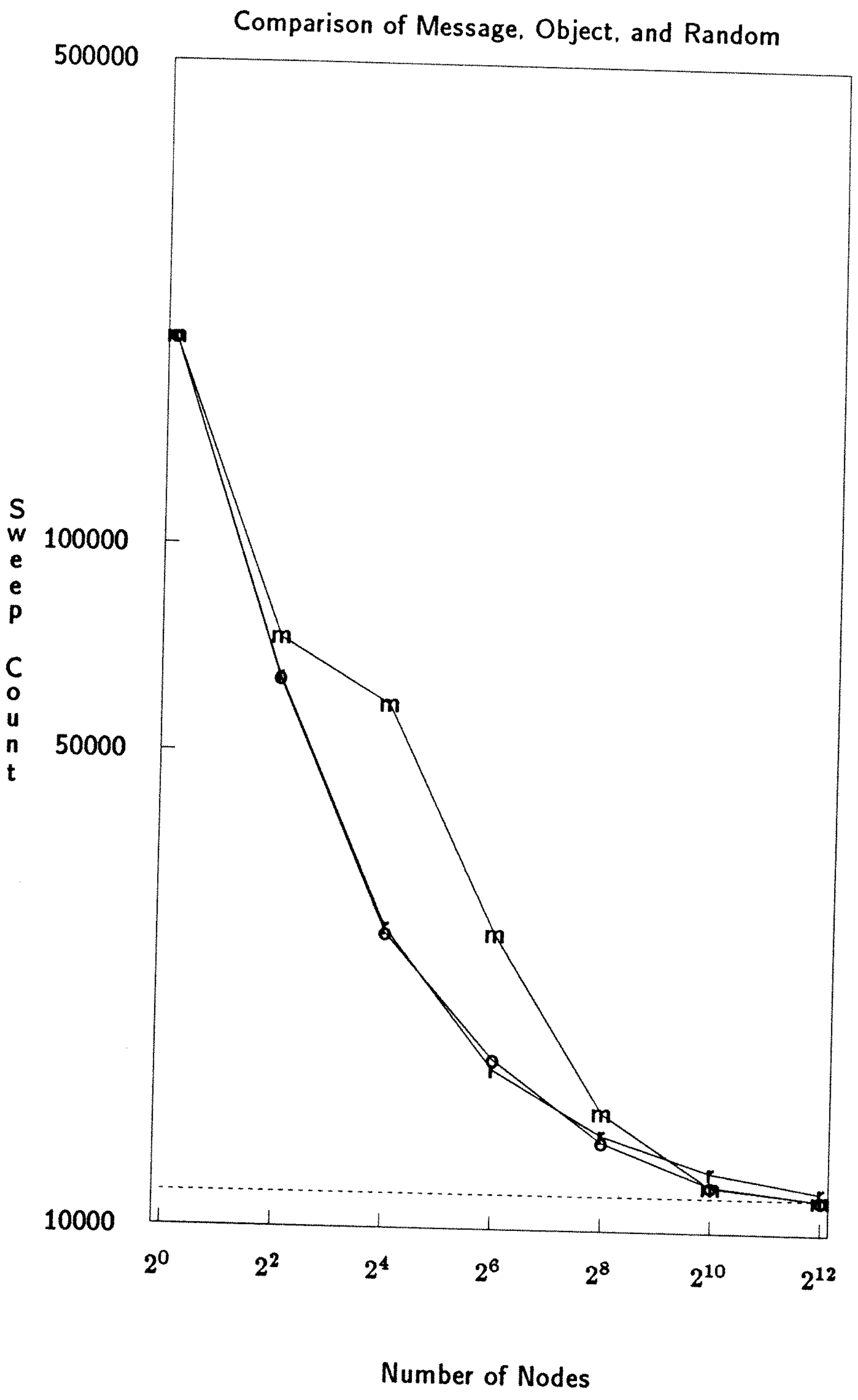


Figure 4.23: Speedup Comparison for Gaussian Elimination

4.3 Program Behavior for Limited Message Throughput

The two important properties of the soft computer are its capability to dynamically change the number of computing sites and its capability to deliver all messages generated during a sweep before the beginning of the next sweep. The negative effects of reducing the number of nodes was shown to be minimal; near-perfect node utilization was possible using the simple assignment strategy of random placement. The second property to be considered is message throughput. For the soft computer, the message-passing network must be capable of an infinite amount of bandwidth between the computing sites, and must also be capable of delivering all messages in the interval of a single sweep. For fixed sized ensembles where the number of active objects is greater than the number of nodes, the effects of message latency are negated, because each node cannot process all of the outstanding messages in one sweep. The messages that are not processed within a sweep are held over to the next sweep. These held over messages remain in the input queue and ensure that the node has a message to process for the subsequent sweep.

For the case of more active objects than nodes, the input queues and output queues for each of the nodes will tend to be non-empty. If the input queues are non-empty, then the node has insufficient message throughput to process the messages as they are delivered. This configuration is desirable because if the input queues are empty, but the output queues are non-empty, then the message-passing network becomes a bottleneck. The nodes then become idle, and the speedup of a program is reduced irrespective of the number of nodes. The message throughput required of the message routing network is determined by the semantics of the object programs.

To measure the impact of limiting message throughput for fixed size ensembles, a set of simulations were performed using the random placement strategy. For each simulation, the number of messages delivered from each node per sweep was limited to 1,2,4, or 8. Nodes are assumed to have limited message bandwidth into the message routing network, but unlimited message bandwidth out of the network. This asymmetry is unrealistic because it allows every node to send a message to a single node in a single sweep, but one node sending a message to k other nodes requires k sweeps. The bias is offset by the use of random placement.

Figures 4.24 through 4.25 contain the speedup curves for the six test programs. All six programs exhibited degradation when the message delivery rate was limited to one message per node. This degradation is to be expected, because to increase the concurrency index requires two sweeps instead of one. For concurrency to be introduced into a computation, an object must send two or more messages for each message processed by the object. Because the concurrency can only grow at a linear rate, the two prime sieves were the least sensitive to restricted message throughput. The pipelined behavior of these two programs permits most of the concurrency to be exploited with only one message sent per node. The Gaussian elimination program also did not show much performance loss. The concurrency in the Gaussian elimination program is only piecewise and not very large, i.e., a maximum concurrency index of 64. Thus, restricting the message throughput does not greatly change program behavior.

The three divide and conquer programs, however, were significantly affected by restricted message throughput. This sensitivity to message throughput can be understood in part by examining the individual programs. For the perfect numbers program, the divisor object requires that three new objects, an adder object and two additional divisor objects, and two messages be sent in response to receiving a single test number. The three new objects and two messages result in five messages sent in response to receiving a single message. Thus, for message delivery rates less than five messages per sweep, the messages generated during a single sweep must be sent in two or more sweeps.

The merge sort program has a similar requirement inside of the msort definition in which two messages are sent and three new objects are created. The eight queens program which is the least sensitive of the three programs requires only a maximum of two messages sent and a single new object created. This case occurs inside of the queen object definition.

The six speedup curves indicate that sensitivity to message throughput increases as the number of nodes increase. Furthermore, for the perfect numbers and merge sort programs, the speedup curves suggest that the speedup becomes asymptotic for each of the four delivery rates. This result shows that the semantics of the object programs require minimum levels of message throughput on a per node basis instead of a per network basis. Although the number of messages delivered per sweep increased proportionately by increasing the number of nodes, the two programs did not benefit from the increase. In conclusion, both number of nodes and the message throughput of the message-passing network become the limiting factors for speedup.

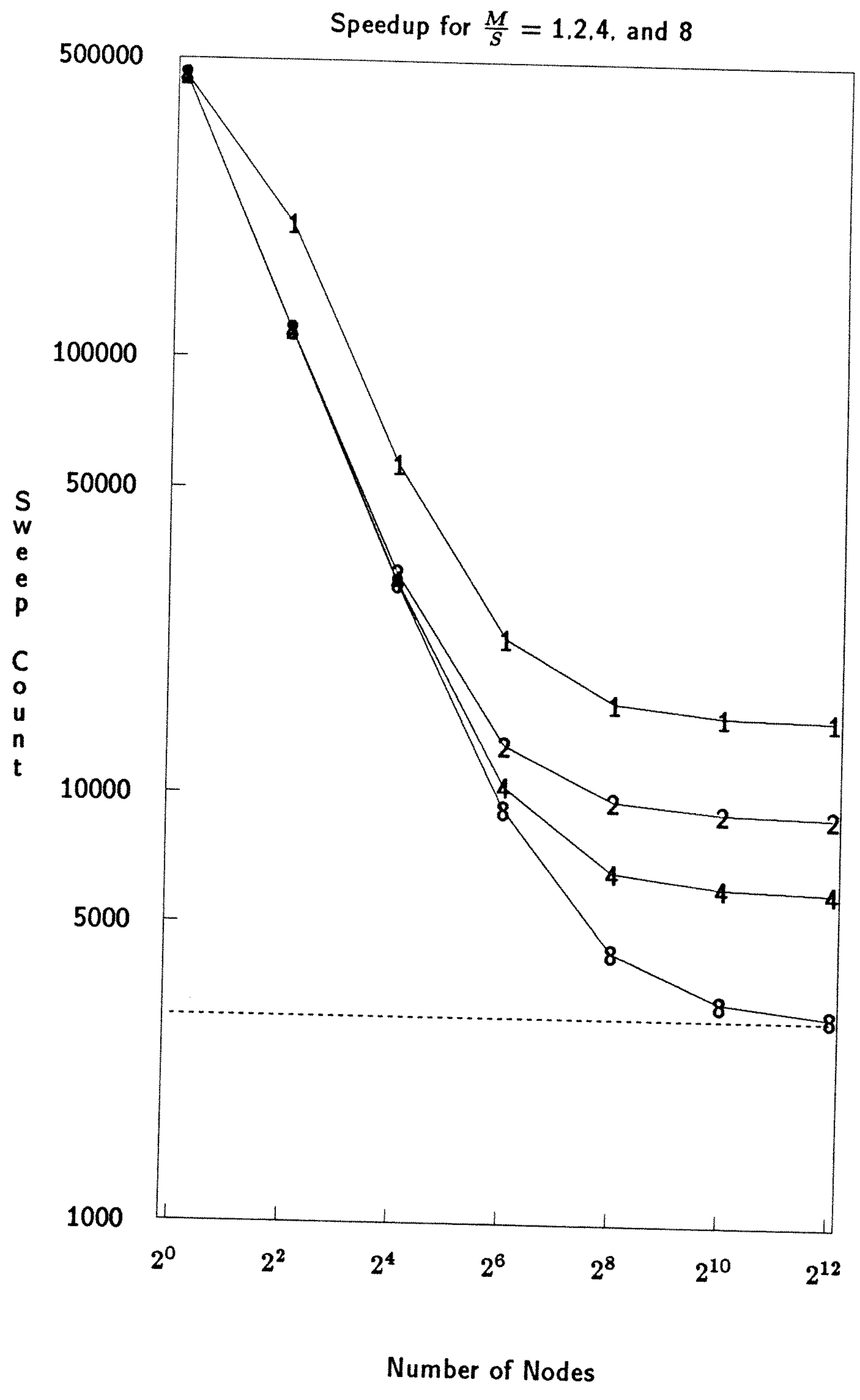


Figure 4.24: Limited Message Throughput for Perfect Numbers

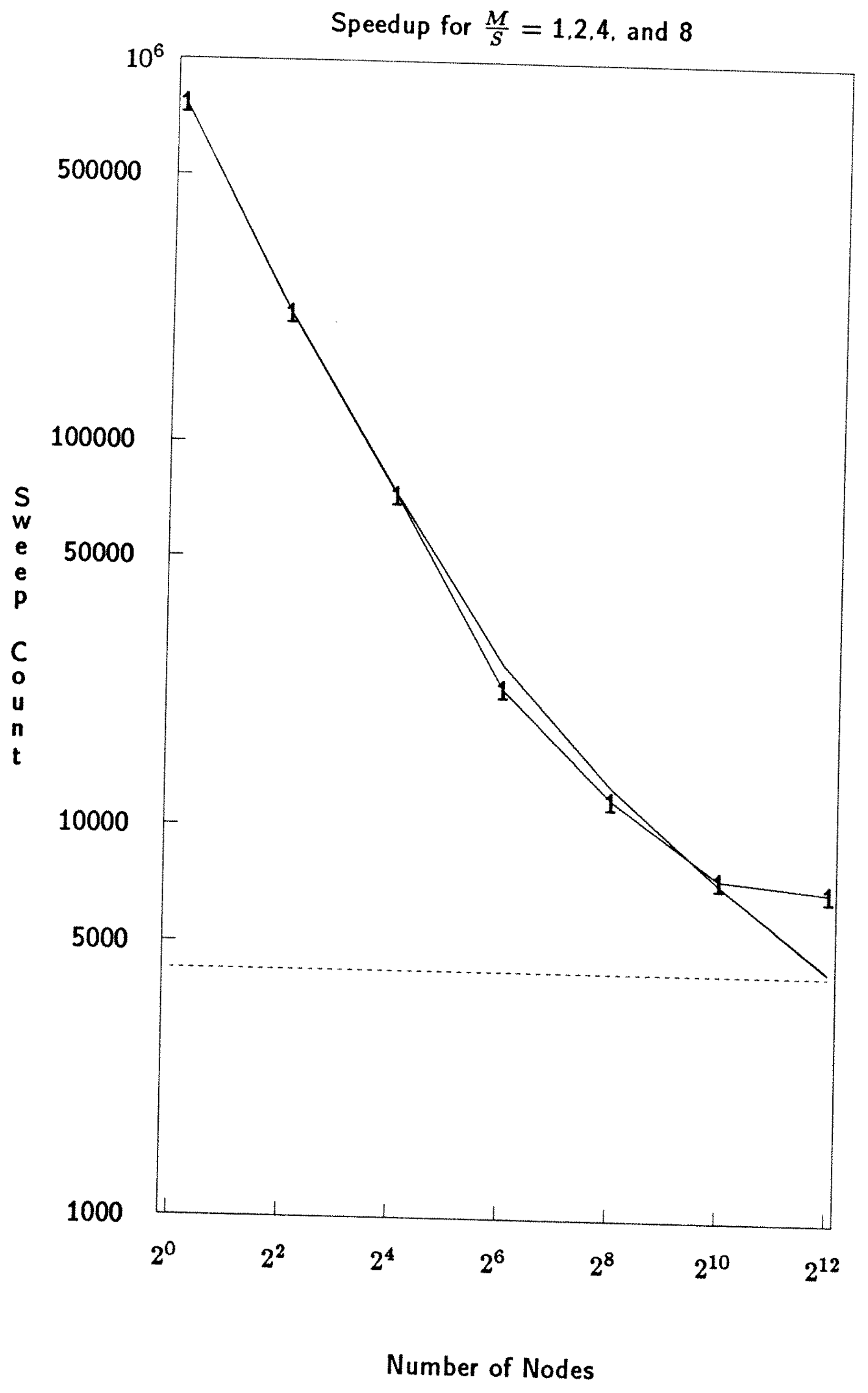


Figure 4.25: Limited Message Throughput for Wheel Driven Prime Sieve

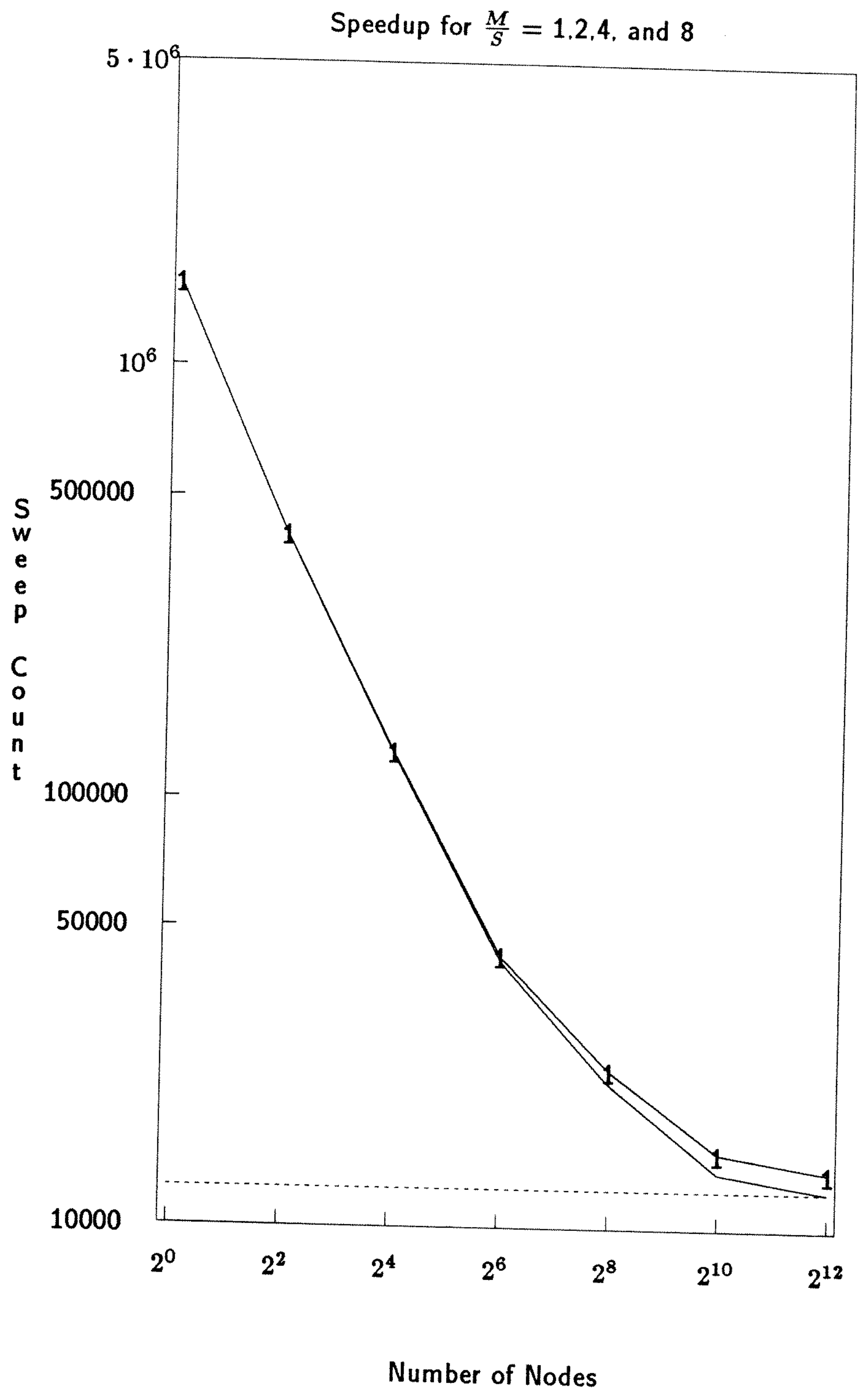


Figure 4.26: Limited Message Throughput for Demand Driven Prime Sieve

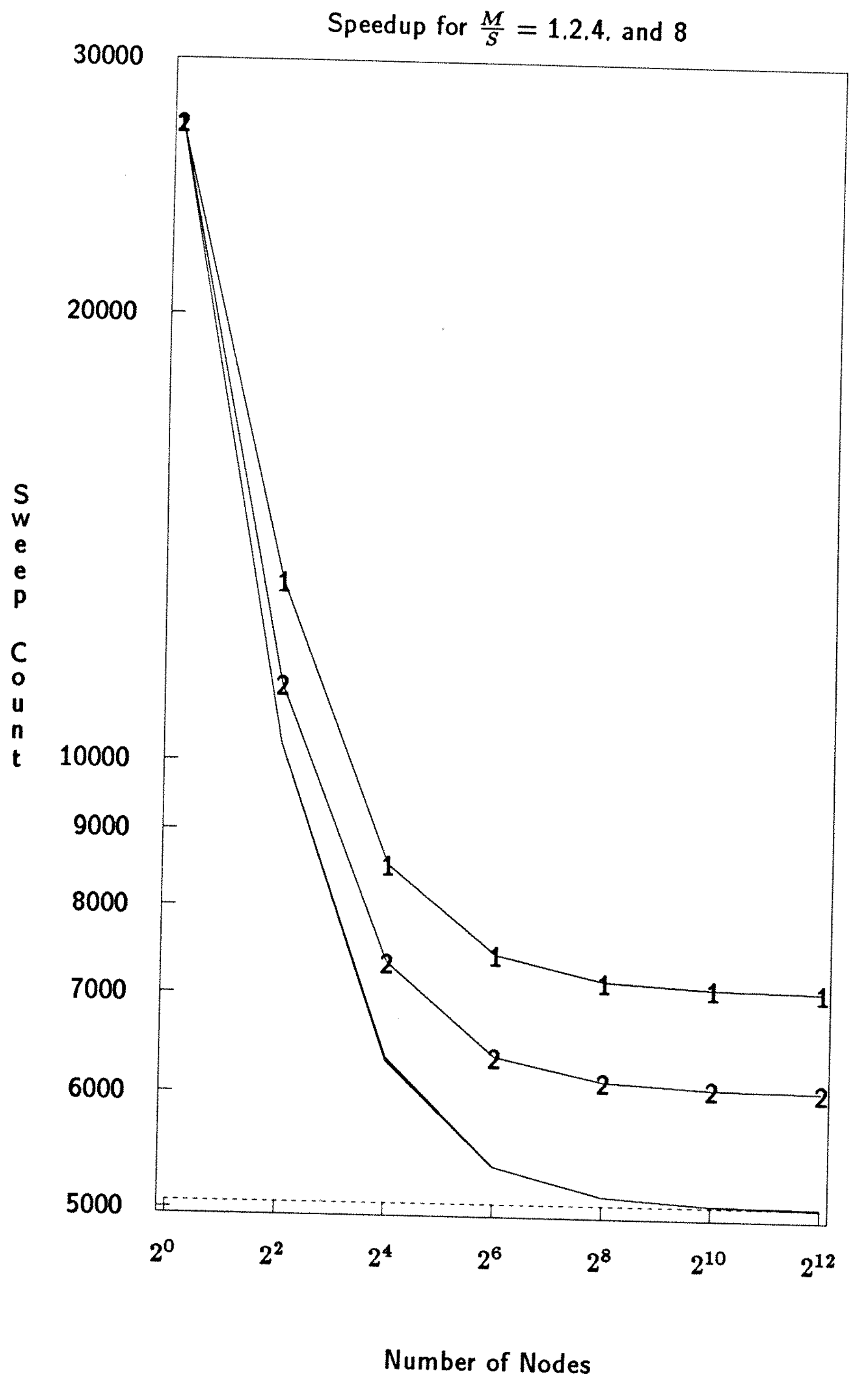


Figure 4.27: Limited Message Throughput for Merge Sort

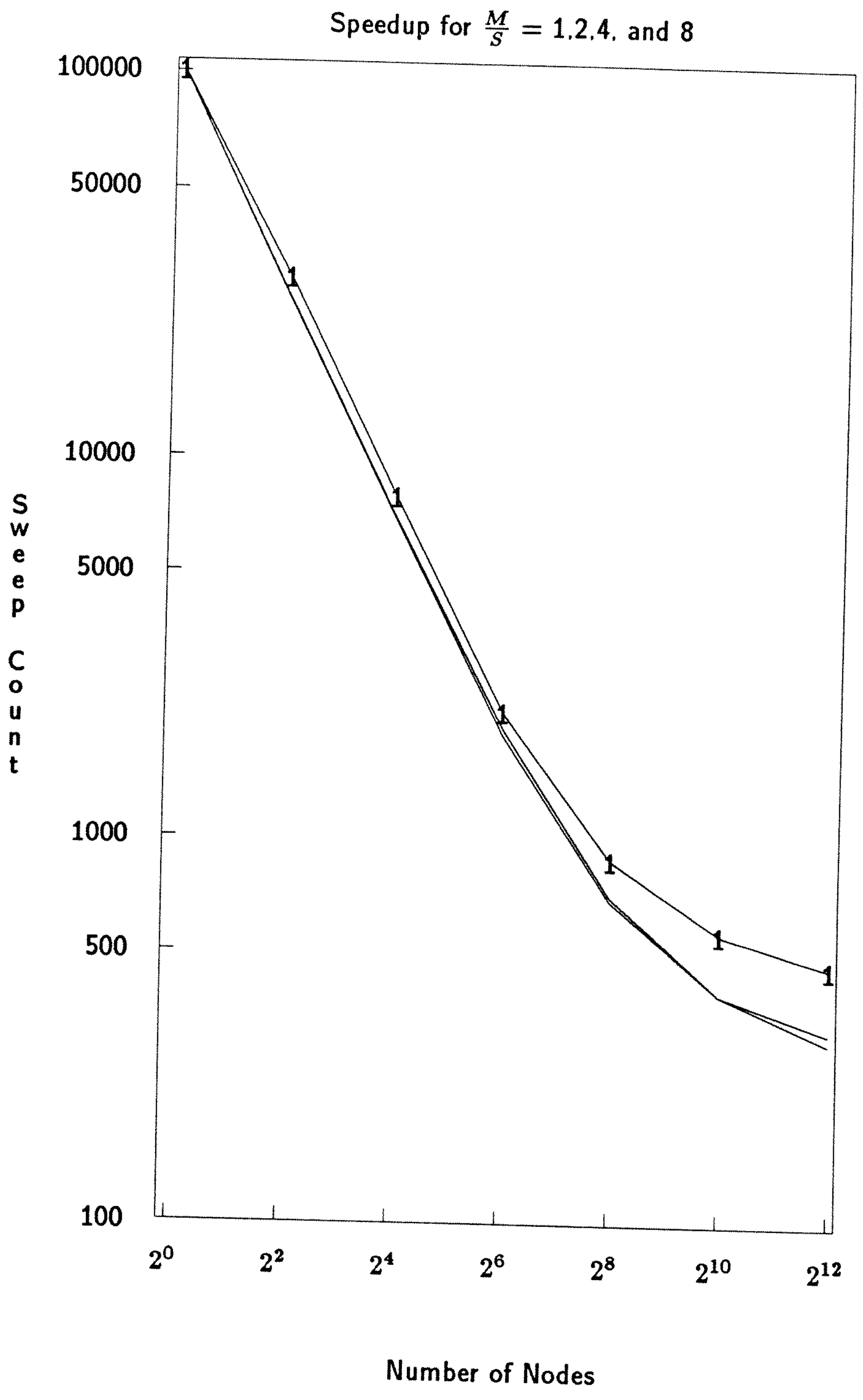


Figure 4.28: Limited Message Throughput for Eight Queens

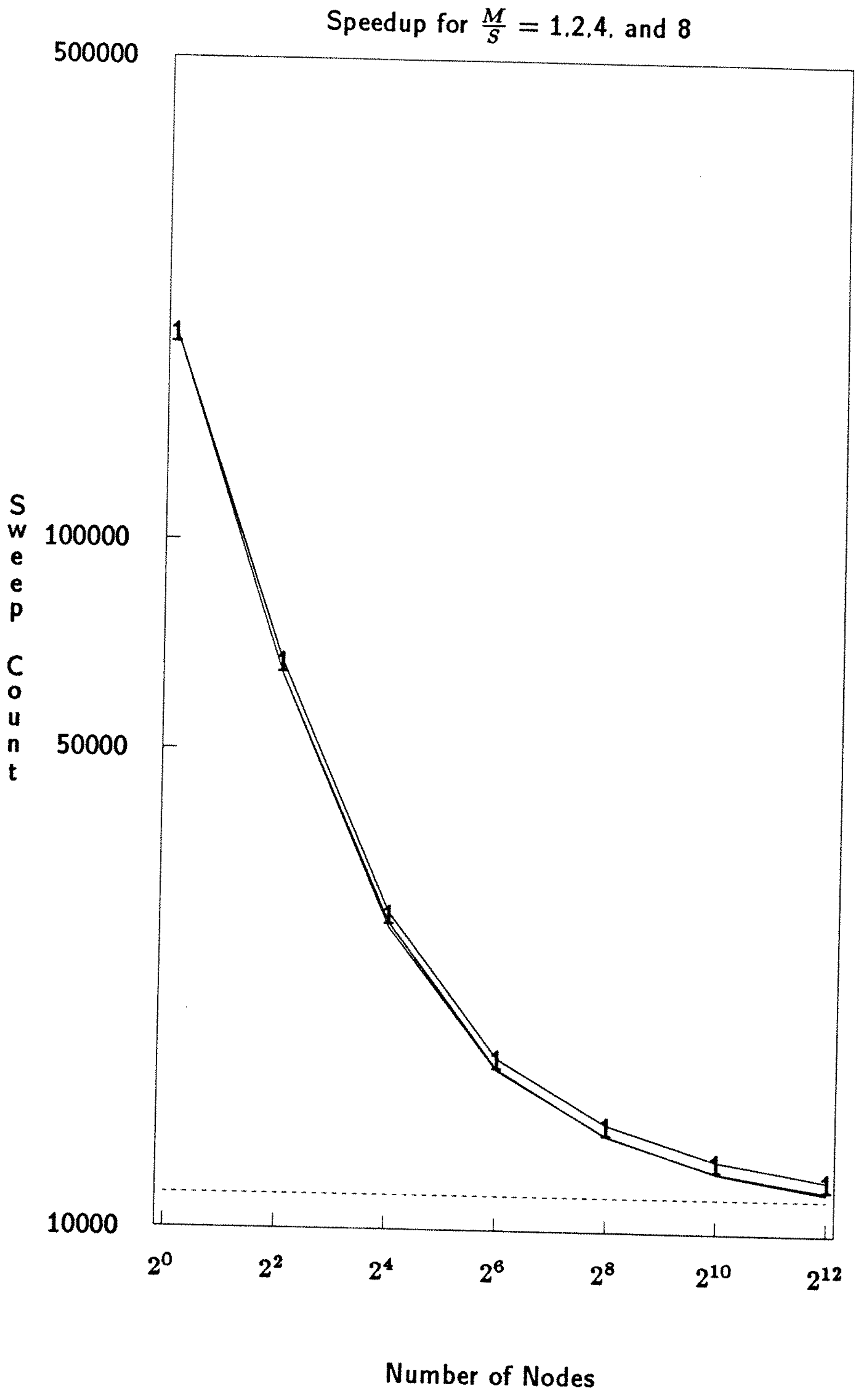


Figure 4.29: Limited Message Throughput for Gaussian Elimination

4.4 Locality

There are two methods for increasing the available message throughput of a message-passing network: increase the bandwidth of the network and introduce locality. Increasing the bandwidth of the network requires that the message transfer time be decreased, or the width of the paths that messages travel over be increased. Either approach to increasing the bandwidth involves a physical cost and is subject to a myriad of constraints, such as interconnection topology, packaging, power dissipation, etc. The assumption is that the architect will always build the highest performance network for a given cost and set of constraints. Introducing locality, however, is performed at the programmer, compiler and runtime system levels. The impetus for locality is straightforward; by reducing the physical distance that messages must travel, more messages may be resident in the network at a given time. The available message throughput is therefore used more efficiently.

General locality, i.e., locality that is topology independent, can be used by programmer in many ways. Possibly the simplest example is the sending of messages to self. A more elaborate technique is to conserve the use of reference values. For example, if reference values are never sent as the contents of a message, then reference values for new objects must be contained exclusively by the object that created the new object. This restriction ensures that the locality of reference is always contained to a fixed set of neighboring objects. In general, sending a reference value as part of the contents of a message destroys locality.

Locality introduced by the compiler and runtime system is controlled by the assignment of objects to nodes. One definition for a cost criterion of the distance between two nodes is to consider objects as placed on two-dimensional processing surface. Each node has an (x, y) coordinate pair and the distance between nodes n_1 and n_2 is simply:

$$|x_2 - x_1| + |y_2 - y_1|$$

This distance criterion calculates the *Manhattan* distance between the two nodes. This distance is the minimal number of steps needed to travel from n_1 to n_2 on a two-dimensional mesh.

For a k by k mesh, the average distance over all possible pairs of nodes is calculated by the formula:

$$\bar{d}_r = \frac{2}{3}(k - \frac{1}{k})$$

From this formula the expected distance for messages sent between two communicating objects can be calculated. The actual average distance can also be calculated by summing the Manhattan distances for all messages delivered during a sweep and dividing this number by the number of messages delivered. Figures 4.30 through 4.41 show the number of messages delivered per sweep and the average distance for all of the messages delivered during each sweep for a 256 node ($k = 16$) ensemble using the random placement strategy. The calculated average distance from the formula for a 16 by 16 mesh is 10.6, and the six average distance graphs display only small deviations from this calculated value. This observation is confirmed by calculating the mean, median, and standard deviations for each of the simulations averaged over all of the sweeps. These calculations are shown in

Program	mean	median	std. dev.
Eight Queens	9.2	9.3	1.76
Gaussian Elimination	10.7	10.7	2.8
Merge Sort	10.6	10.7	4.4
Perfect Numbers	10.6	10.6	0.9
Demand Driven Primes	10.5	10.5	0.9
Wheel Driven Primes	10.5	10.5	0.9

Table 4.1: Mean, Median and Standard Deviations

Table 4.1. Both average and median values are close to the calculated average distance from the formula except, for the eight queens program. The eight queens program has a smaller than expected average distance value because a large portion of the send statements are messages sent to self. The programs with the largest deviation from the expected distance are Gaussian elimination and merge sort. These two programs have the lowest levels of concurrency of the six test programs. Because the concurrency levels are small, the number of messages processed in a sweep is small and thus the sample set for calculating the average distance per sweep is also small. These small sample sets result in a large variance on a per sweep basis, but the effects of averaging the distances over all of the sweeps shows the average values to be close to the expected value of the formula. This effect can clearly be seen in the message throughput and average proximity graphs for merge sort (Figures 4.36 and 4.37). The average proximity shows large amounts of deviation until the concurrency sharply rises; the proximity then quickly stabilizes. Once the concurrency decreases sharply, the proximity then again shows large deviations.

The six message throughput graphs show the total number of messages that must be delivered per sweep to prevent the computations from becoming sensitive to message throughput, i.e., to maintain the illusion of infinite bandwidth. For the two prime sieves and Gaussian elimination, the number of messages delivered per sweep is less than half the number of nodes. These three programs were also the least sensitive to message throughput. The three divide and conquer programs required message delivery rates equal to or greater than the number of nodes. These three programs were the most sensitive to message throughput.

Assuming that the cost of sending a message is proportional to the distance that the message must travel, then by the average distance formula, the cost is the same for all messages. By scaling the message throughput graphs by $\bar{d}_r$, the graphs demonstrate the amount of message resources that have to be available per sweep to keep the computation insensitive to message throughput. The message throughput graphs are a function of program behavior and cannot be changed without modifying the source program or altering the size of the ensemble.

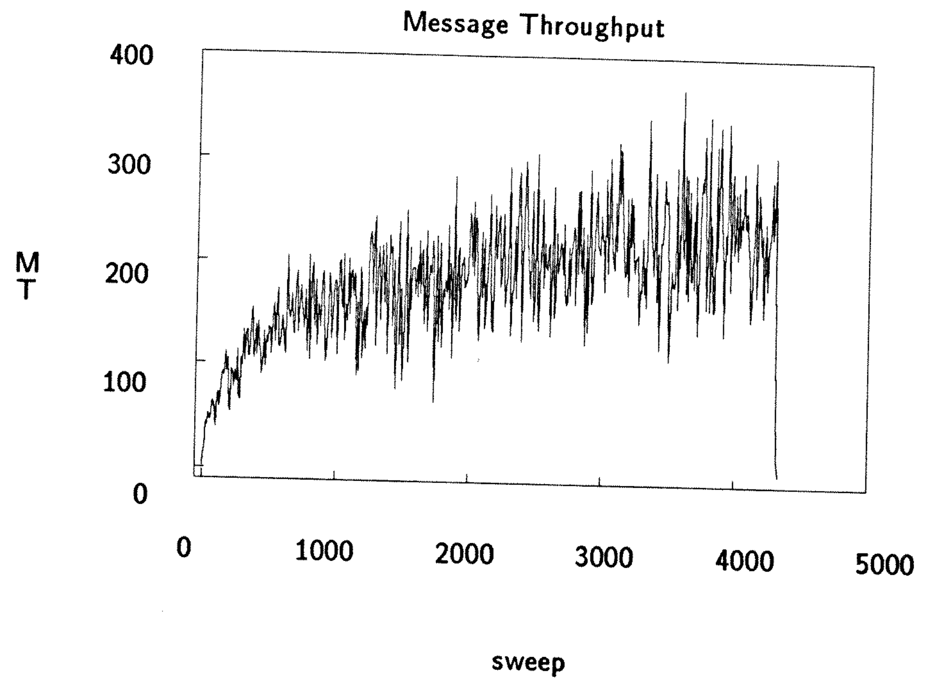


Figure 4.30: Perfect Numbers ($N = 256$)

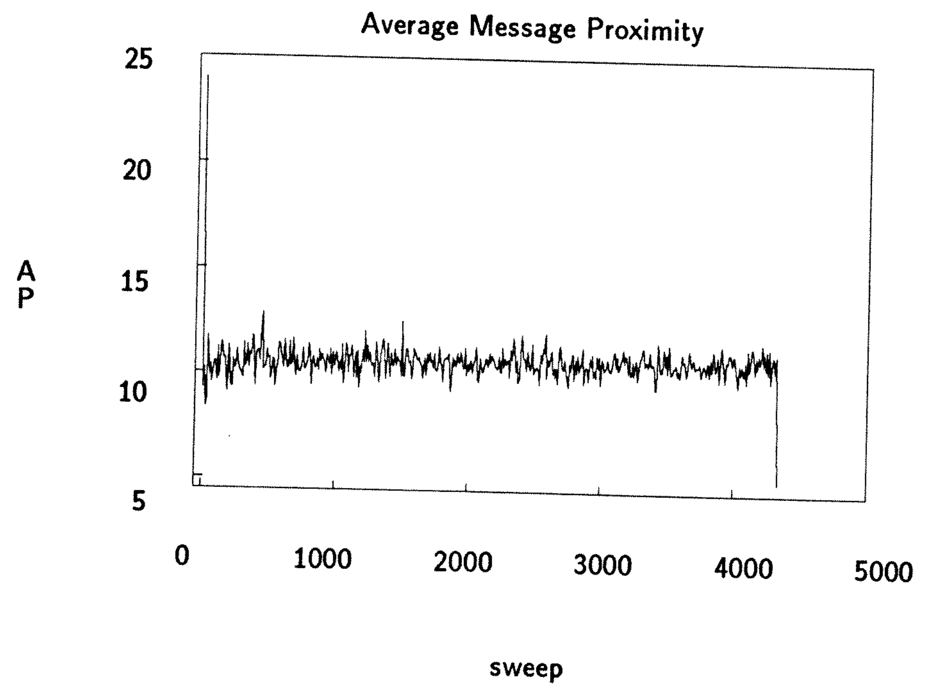


Figure 4.31: Perfect Numbers ($N = 256$)

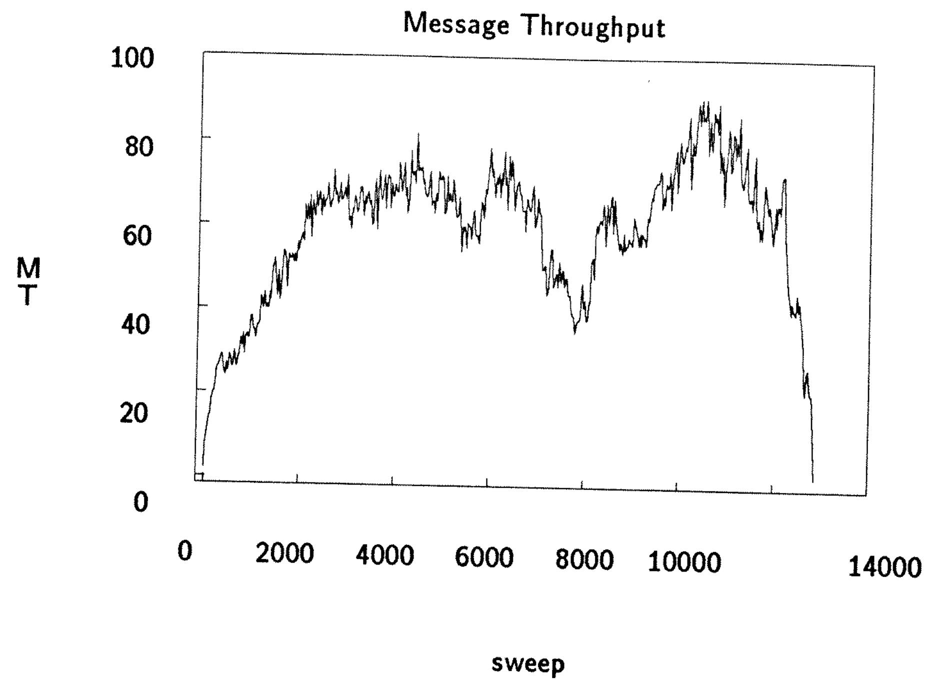


Figure 4.32: Wheel Driven Prime Sieve ($N = 256$)

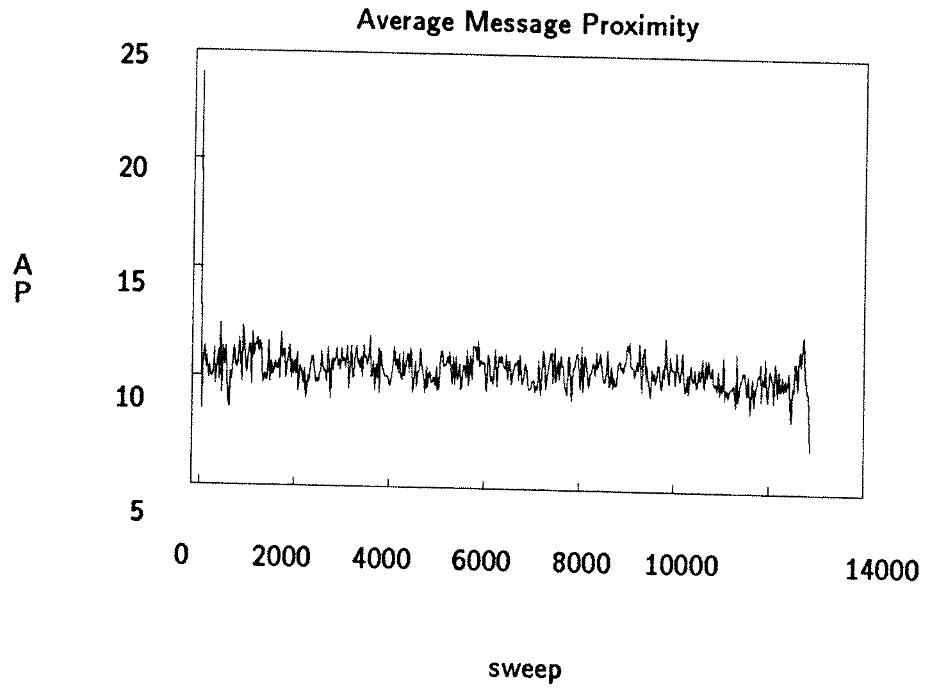


Figure 4.33: Wheel Driven Prime Sieve ($N = 256$)

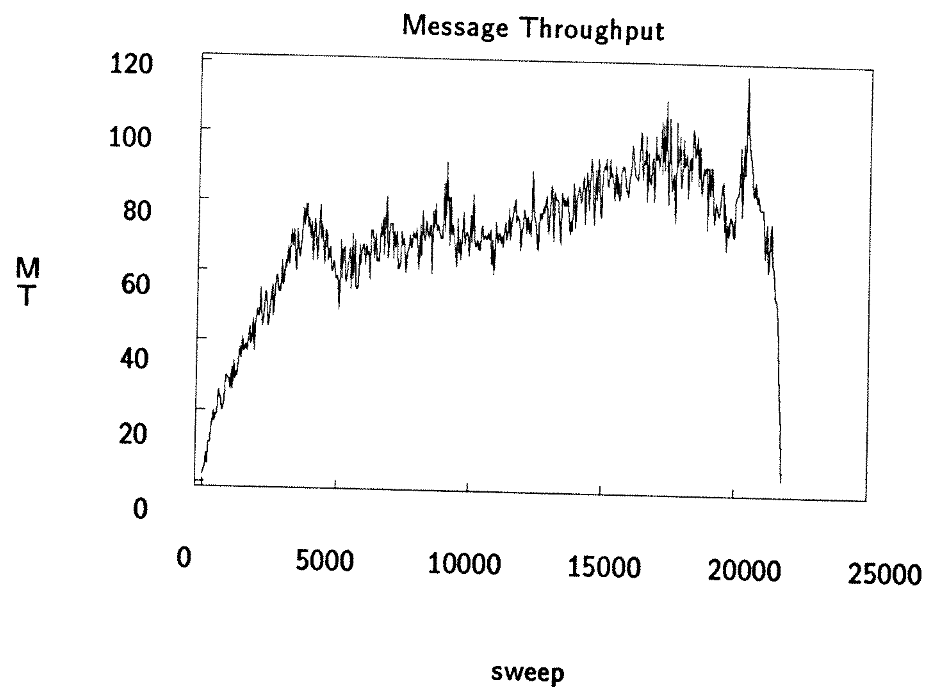


Figure 4.34: Demand Driven Prime Sieve ($N = 256$)

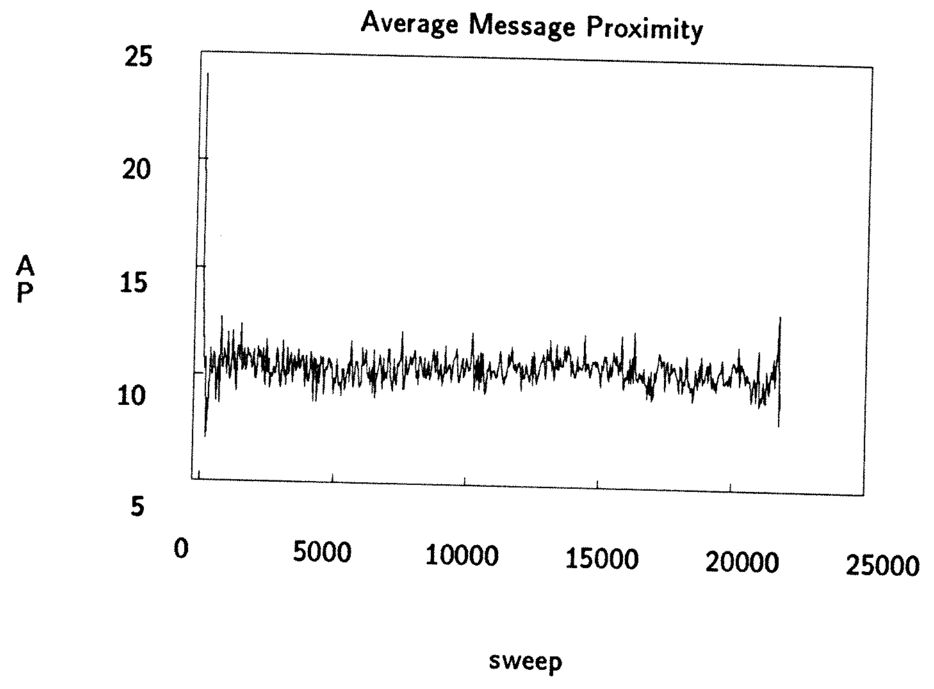


Figure 4.35: Demand Driven Prime Sieve ($N = 256$)

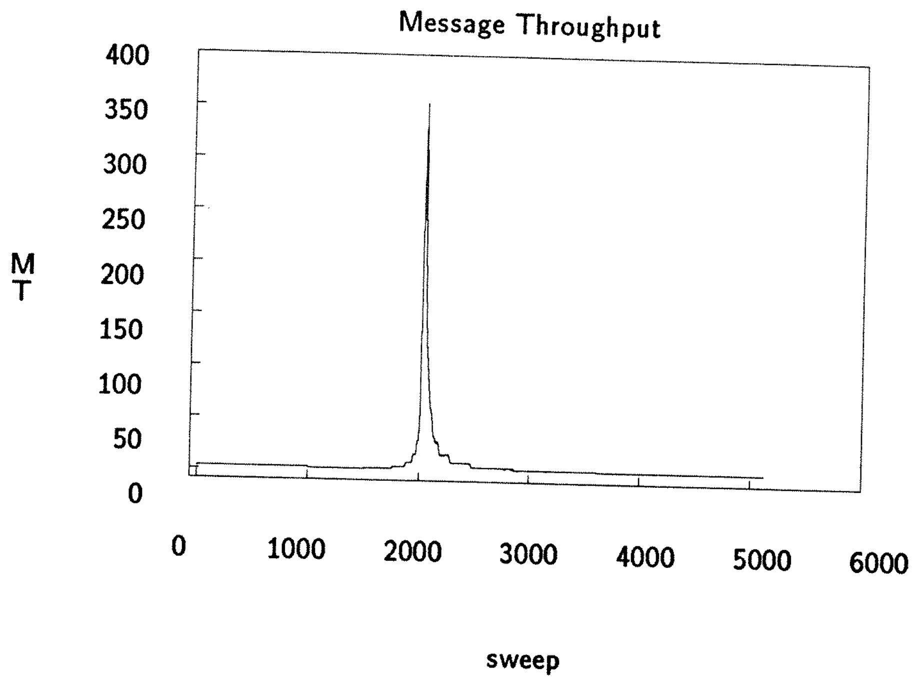


Figure 4.36: Merge Sort ($N = 256$)

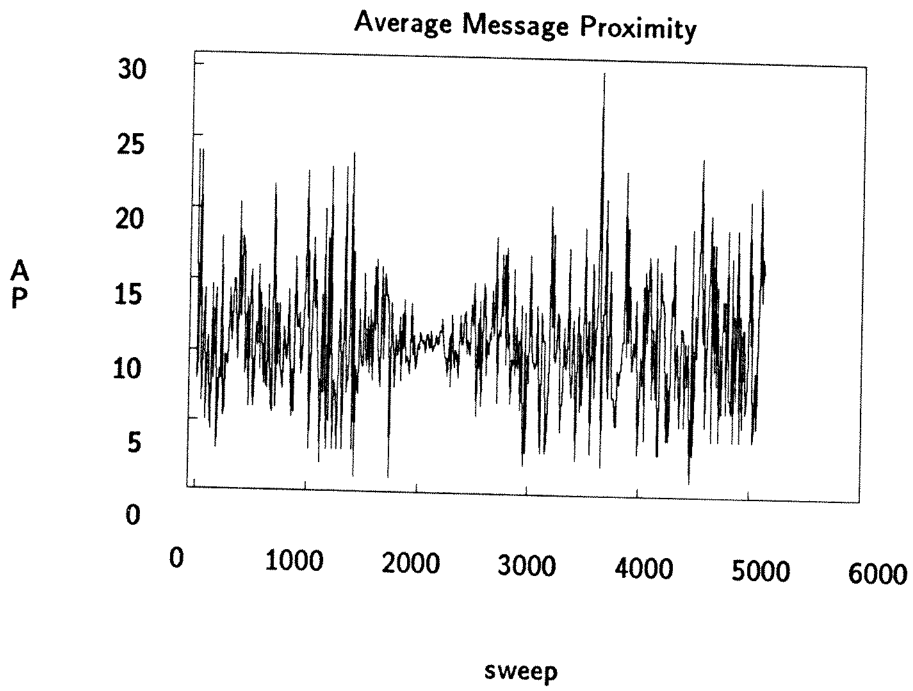


Figure 4.37: Merge Sort ($N = 256$)

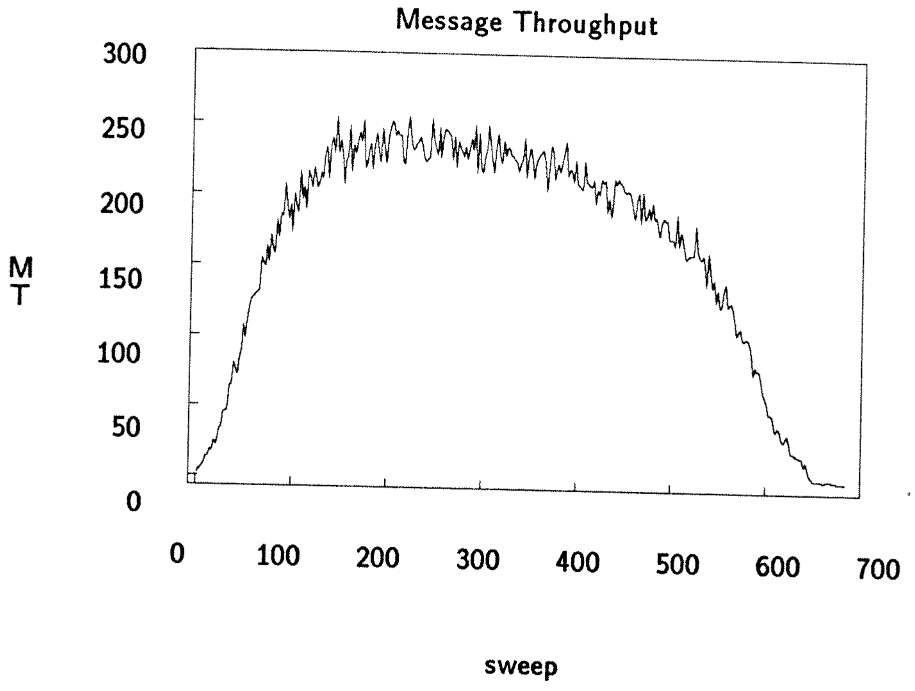


Figure 4.38: Eight Queens ($N = 256$)

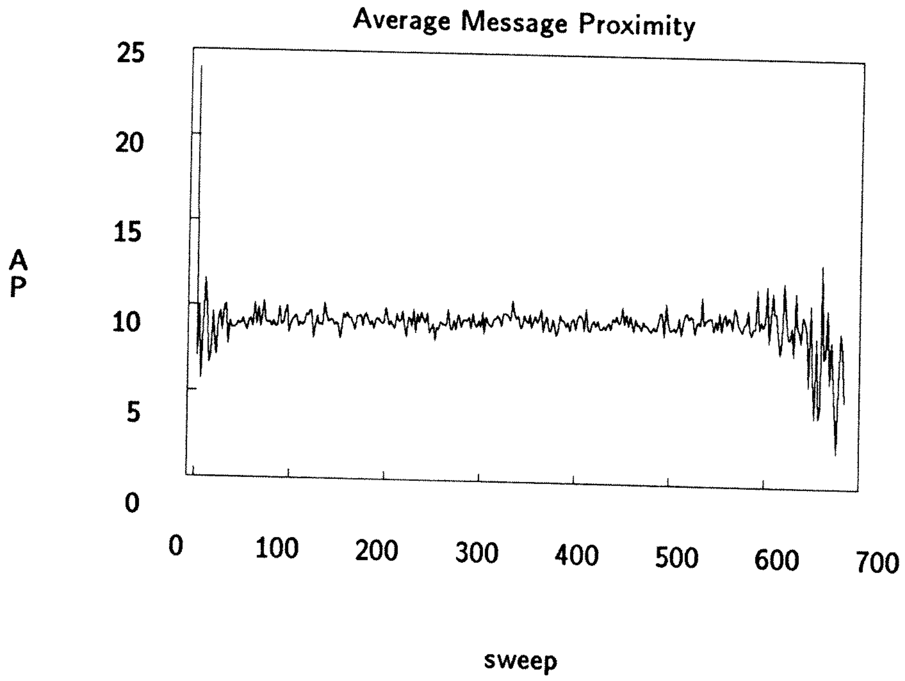


Figure 4.39: Eight Queens ($N = 256$)

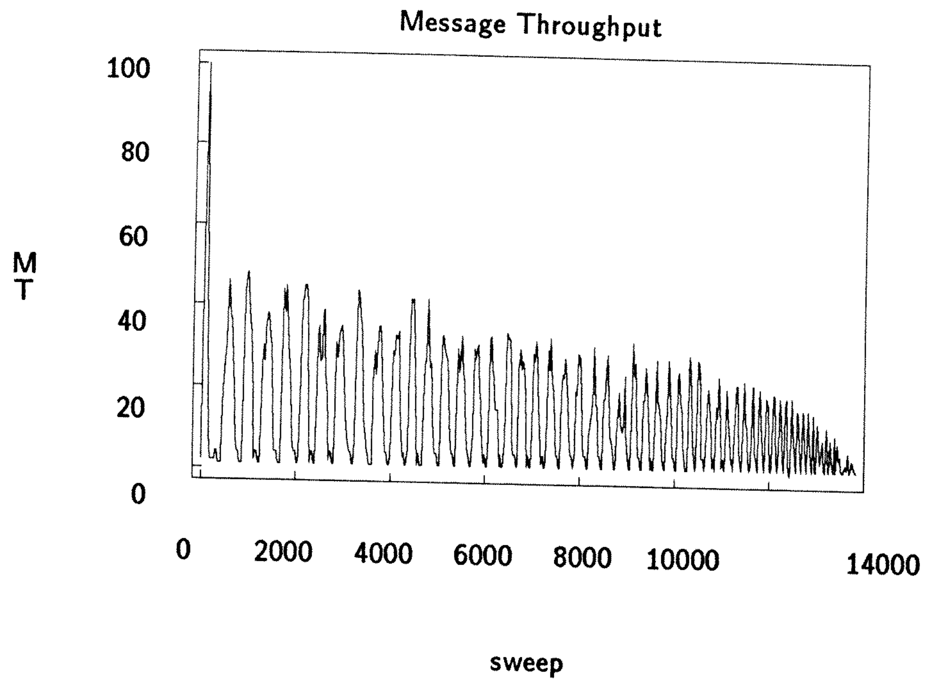


Figure 4.40: Gaussian Elimination ($N = 256$)

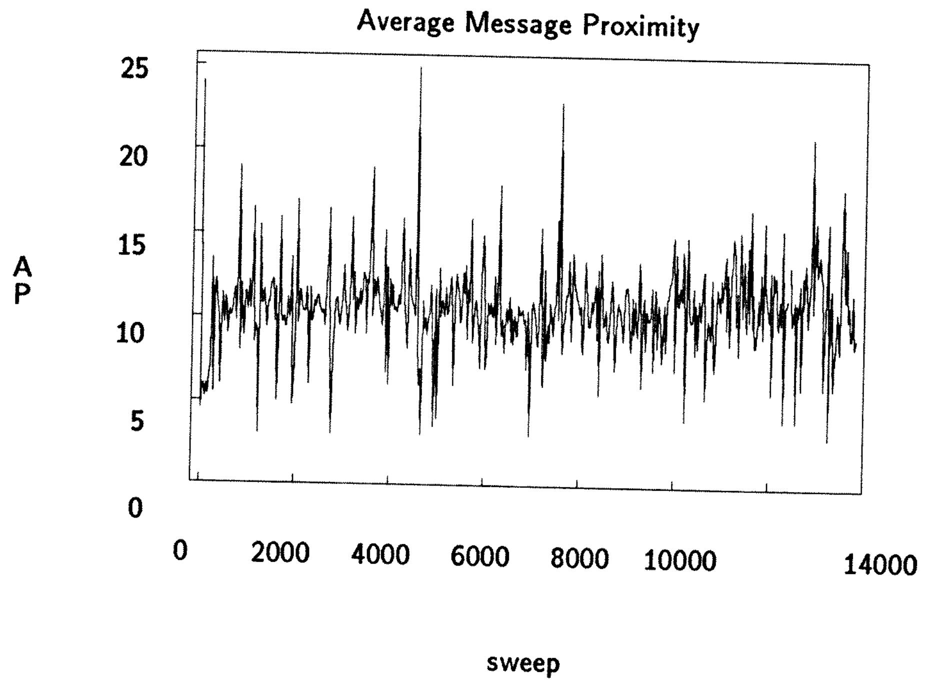


Figure 4.41: Gaussian Elimination ($N = 256$)

4.4.1 Logical Neighborhoods

The average distance between communicating objects can be decreased for many programs by using a placement strategy that has a built-in locality metric. The locality metric is defined so as to minimize the Manhattan distance for the assignment of new objects to nodes. A straightforward locality metric is to assign to each node a set of neighboring *logical* nodes and limit the assignment of new objects to the logical neighbors. The logical subsets are selected so that the Manhattan distance between the object and the object that is being created is minimized.

The possible methods for selecting the logical neighborhoods are myriad. A straightforward approach is to view the nodes as vertices on a undirected graph called the logical graph. The mapping between nodes and vertices is one-to-one and the vertices adjacent to a vertex correspond to the set of logical neighbors for a node. The mapping of nodes to vertices is performed so as to minimize the Manhattan distance between neighboring logical nodes. For the general case, finding the optimal assignment of nodes to vertices is an NP-complete problem. The Manhattan distance metric by its definition is, however, equivalent to placing the nodes on a two-dimensional mesh. By carefully selecting the logical graph, the optimal embedding of the logical graph onto the mesh can be derived analytically.

The three carefully chosen logical graphs are a binary n-cube, and a two- and three-dimensional torus referred to as torus and cube, respectively. The optimal embedding of the torus into the mesh is shown in Figure 4.42. Assuming that message traffic is uniformly distributed within the logical subset of nodes, the average distance between two logical neighbors asymptotically approachs two. This average distance value holds also for the less ingenious strategy of assigning $\sqrt{N}-1$ logical connections of Manhattan distance 1 and a single Manhattan distance of $\sqrt{N}$ for the wrap around.

The embedding of the cube is accomplished by first tesselating the plane with a set of $N^{\frac{1}{3}}$ tori and then linking the two-dimensional tori into the three-dimensional torus. The embedding of the n-cube follows a recursive construction in which the binary valued dimensions are grouped into pairs and each pair is assigned a Manhattan distance that is a power of 2 from the origin. Using these constructions and assuming that message traffic is uniform among the neighbors, the average distance for the torus, cube and n-cube are:

$$\bar{d}_t = 2(1 - N^{-\frac{1}{2}}) \tag{4.1}$$

$$\bar{d}_c = \frac{1}{3}(2(1 - N^{-\frac{1}{3}}) + N^{\frac{1}{3}}) \tag{4.2}$$

$$\bar{d}_n = \frac{2}{\log_2 N}(N^{\frac{1}{2}} - 1) \tag{4.3}$$

Each of the three logical graphs trade off locality for dispersion or load balancing. The torus and cube are the most conservative for locality, limiting the set of neighboring nodes to four and six, respectively. In contrast, the n-cube graph logarithmically increases the size of the logical neighborhood as the number of nodes is increased. The effects of these three strategies upon locality and load balancing are shown in Figures 4.43 through 4.54. The strategy for picking a neighbor within a logical subset is to choose the neighbor with the lowest object count. Message load, random selection, or a composite scheme could equally have been tried. For the case of multiple logical neighbors having equal object counts, ties are broken in favor of the node with the smallest Manhattan distance. For reference purposes, all three strategies were compared against random placement.

The average distance curves and speedup curves, when compared side by side, establish the trade-offs between imposing locality and disturbing the load balancing. The two prime sieves showed the most improvement using the new placement strategies. The average distance between communicating objects decreased by over a factor of five for the 4096 node case, and load balancing improved. This favorable result is easily explained by the semantics of these two programs. From the behavior of the sieve object definitions for both programs, at most one new sieve object is created for each existing sieve object. Except for messages sent to console, message flow in the sieve is limited to adjacent stages. Thus, locality of reference is quite strong for these two programs. Using random placement for these two programs is wasteful of message resources.

Locality of reference is also quite strong in the perfect numbers programs, and Figures 4.43 and 4.44 reflect this strength. Excluding cube placement, for average distance, the larger the logical subset, the larger the average distance. For speedup, the random, cube, and *n*-cube strategies were not appreciably different. The torus placement strategy, however, performed consistently worse for ensembles larger than 64 nodes. The reasons for this difference cannot be explained without examining in detail the assignment of divisor objects.

The speedup and average distance graphs for the merge sort program also display anomalous behavior for the torus and cube placement strategies. For the merge sort program, random placement did appreciably better for speedup. The torus placement strategy performed poorly until the number of nodes increased to 1,024. This dramatic change in performance suggests that since the list to be sorted is comprised of 1000 objects, when the number of nodes reaches 1,024, the assignment task becomes trivial with one object per node. For average distance, *n*-cube produced consistently better average values up to 1,024 nodes, although the size of the logical neighborhood was the largest. The average distance for the cube strategy was close to the *n*-cube strategy until the ensemble size increased to 1,024, once again indicating some fundamental interaction between the number of list objects and the number of nodes.

The eight queens and Gaussian elimination programs performed most closely to the predicted results, namely that torus would be worst for load balancing, followed by cube, and then *n*-cube. This behavior is shown clearly in the Gaussian elimination speedup graph (Figure 4.54). For the eight queens program however, the performance of *n*-cube placement was not appreciably different from random placement. In contrast, the average distance curves show *n*-cube placement superior to random placement, as well as the other smaller subset strategies. Overall though, the two average distance graphs for Gaussian elimination and eight queens demonstrate that locality is not being exploited. This lack of locality is in the programs since both programs frequently send reference values as the component of a message, and is also in the placement algorithms since they do not lock onto the object graphs that each of the programs build.

In summary, the *n*-cube placement strategy did surprising well. The dispersive qualities of the *n*-cube, namely that it contains all periodic graphs of lower dimension, proved

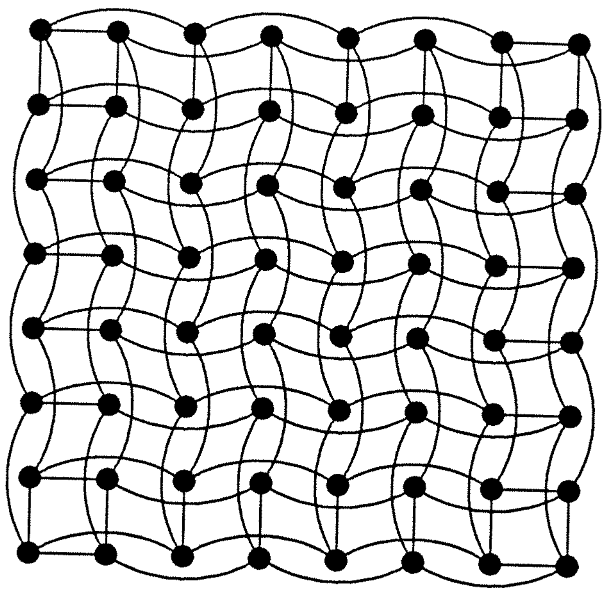


Figure 4.42: Two-Dimensional Torus

to be efficient for both load balancing and locality. In contrast to random placement which always assumes maximum dispersion, n-cube placement incrementally introduces dispersion by selecting destination nodes from minimum to maximum Manhattan distance.

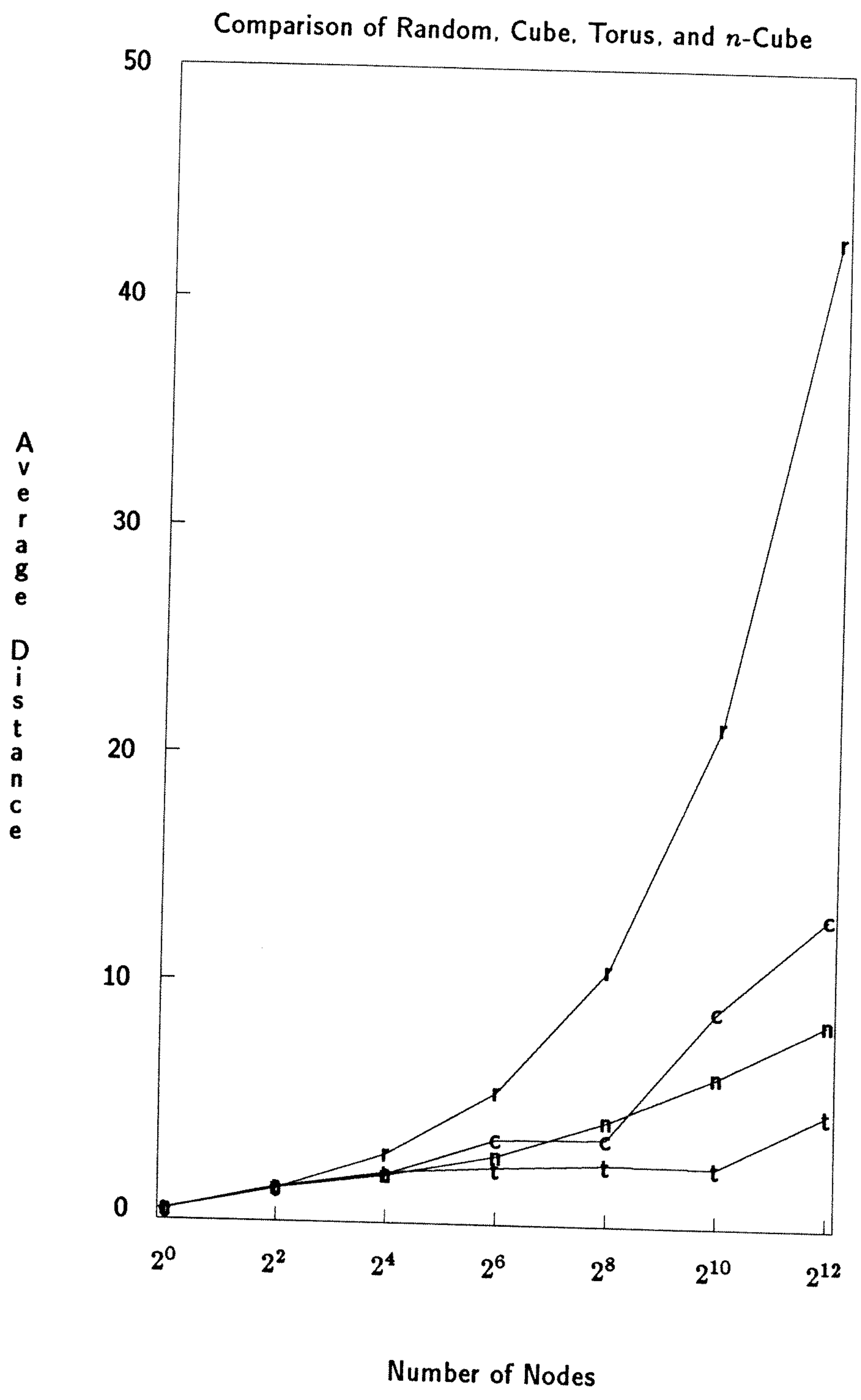


Figure 4.43: Average Distance Comparison for Perfect Numbers

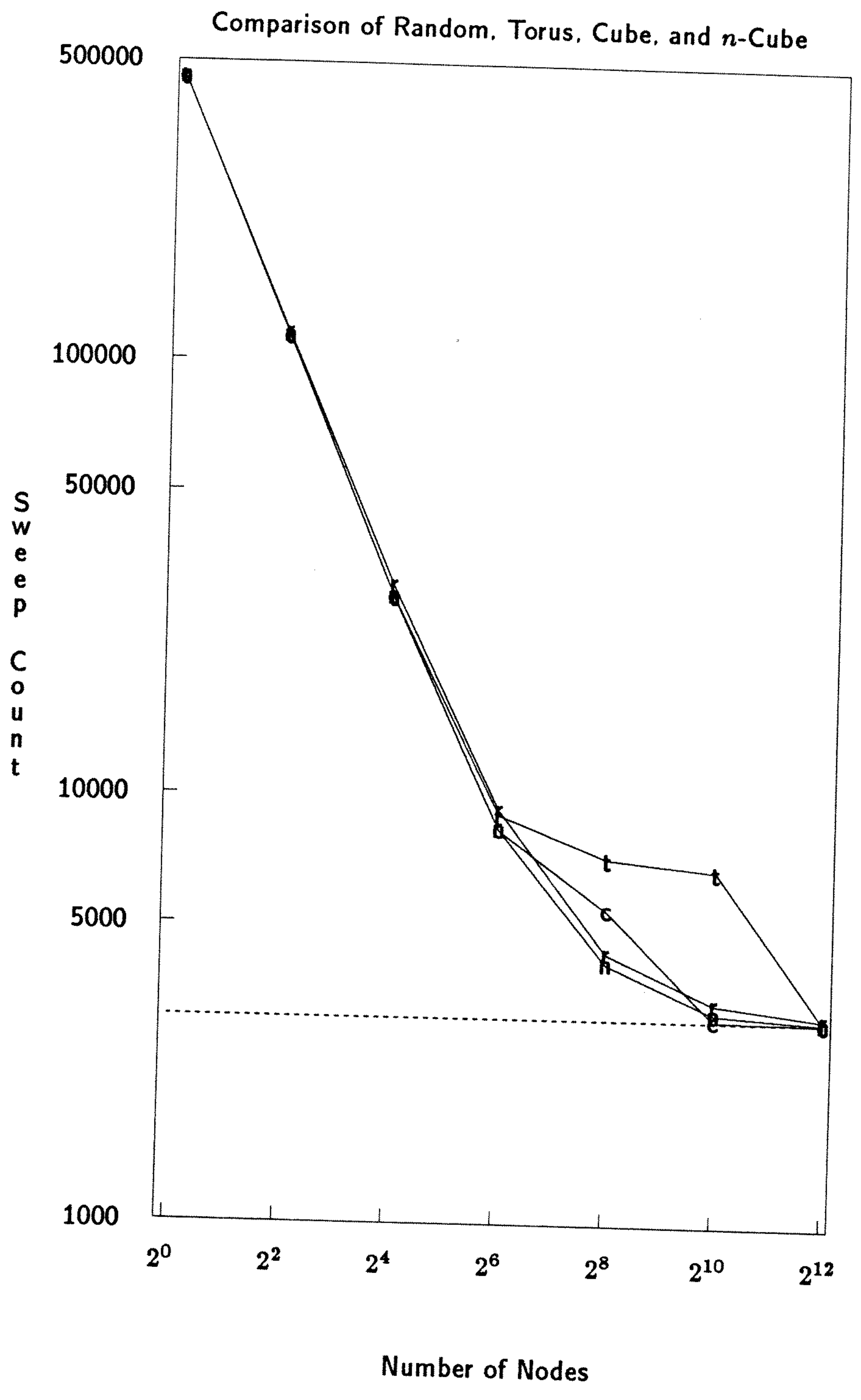


Figure 4.44: Speedup with Locality for Perfect Numbers

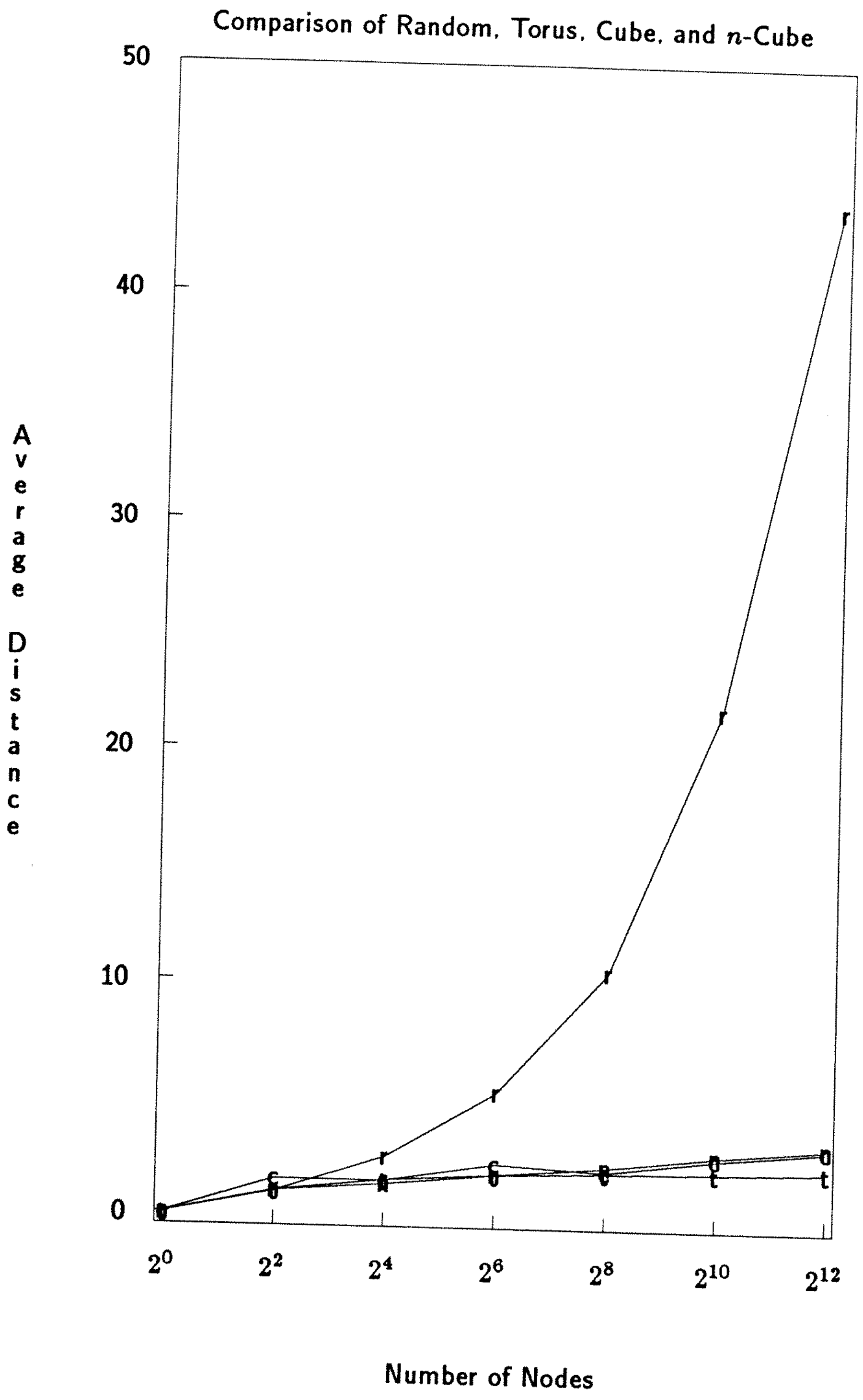


Figure 4.45: Average Distance Comparison for Wheel Driven Prime Sieve

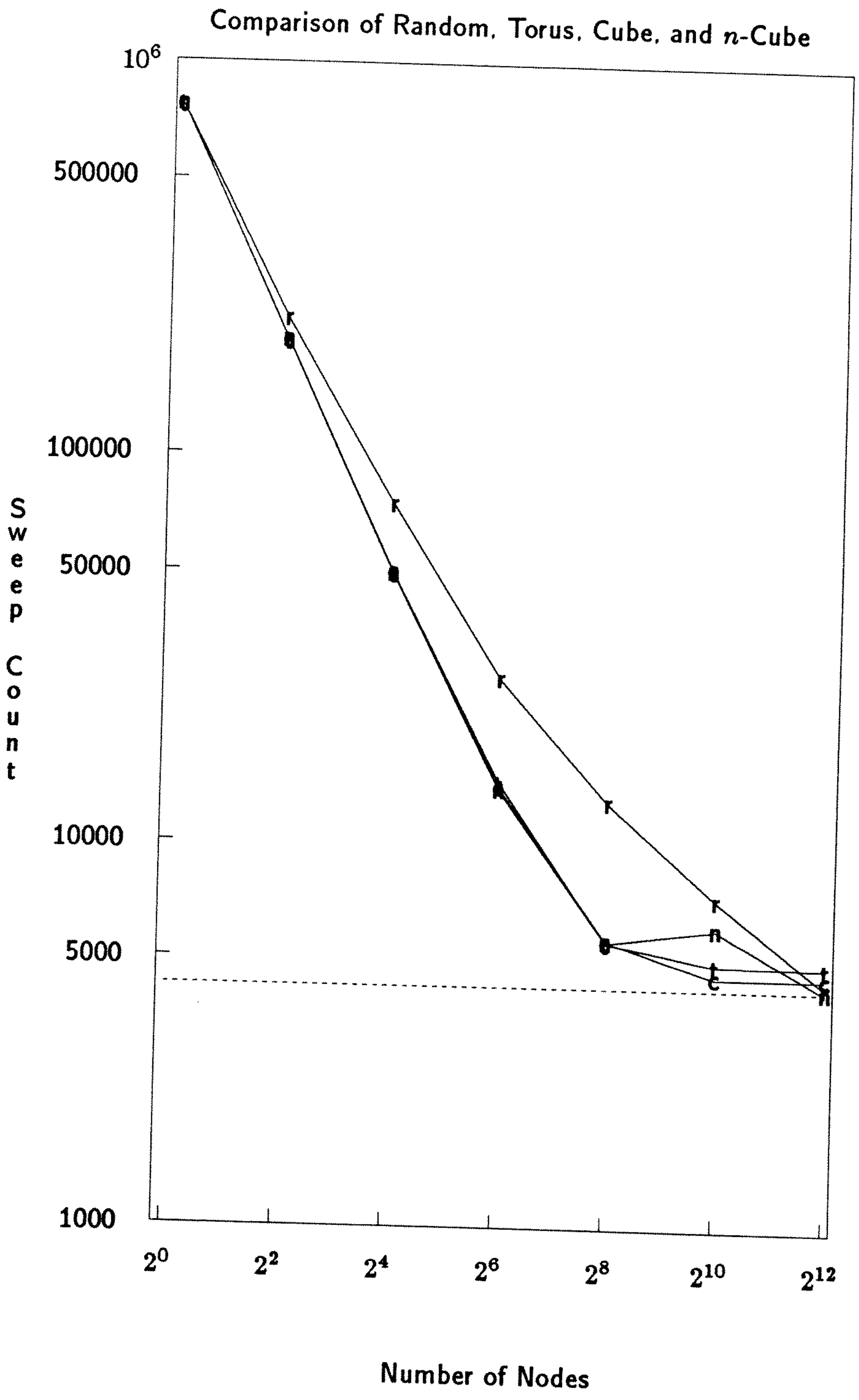


Figure 4.46: Speedup with Locality for Wheel Driven Prime Sieve

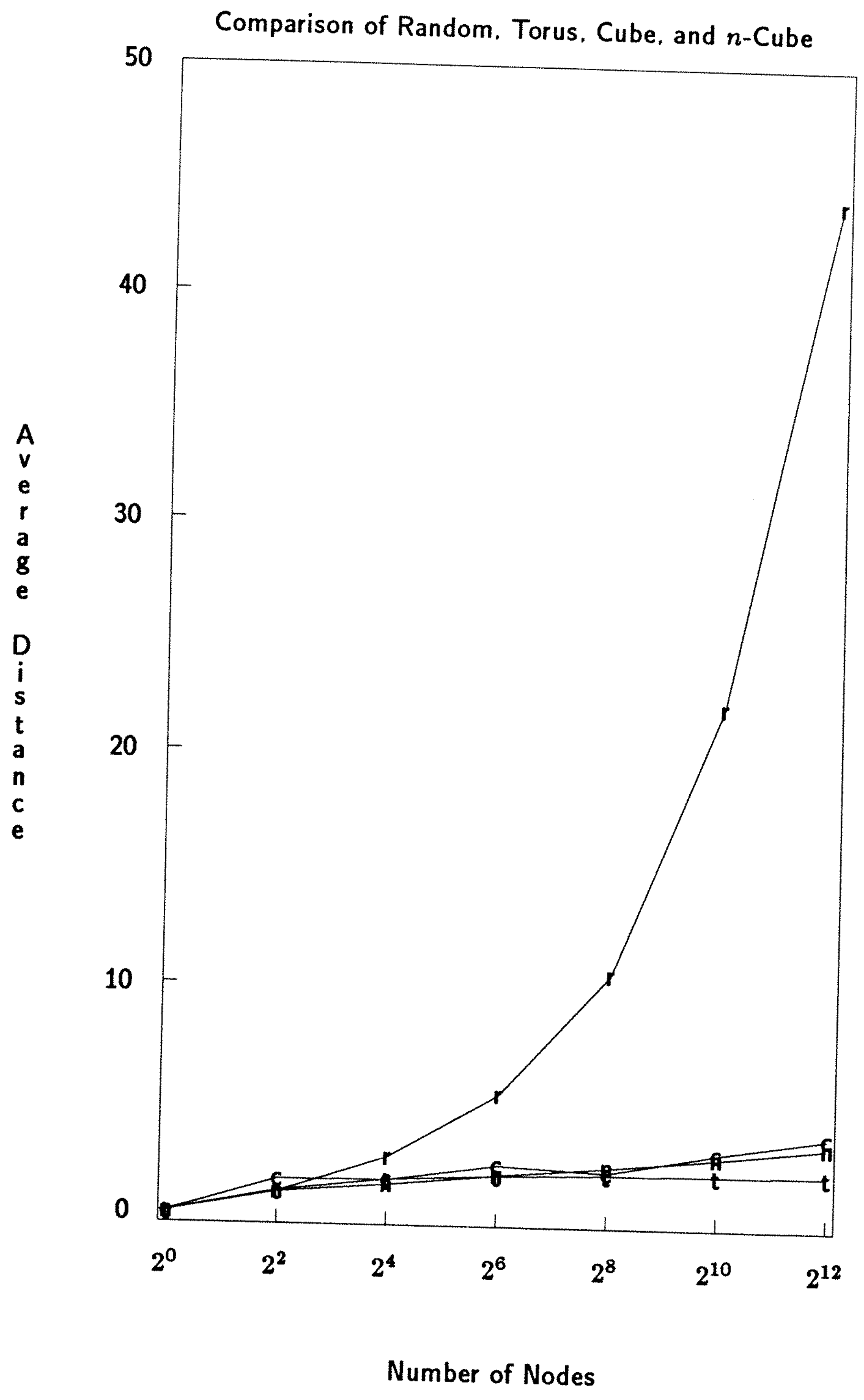


Figure 4.47: Average Distance Comparison for Demand Driven Prime Sieve

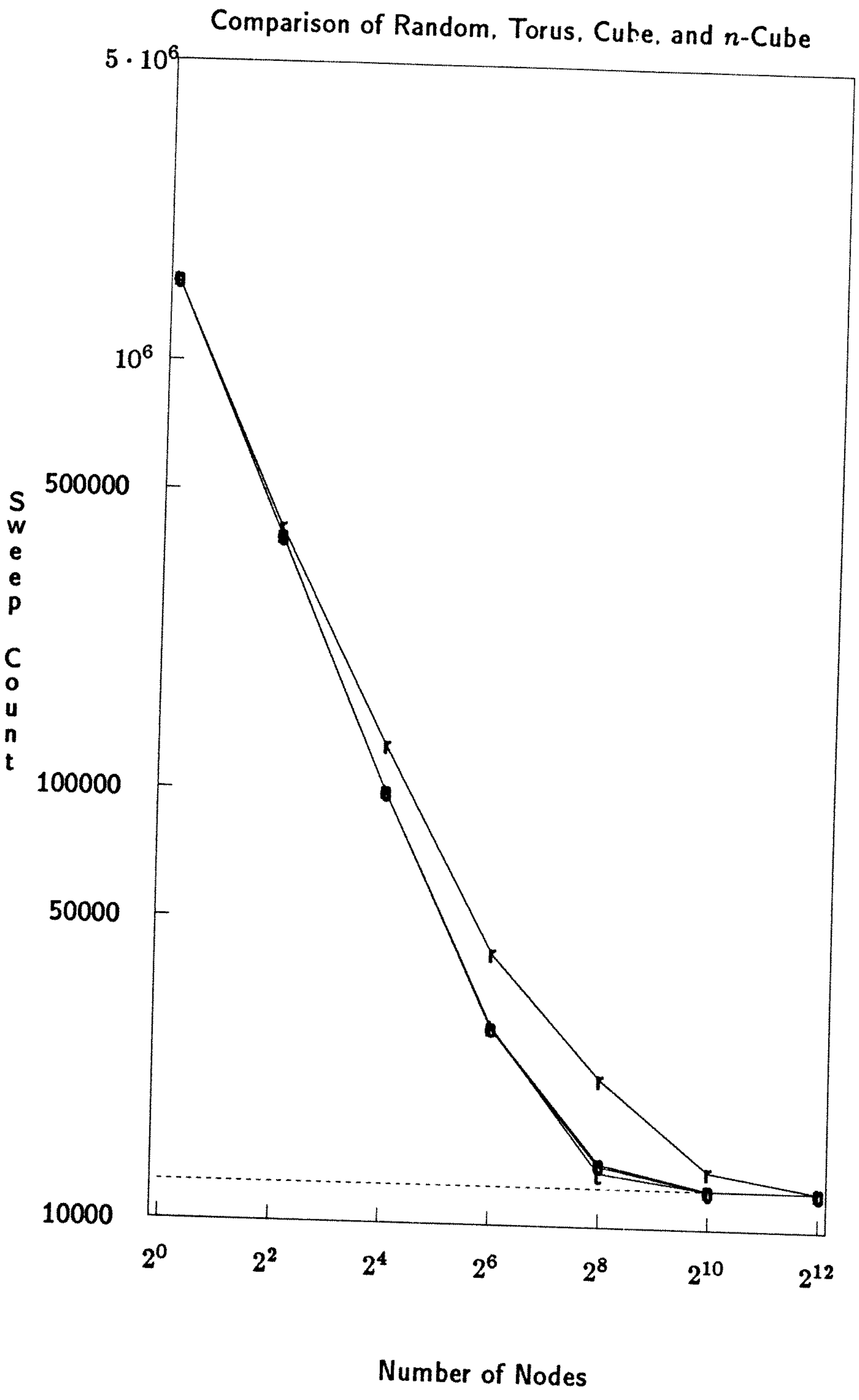


Figure 4.48: Speedup with Locality for Demand Driven Prime Sieve

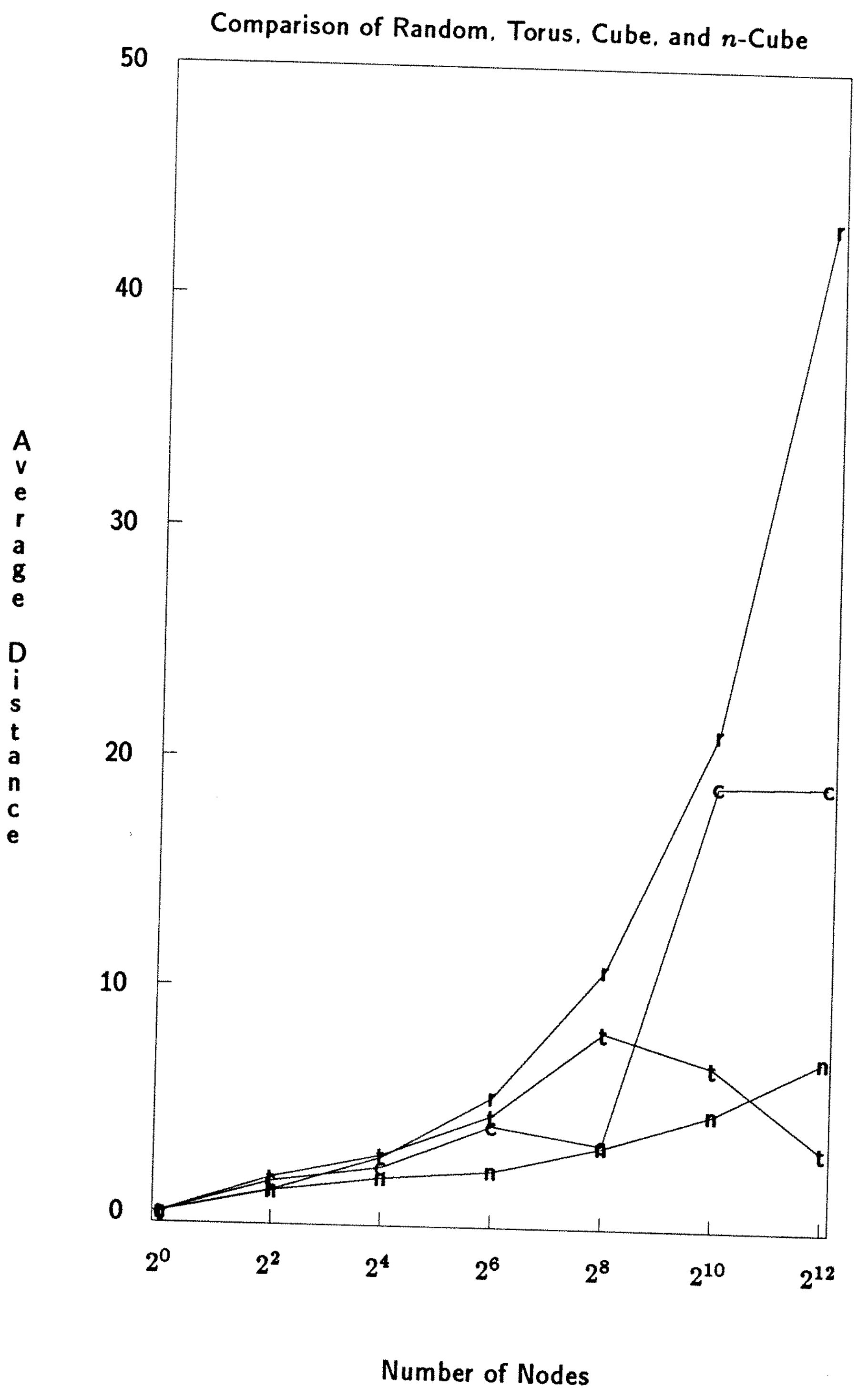


Figure 4.49: Average Distance Comparison for Merge Sort

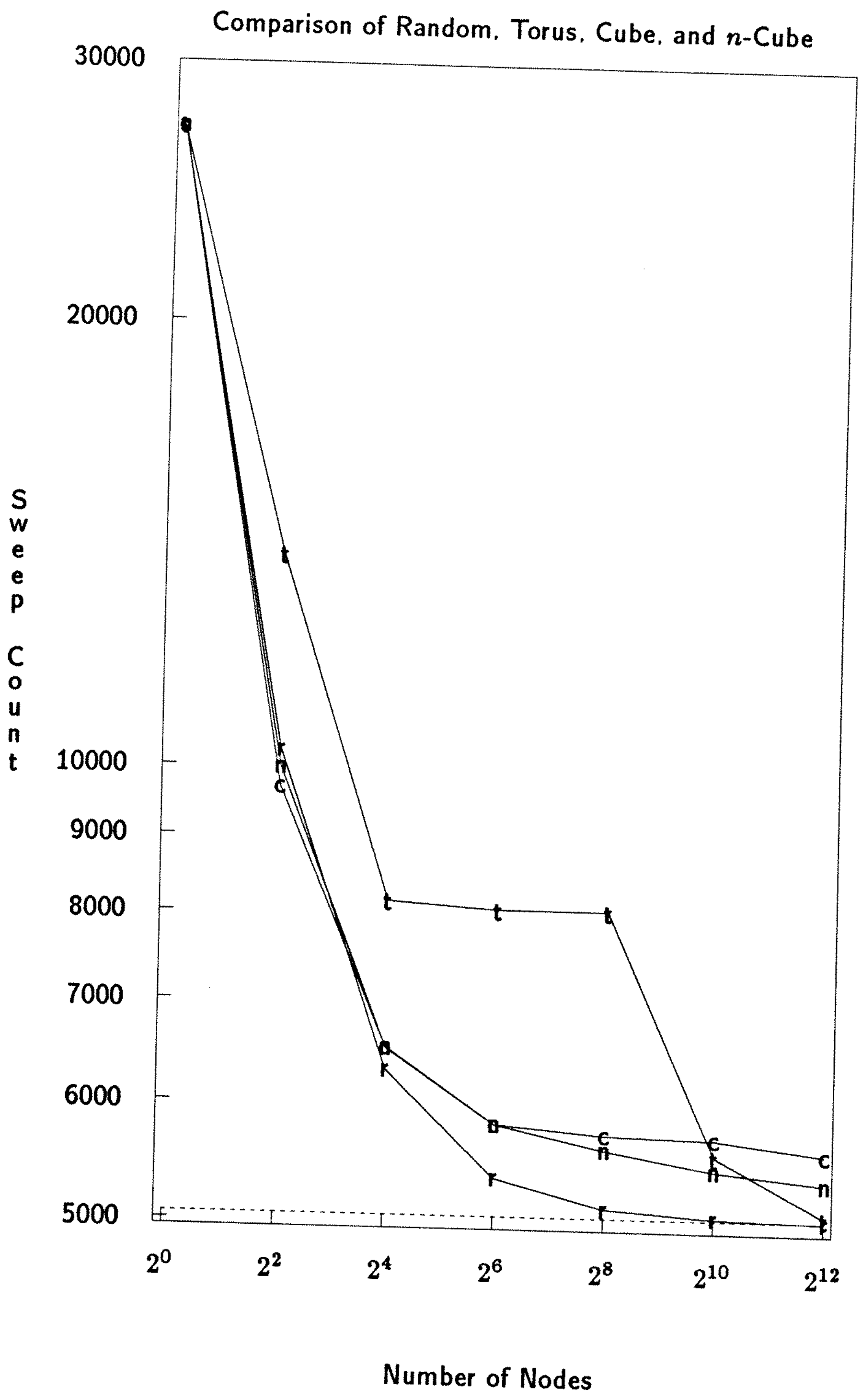


Figure 4.50: Merge Sort

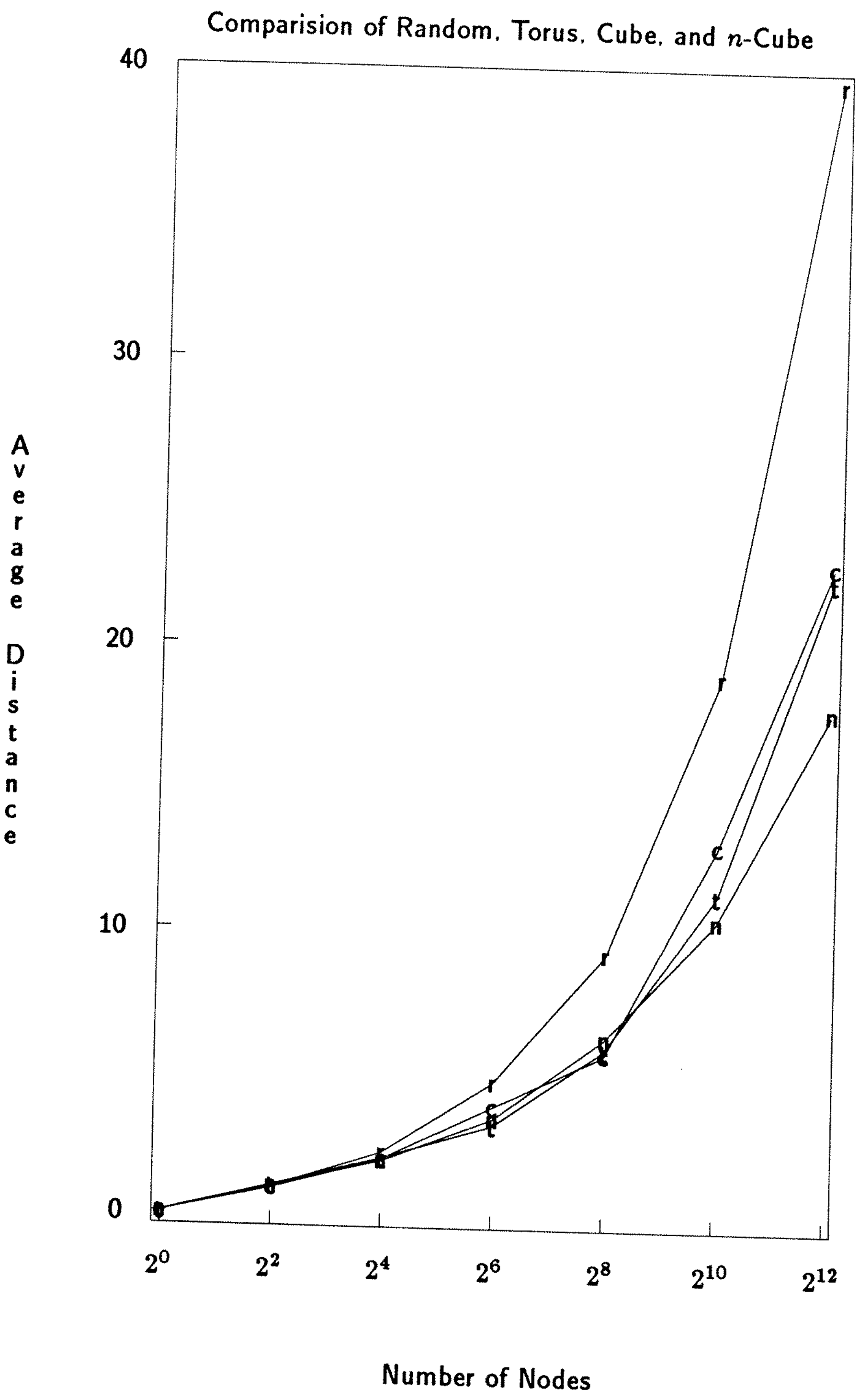


Figure 4.51: Average Distance Comparison for Eight Queens

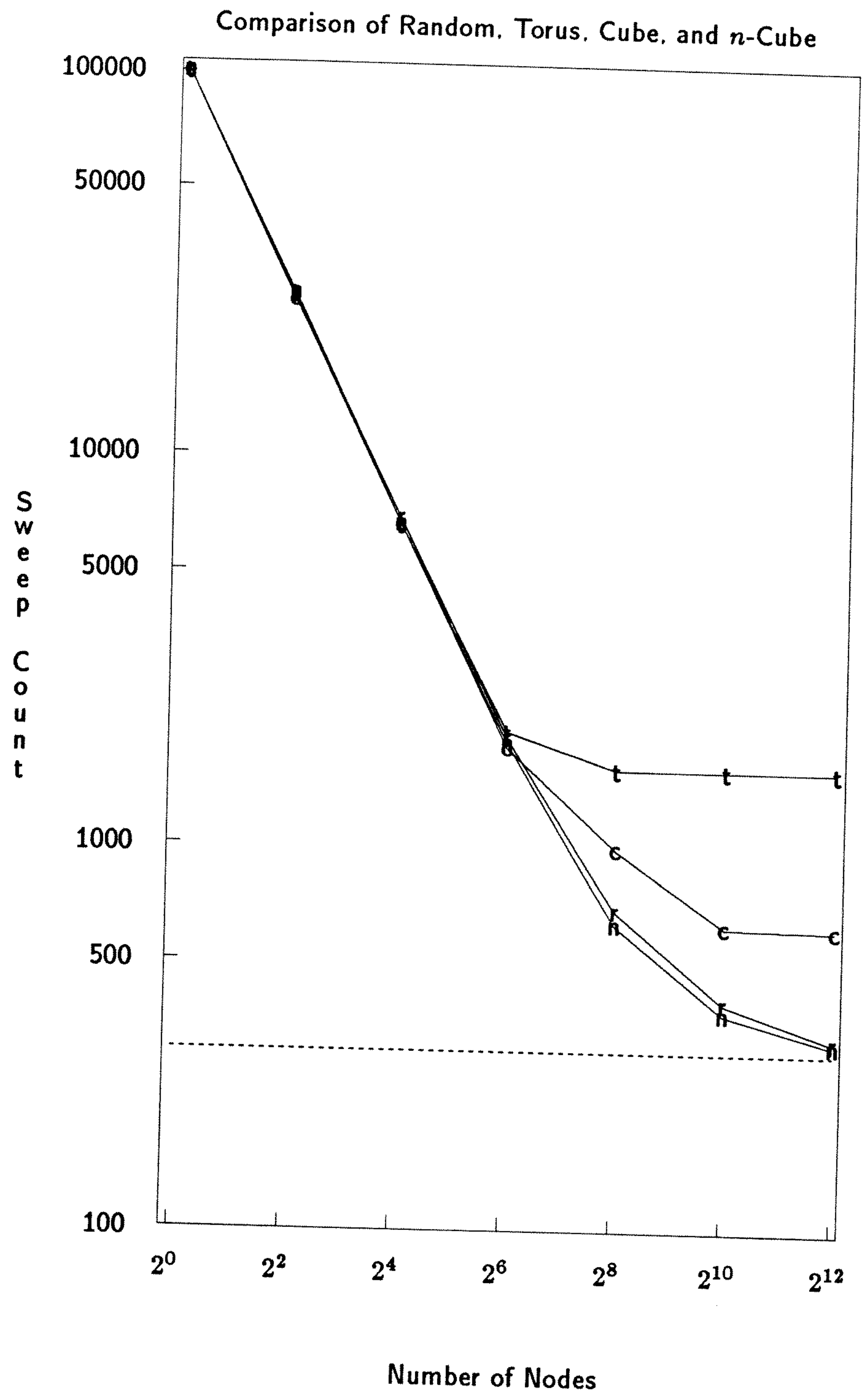


Figure 4.52: Speedup with Locality for Eight Queens

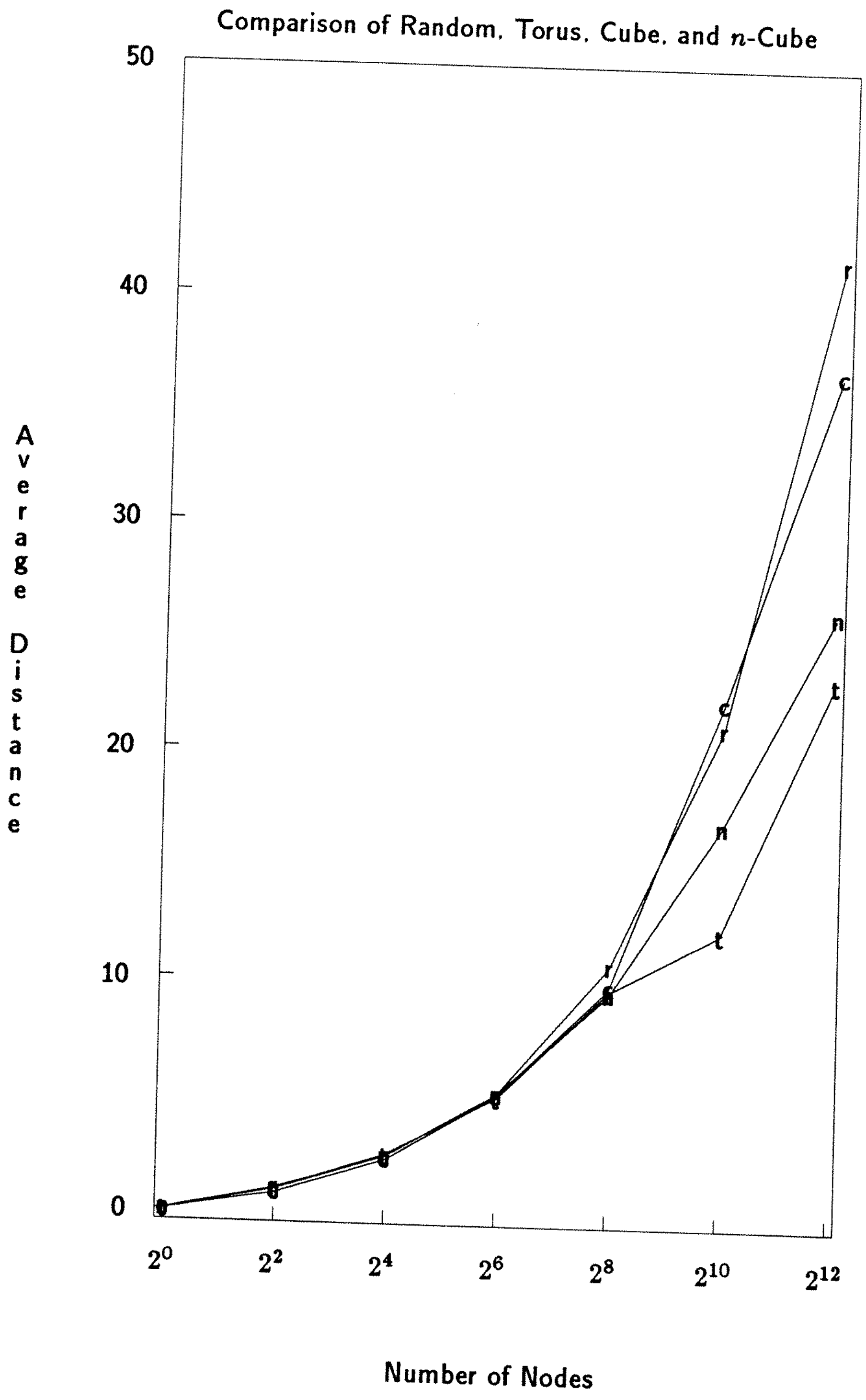


Figure 4.53: Average Distance Comparison for Gaussian Elimination

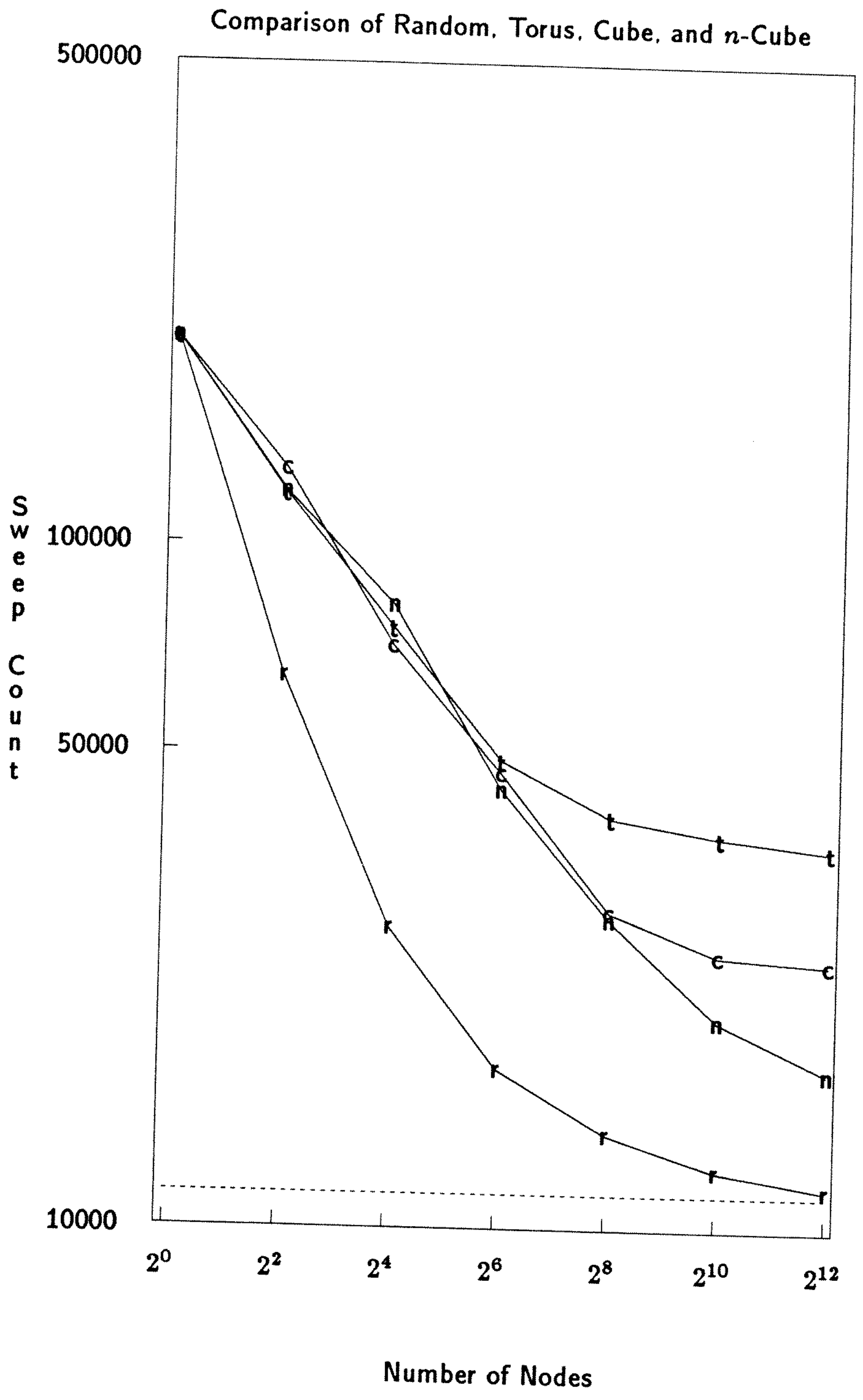


Figure 4.54: Speedup with Locality for Gaussian Elimination

4.4.2 Compiler-Assisted Placement

The results of the neighborhood placement strategies demonstrated the superiority of the n-cube placement strategy for both load balancing and locality for all of the test programs, except Gaussian elimination and merge sort. For these two programs, load balancing and locality behaved purely as opposing constraints. For the other four programs, large logical neighborhoods were unnecessary and thus were wasteful of message resources. Efficient placement strategies, therefore, should only increase the Manhattan distance for the placement of new objects when necessary. This approach is inherent in the n-cube placement strategy because, by construction, the logical neighbors are partitioned pairwise into groups that are powers of two apart in distance.

If program behavior could be predicted before executing the program, then the best placement strategy could be selected. For example, if the program was known to exhibit behavior similar to the two prime sieve programs, then torus placement strategy would be selected. Unfortunately, program behavior is dependent upon many diverse factors. However, information can be gathered during compilation and used ***heuristically*** to assist with the placement of new objects during runtime. The information to be gathered during compilation is the maximum ***fanout*** for each object definition, i.e., the maximum number of messages sent and new objects created in response to receiving a single message, and the propensity for an object to generate more messages and new objects.

The first two quantities are called the ***send factor*** (SF) and ***new factor*** (NF) for the object. The third quantity is called the ***execute factor*** (EF). All three quantities are calculated at compile time by statically analyzing the source program text. The send factor is a count of the maximum number of sends that can occur inside a description and likewise for the new factor. The execute factor is the number of times the object can execute; usually this number is 1 or ∞.

The send, new, and execute factors are available to the runtime system on a per object basis. These three factors can be used to make heuristic decisions about which placement strategy to use for creating new objects. Unfortunately, whether a new object will be used concurrently or not cannot be determined from these three factors. For example, an object with a send factor of one will not generate concurrency. The object, however, can be used indefinitely to ***sustain*** concurrency. Therefore, when creating a new object, the new object can be determined to be capable of either:

- Generating concurrency: $SF > 1$.

- Sustaining concurrency: $SF = 1$.

- Reducing concurrency: $SF < EF$.

An example of an object that reduces concurrency is the adder object of the perfect numbers program. An example of an object that sustains concurrency is the idler object of the wheel driven prime sieve program. From the three factors, numerous program properties can be proved. For example, a program is purely sequential if each object definition used by the program has a send factor ≤ 1.

The three factors are tabulated for the merge sort program in Table 4.2. The list and join are capable of sustaining concurrency, but the msort object is capable of generating

object definition	SF	NF	EF
main	1	1	3
mergesort	1	1	2
msort	2	3	2
join	1	0	3
serializer	2	0	∞
list	1	0	∞
make_list	1	2	∞

Table 4.2: Send, New, and Execute Factors for Merge Sort

concurrency. A placement strategy that uses random placement for new objects that generate concurrency, and the n-cube strategy for new objects that sustain concurrency was applied to the merge sort program. The speedup and average distance graphs are displayed in Figures 4.56 and 4.55. The speedup and average distance curves are compared with pure random and n-cube placement. The two graphs show the desired result of improved locality over the random strategy and better load balancing than the n-cube strategy.

The same placement strategy was repeated on the Gaussian elimination program. The three factors are tabulated in Table 4.3 and the speedup and average distance graphs are shown in Figures 4.58 and 4.57. The results are comparable to the merge sort program: the load balancing is improved over the n-cube strategy and locality is improved over the random strategy.

object definition	SF	NF	EF
main	1	1	1
list	1	0	∞
list_head	2	3	∞
compare	1	0	∞
gauss	1	1	∞
build_list	1	0	1
build_matrix	3	2	∞

Table 4.3: Send, New, and Execute Factors for Gaussian Elimination

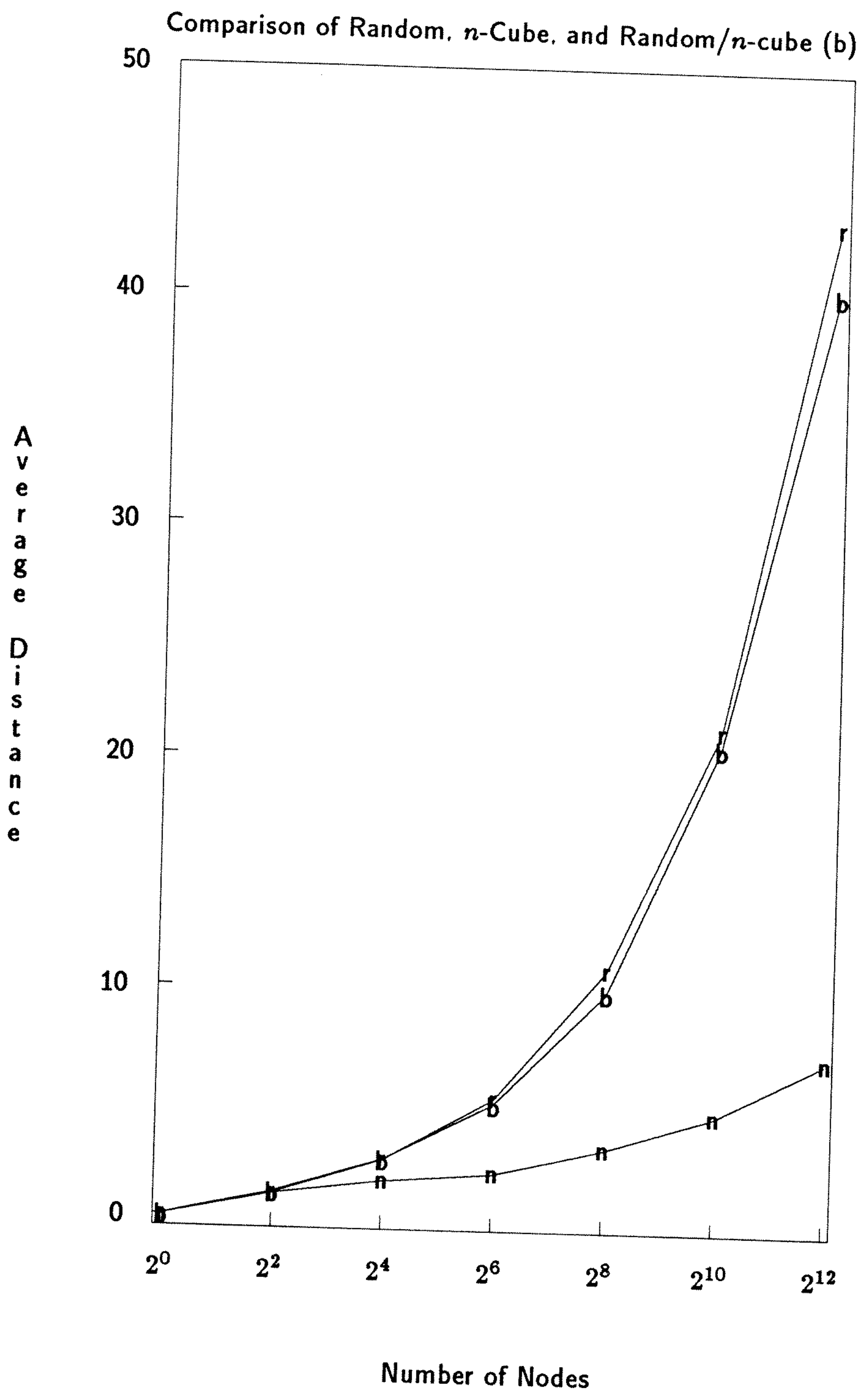


Figure 4.55: Average Distance with Compiler-Assisted Placement for Merge Sort

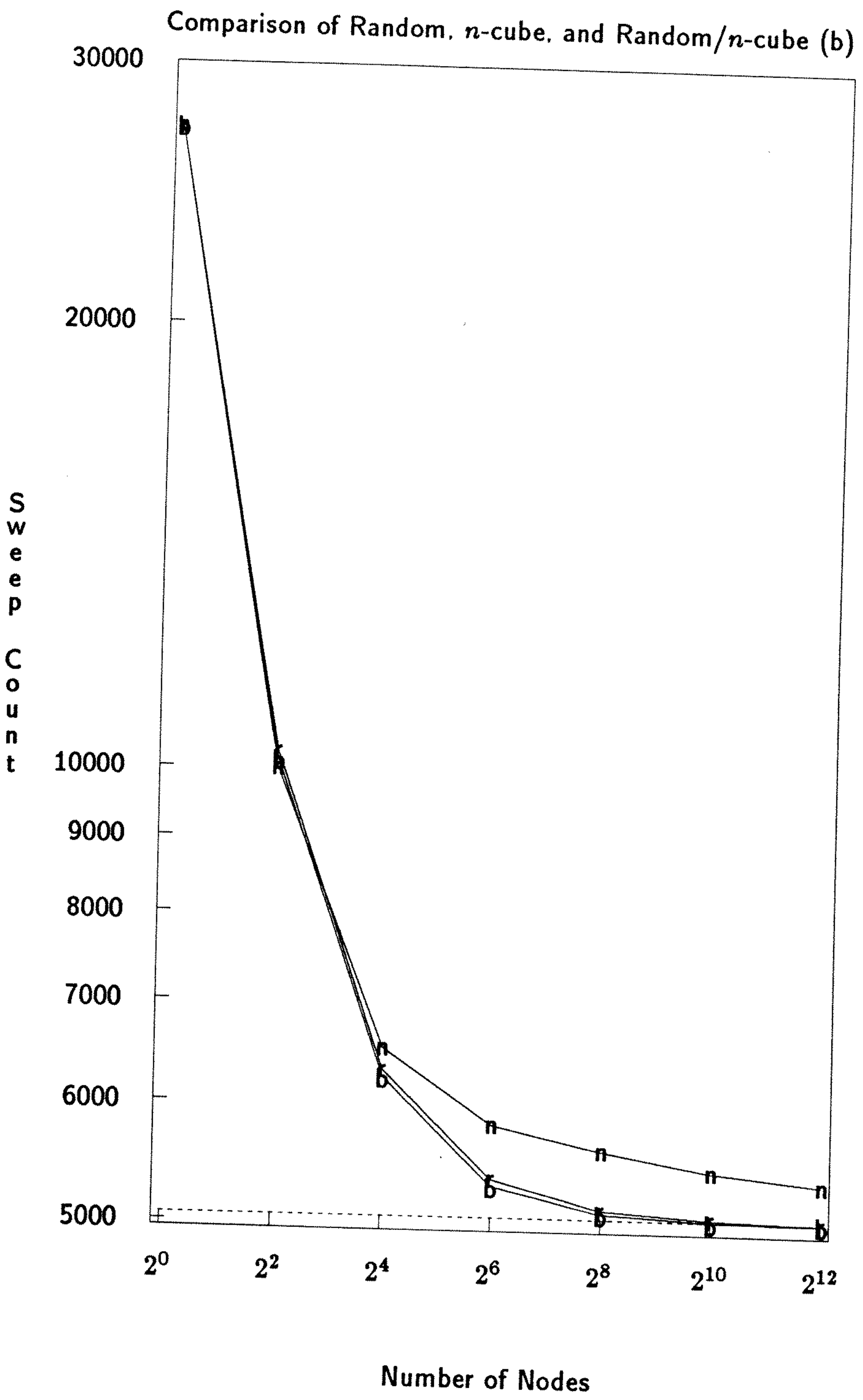


Figure 4.56: Speedup with Compiler-Assisted Placement for Merge Sort

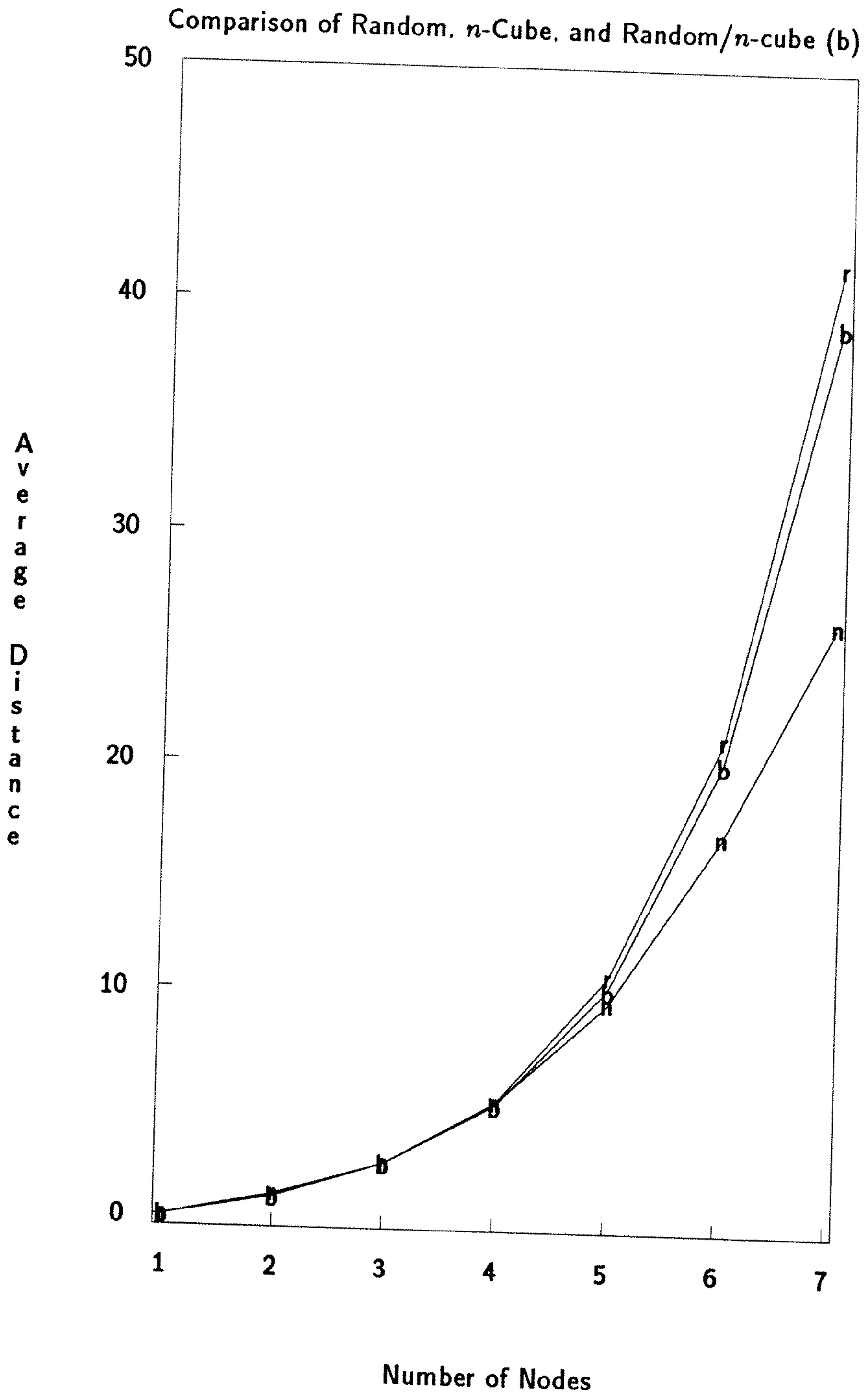


Figure 4.57: Average Distance with Compiler-Assisted Placement for Gaussian Elimination

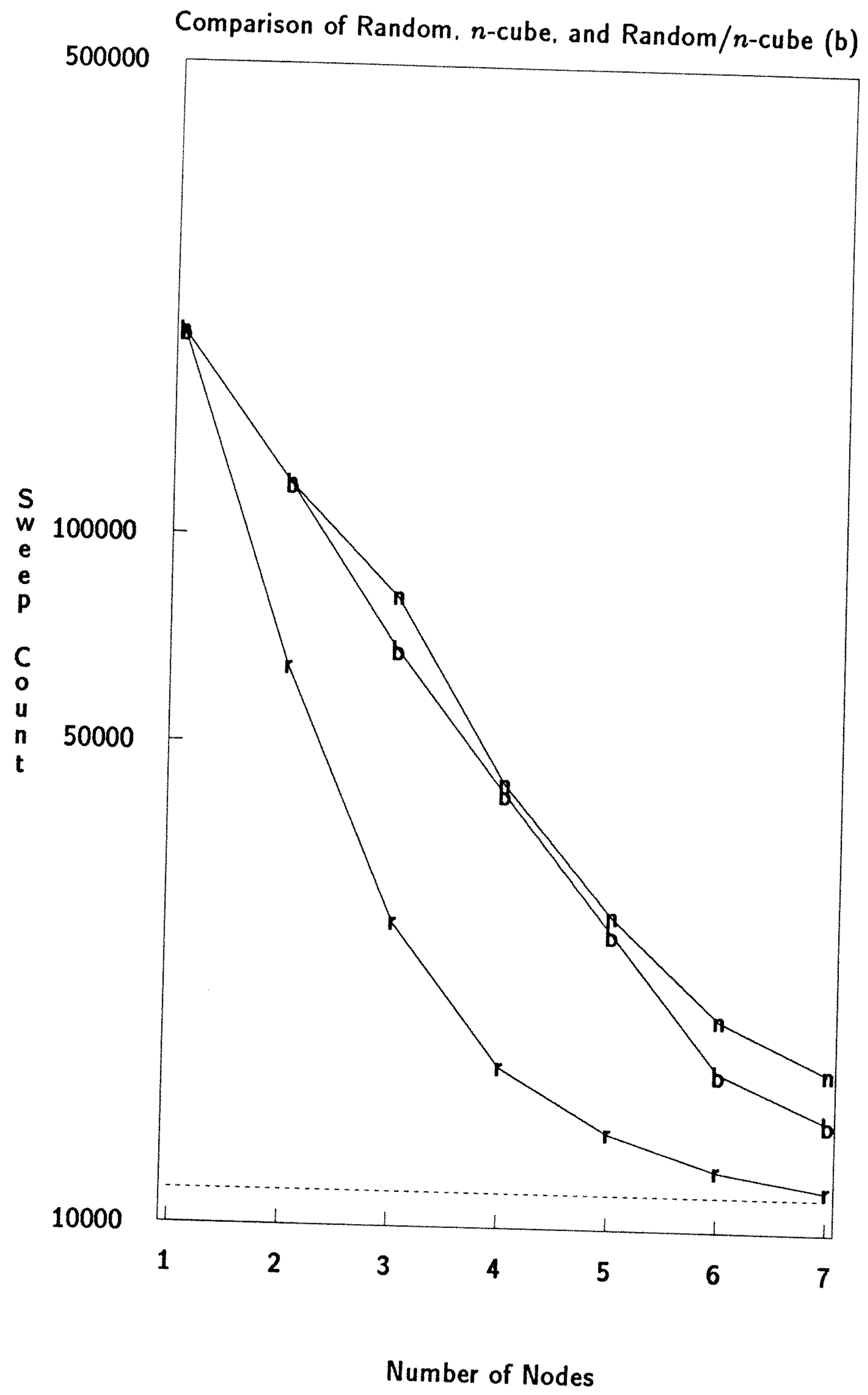


Figure 4.58: Speedup with Compiler-Assisted Placement for Gaussian Elimination

4.5 Summary

For programming environments where locality can be ignored, e.g., shared-memory computers or ensemble machines where the message throughput message-passing network is sufficiently larger than the rate at which the nodes can process messages, random placement is a promising load balancing strategy for assigning objects to nodes. For environments where the cost of message-passing is not negligible, strategies such as the logical neighborhood approach can be used to increase locality. These locality imposing strategies, however, can reduce the exploitable concurrency for some types of object programs, namely, for object definitions that have large send and new factors, and object programs where locality of reference is not used extensively. For these programs, the composite approach of using a global placement strategy for the high fanout objects and a local placement strategy for the relatively docile objects can be used to improve load balancing and locality. The gains are marginal, however, because the lack of locality in reference is embedded in the semantics of the program.

Compiler-assisted placement is one form of resource allocation that is performed when the source program is compiled. The larger strategy is to use the future flow framework of Chapter 3 to construct an object graph at compile time and then apply a graph embedding technique such as simulated annealing [36] or a heuristic algorithm [8] [23] to embed the object graph onto the ensemble. The advantage of this approach is that the best placement is known before the program is loaded. The primary disadvantage of this approach is that data dependencies and non-determinism in message-passing can affect the shape of the object graph. Thus, the object graph computed at compile time is at best a guess of the actual object graph. A second drawback is that the compilation time should not be comparable to the execution time; thus, only object computations that are expected to require considerable amounts of time to execute should be subjected to lengthy compile time analysis.

In practice, a combination of the two approaches will most likely be used. The compiler can be expected to yield a best guess of the object graph and the compiler can trade guess time against accuracy. Compiler derived information, e.g., the three factors, can always be quickly computed and made available to the runtime system. The runtime system can be expected to use a variety of object placement strategies based upon compiler derived information, information about the interconnection topology, and also more detailed information about message and object loads across the different parts of the ensemble. The runtime system can also relocate objects that are deemed to be long-term objects. This approach is appealing because the amount of overhead in placing an object can be made proportional to the importance of the object. For example, objects could initially be placed at random. The runtime system then keeps track of the message flow for each of the objects. The runtime system can then relocate those objects that process many messages and are known to be indefinitely persistent, e.g., EF $= \infty$.

Chapter 5

The Object Model Directed Architecture

Cantor and the object model serve as a basis for developing a wide range of software and hardware architectures. Software architectures for Cantor include implementations on existing concurrent computers, such as the Caltech Cosmic Cubes and Intel iPSC, and on sequential computers such as SUNs and VAXes. The Caltech Mosaic C project and the Cantor Engine project represent two ongoing hardware architecture projects for Cantor and the object model. Both of these hardware architectures are fine grain ensemble machines.

Concurrency is exploited in the hardware architectures when objects process messages concurrently. For example, the ensemble machine architecture of Figure 5.1 allows the placing of one or more objects on each of the processing nodes, and allows the nodes to process messages concurrently. Each object is an atomic component and is never distributed across multiple nodes. Concurrency can also be exploited by allowing multiple objects to execute concurrently within a node. These two sources of concurrency define the range of possible architectures for object computations. The degenerate architecture is the sequential computer, which consists of a single node capable of executing one object at a time. A shared-memory computer consists of a single node capable of executing multiple objects concurrently.

This chapter presents architectures for the node computer of an ensemble machine and also for shared-memory computers. Both architectures are organized upon a common data structure used to represent both messages and objects. Fault tolerant techniques for host object programs on large scale ensembles and the hardware support necessary for garbage collection are also presented.

5.1 Message and Object Dualism

An object model directed architecture, whether cast in software or hardware, must support substantial levels of dynamic storage management. This requirement stems from the late binding aspect of the message-passing semantics and from the queue based formulation of the formal semantics developed in Chapter 3. Storage management at the architecture level involves the allocation and deallocation of messages and of objects.

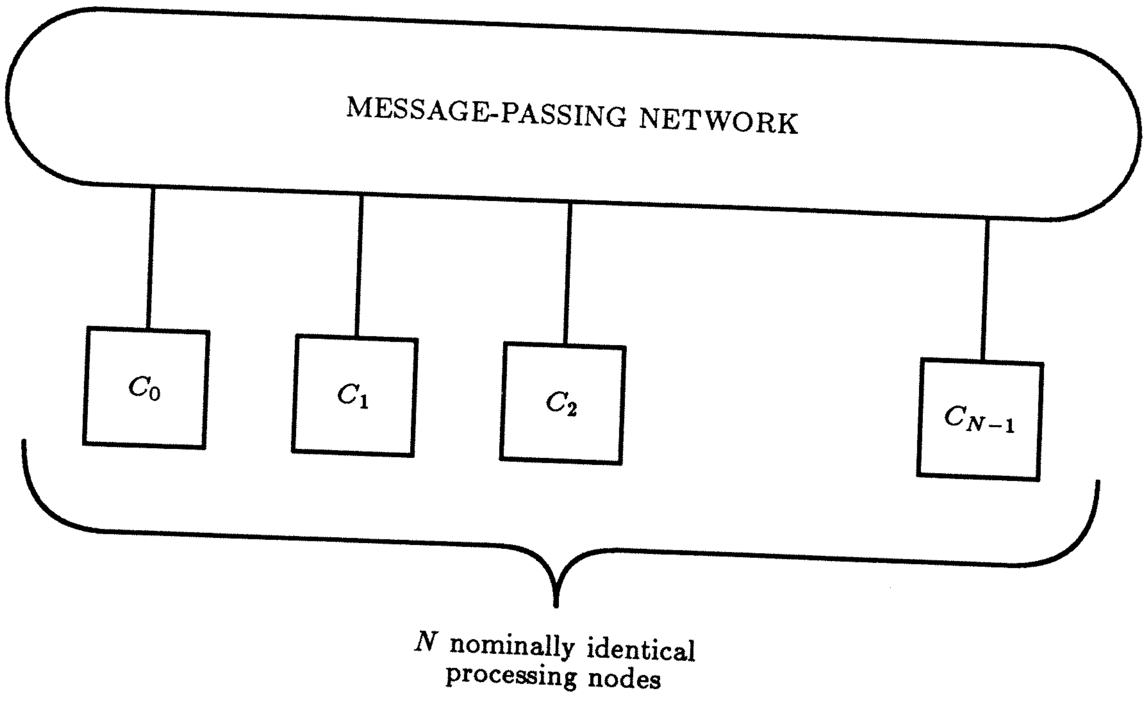


Figure 5.1: Ensemble Machine for Object Programs

The storage management problem is simplified by considering messages and objects to be semantic duals. Objects are stationary, reactive, and relatively persistent. Messages are mobile, inert, and relatively transitory. The behaviors of messages and objects are therefore complementary; however, their representations need not necessarily be different.

The following data structure conveys all the information necessary to represent either objects or messages and to distinguish between them:

destination
length
code pointer
value #1
value #2
⋮

The first three components of the data structure convey the "header" information necessary to describe the message or object. The remaining components are the message or persistent variables. The destination field identifies the object that is to receive the message. Length is a count of the number of data values contained in the message. Length is non-essential, but is useful for detecting programming errors if the object expects more data than is contained in a message. With these two components, messages can be represented. Objects require a third component, named the code pointer, to refer to the code sequence for the object. This third component can be regarded as a special persistent variable. A switch instruction will cause the code pointer to be modified.

The code pointer determines whether the message is to become an object when the message reaches its destination. By convention, all messages have the same code pointer, called NIL, and this code pointer is distinguished from any valid reference to a code sequence. The transformation between objects and messages is a one step operation. A new object is constructed as a message, but its code pointer is non-NIL.

Code pointers are examined on message delivery. If the code pointer is NIL, then the message is delivered as a normal message. If the code pointer is non-NIL, then the message becomes a local object. The message variables become the persistent variables for the object.

5.2 Generic Node Architecture

Central to the development of hardware architectures for object programs is the partition between message *processing* and message *management*. In the formal semantics, the management of messages is hidden by the use of sets, mapping functions, and k-tuples of values. For a hardware architecture, the effect of these mathematical objects must be explicitly supported by data structures and hardware mechanisms. The possible organizations for the node computers are endless. However, the hypothetical organization of Figure 5.2 captures the general data flow.

The control of the node computer resources is split between the Object Manager and Object Engine. The Object Manager is responsible for message management and the engine is responsible for message processing. Messages are input to the node from the Network Interface and are enqueued into the Receive Queue. The Object Manager moves messages from the front of the Receive Queue into the Backing Store. Before a message is moved, the code pointer of the message is inspected. Assuming only one object per node, if the code pointer is non-NIL, then the message becomes the object local to the node. The only step required to transform the message into an object is to record the pointer to the message as an object pointer. If the code pointer is NIL, then the message pointer is placed into the Input Queue.

From the semantics of the object model, the possibility exists that a message may arrive for an object before the node contains the object. For this case, the messages are queued for the object in the order in which they arrive. After the message arrives that contains the new object, the queued messages are moved into the Input Queue. The Input Queue and Output Queue are nominally not separate hardware structures but are data structures maintained in the Backing Store. Messages that have not been processed by the Object Engine are kept in the Input Queue. Because every message that is placed into the Input Queue will be processed regardless of its contents, the Input Queue serves a dual purpose; it keeps track of the unprocessed messages, and also determines their scheduling sequence for the Object Engine.

The Object Engine operates concurrently with the Object Manager and interprets an instruction stream similar to the primitive notation developed in Chapter 3. Upon interpreting a switch instruction, the front of the Input Queue is advanced and the message pointer at the new queue front is loaded into the Object Engine. Execution

Program	lines of code	min.	max.
Circuit Simulation	88	0	5
Eight Queens	106	0	4
Floyd's Shortest Path	472	0	12
Gaussian Elimination	136	0	8
Merge Sort	113	0	7
Perfect Numbers	24	1	3
Primes	45	1	4

Table 5.1: Minimum and Maximum Number of Variables Per Message

continues for the object at the instruction address contained in the switch instruction.

When the Input Queue front is advanced, the message pointed to by the old queue front is deallocated. Likewise, when a message has been completely absorbed into the message system from the Send Queue, the message contents are discarded. Incoming messages enqueued into the Receive Queue, and new messages and objects generated by the Object Engine, require new messages to be allocated.

These four operations require a high performance storage allocator. Traditional strategies such as the buddy-system [24] can be used. The simplest and fastest system, however, is to maintain a single list of free message buffers. The maximum size over all the objects and messages can be determined when a program is compiled. This maximum size becomes the size of each buffer on the free buffer list. Each message received by the Network Interface will fit into one buffer. Each new message or object generated by the Object Engine will also fit into one buffer. The allocation operation is one step: unlink buffer from the free list. Likewise, the deallocation operation is also one step: link buffer onto the free list.

This simple strategy is fast, but may be inefficient in the use of storage, if there is a large variation in the number of data values per messages. In practice, the variation is small. For example, the minimum and maximum number of variables per message for a sample set of programs is tabulated in Table 5.1. A promising variation on this strategy is to have the Object Manager pick a buffer size and format the free list accordingly. For the Message Interface, if an incoming message overflows a buffer, then a second buffer is used to contain the overflow. A single message becomes a multiple of message buffers. For the Object Engine, new messages and objects must be constructed out of multiple buffers when necessary. If the compiler is given the buffer size before a program is compiled, the compiler can generate code that treats each message and object as being distributed across multiple buffers. Each multiple buffer message is assigned a *display* [32] that contains a list of pointers to each of the buffers that comprise the message.

The primary advantage to this scheme is that the Object Manager can choose the buffer size independently of the compiler requirements. A second advantage is that messages received by the node never have to be reassembled, and both the Network Interface and Object Engine work on messages as groups of buffers.

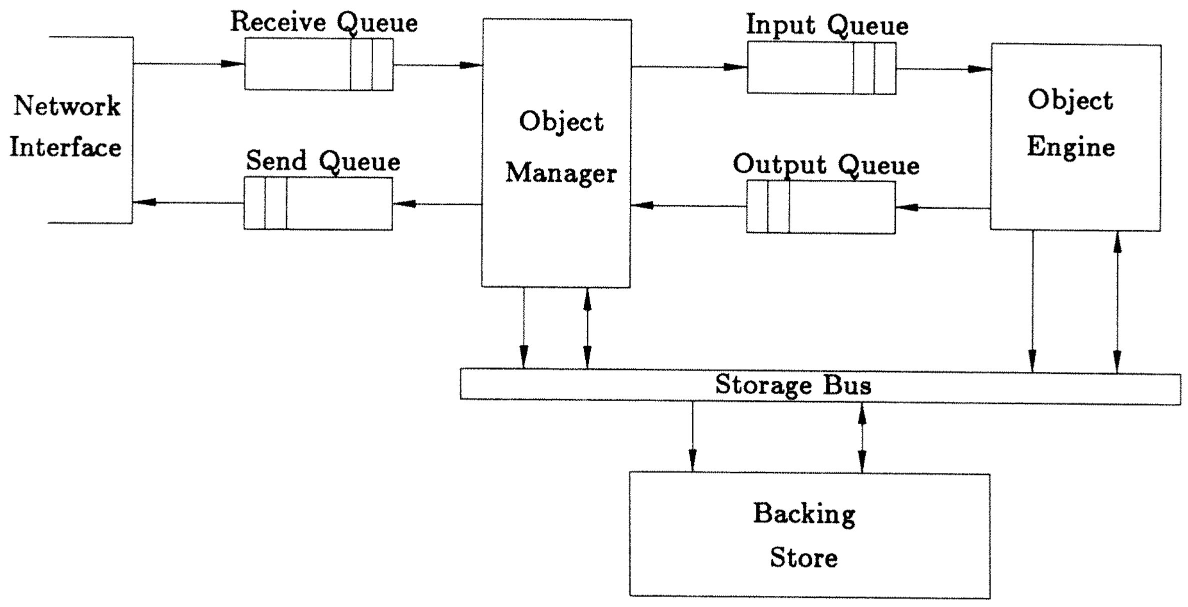


Figure 5.2: Generic Node Architecture for Single Object

5.3 Multiple Objects per Node

The assumption that each node computer hosts a single object is difficult to achieve in practice. For example, the Gaussian Elimination program of Chapter 2 will deterministically generate over 4,000 objects that persist for the duration of the computation. The obvious solution is to assign multiple objects per node, sacrificing concurrency if there are more concurrent objects than nodes, but potentially improving utilization of each of the nodes.

To manage messages for multiple objects, a table called the Object Table is added to the Object Manager. The Object Table maps reference values into message buffer pointers. A message at the front of the Receive Queue that has a non-NIL code pointer becomes an object local to the node by writing a pointer to the message buffer into Object Table. If the message is a normal message, then the destination reference value is used as an index into the table to locate the destination object for the message. This object pointer and the message pointer are enqueued as a pair into the Input Queue.

The operation of the Object Engine must also be modified, because the Object Engine is now shared among multiple objects. The code pointer must be saved when the Object Engine interprets a <u>switch</u> instruction. The code reference embedded in the <u>switch</u> instruction is stored as the code pointer for the object. After the code pointer is saved, the message pointer at the front of the Input Queue is deallocated, and the front of the Input Queue is advanced. The message and object pointer pair at the front of the queue are then loaded into the Object Engine. The Object Engine accesses the code pointer of the object to find the instruction location where execution is to begin.

Objects can be coded for self-destruction by embedding the NIL code pointer reference into the <u>switch</u> instruction. After the code pointer is saved, the two pointers at the front of the Input Queue have their code pointers checked. For each code pointer that is NIL, the associated message buffer is deallocated.

An available optimization to the Object Manager is to check the destination for messages in the Output Queue and immediately requeue messages into the Input Queue if their destination is local. This shortcut accomplishes the message passing action by moving the message pointer between the two queues.

An assumption built into this node organization for the ensemble machine is that the code for the objects is present wherever needed. The obvious way to accommodate this assumption is to place the code for all of the object definitions on all of the nodes. For fine grain ensembles, this approach will not be practical. One solution is to fetch code segments for objects on demand. When an object is to be installed on a node, a check is made by the Object Manager to see if the code segment is present. If not, a request message is sent to the other nodes to find a copy of the code segment. The code segment is then sent from another node as the reply to the request.

This strategy is sufficient, but has the undesirable property that creating a new object is delayed if the code is not present. An alternate scheme is to place the code for object definitions on subsets of nodes. The result is that code is *zoned* by object definition and node number. When a new object is created, the object definition for the new object is known. The object definition is used to consult a directory that gives a list of nodes where the code for the object definition is resident. The assignment for the new object is limited to the list of nodes in the proper zone.

The degenerate case for the ensemble machine is when there is a single node in the ensemble, i.e., the sequential computer. For this case, the *shorting out* of messages whose destination is the local node will ensure that the Network Interface is never used. An equivalent solution is to connect the Output, Send, and Receive Queue into a single queue. The check that the destination of the message is local can then be eliminated.

5.4 Multiple Object Engines per Node

For shared-memory computers all messages are local to the node, and the transferring of the message buffer pointers from the Output Queue back to the Input Queue is the mechanism for delivering messages. Figure 5.3 depicts one possible configuration for using a shared-memory computer. Each Object Engine represents one of the instruction processors that share the common memory or Backing Store via the Storage Bus. Each Object Engine has its own Input and Output Queue, and the operation of the Object Engines remains unchanged. The main advantage to this organization is that message-passing involves only moving a pointer between queues. The main disadvantage is the competition among the Object Engines and Object Manager for access to the Backing Store.

The shared-memory organization obviates the need for the Network Interface and the message-passing network. The problems of load balancing are not solved, however, by using shared memory. The operation of the Object Manager is complicated because it now has to map incoming messages and objects onto multiple Object Engines. The goal of this mapping is to minimize the contention for Object Engines among the objects hosted on the node. Ideally, every Object Engine is processing a message and has a non-empty Input Queue, indicating 100% current utilization and 100% immediate future utilization, respectively. Clearly if the number of concurrent objects is less than the number of Object Engines, then complete utilization is impossible. However, the situation may arise that the Input Queue for an Object Engine contains multiple entries, but the Input Queues for the other Object Engines are empty.

This situation does not necessarily indicate that the Object Manager has performed a poor choice of assigning message and object pointer pairs to Object Engines. The semantics of the object model defines the contents of the persistent variables of an object as a critical section. An object cannot be assigned to multiple Object Engines, because the contents of the object may be updated concurrently. A second problem is that message order cannot be preserved if an object is assigned to more than one Input Queue.

The message order restriction does not apply if the object is history-insensitive. From the results of Chapter 3, if each invocation of the object is assigned a separate area for temporary calculations, the object can be assigned to multiple Input Queues.

The enqueueing of a pointer pair into an Input Queue represents an assignment of the object to an Object Engine. The Object Manager must keep track of the mapping between objects and Object Engines, and ensure that the same object does not get mapped onto more than one Object Engine. The enqueueing and subsequent processing of the pointer pair by the Object Engine indicates an interval during which the assignment

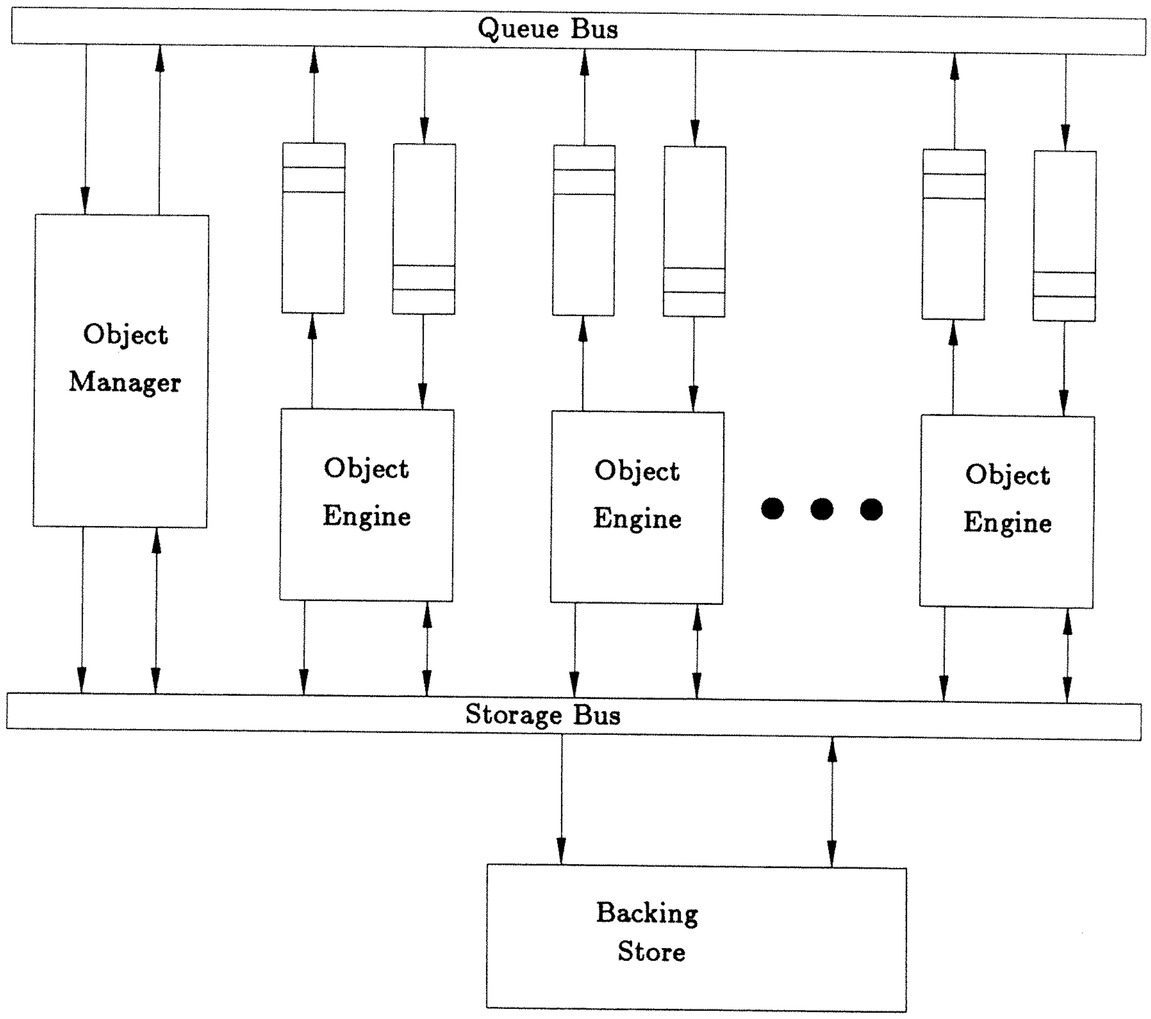


Figure 5.3: Multiple Object Engines per Node

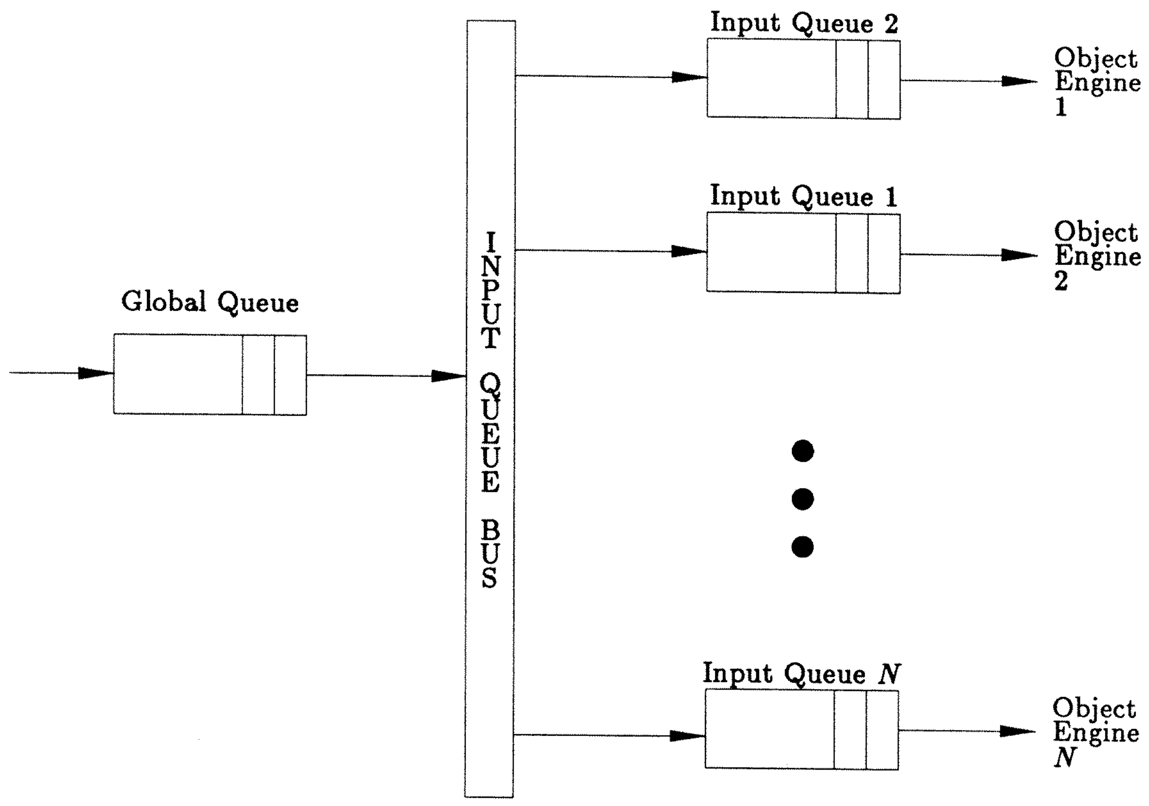


Figure 5.4: Global Queue and Input Queues

of the object is "locked" to the Object Engine. Enqueuing of additional pointer pairs for an object already locked to an Object Engine extends the interval, but eliminates both the synchronization with the unlocking of the assignment and the subsequent reassignment step.

The problem of assigning objects to Object Engines is comparable to the problem of assigning objects to processing nodes in the ensemble machine, except that locality is not an issue. The random placement strategy coupled with assuming that objects are *permanently* locked to an Object engine should be efficient for load balancing.

An alternative is to replace all of the Input Queues with a single Global Queue and to perform the assignment of objects to Object Engines on demand. After a message is processed by an Object Engine, the message and object pair at the front of the Global Queue are loaded into the Object Engine. This scheme is satisfactory for history-insensitive objects, but for history-sensitive objects, the message and object pair cannot be assigned to an Object Engine unless the object is currently not assigned to another Object Engine, i.e., the object is locked. For this case, the Global Input Queue must be searched from front to back to find a unlocked object pointer.

The searching of the Global Queue may impact Object Engine utilization, because access to the Global Queue must be inside a critical section. Multiple requests for message and object pointer pairs from the front of the Global Queue must be serialized. While an Object Engine waits for its turn to access the Global Queue, the engine remains idle.

The advantage of making an early, permanent assignment of objects to Object Engines is that message delivery and message processing can be overlapped. The potential drawback of this scheme is that some Object Engines can become idle due to a poor choice in the assignment. The advantage of message processing on demand is that the poor assignment problem cannot occur. Message delivery and the processing of messages cannot however be overlapped; i.e., an Object Engine is idle while it waits for its turn to access the front of the Global Queue. Multiple Object Engines attempting to access the Global Queue simultaneously can cause lengthy idle periods.

The minimum requirements for complete overlap of message delivery and message processing are dependent upon the relative speeds between message delivery time and message processing time. For example, assuming that the time to deliver a message is equal to or less than the time to process a message, complete overlap is obtained if each Input Queue always contains one or more messages. If the relative speeds can be determined, the best solution is to use a combination of the two strategies. The Global Queue can be used to feed the individual Input Queues as shown in Figure 5.4. Object and message pairs are transferred to the individual Input Queues by the Object Manager so that the minimum queue depth for complete overlap is maintained on a per queue basis.

5.5 The Cantor Engine

The Cantor Engine and the Mosaic C processor are two VLSI architecture experiments for fine grain ensemble machines. The Mosaic C is a general-purpose *two-sequence* processor. Once sequence handles Object Manager responsibilities while the other handles Object Engine tasks. The two sequences are interleaved on demand of the message-passing network. The Cantor Engine is a VLSI architecture for an Object Engine that directly implements the semantics of Cantor programs. The engine is partitioned into the Instruction Processor and the Bus Manager as shown in Figure 5.5. Because the Backing Store is shared by the Object Manager and one or more Cantor Engines, the data bus for the engine is expected to exhibit high latency. The Cantor Engine is therefore a *Harvard* architecture with separate instruction and data streams. Instructions and data constants are fetched from the Instruction Store which is local to each Cantor Engine and is read only.

To minimize the effects of latency when accessing the Backing Store, the Cantor Engine uses *hardware futures.* Each register of the Data Register File can be marked as a future. When the Instruction Processor executes a load or store instruction, the register for the instruction is marked as a future. The register number; effective address; and if the instruction is a store instruction, the data value in the register are enqueued into the Load/Store Queue. The Bus Manager dequeues load/store requests from the queue and then requests the use of the Storage Bus. After the load or store operation is completed, the Bus Manager uses the register number for the instruction to access the Data Register File and change the future register back into a normal register.

Instruction sequences that consist of a sequence of load instructions followed by a sequence of arithmetic operations and concluded by a store instruction will benefit from hardware futures by allowing data to be loaded while other data is computed on. The actual performance advantage will depend upon the latency of the Storage Bus relative to instruction execution time, and upon the success of the compiler at arranging the instruction code.

The Cantor Engine is a queue-based architecture in contrast to a stack-based architecture. The Instruction Processor uses five address registers: T,M,P,Q, and F for all address calculations. The T address register points to a scratchpad area in the Backing Store for saving temporary calculations that do not fit inside of the Data Register File. The M address register points directly into the Input Queue to access the contents of a message. The pointer stored in the M register is advanced when a switch instruction is executed. The P address register points to the buffer that contains the persistent variables of the object. The Q register points into the Output Queue and is advanced upon a send instruction. For the Cantor Engine, the Q register serves a dual purpose. The Q register points to the end of the Output Queue but also points to the front of the free message buffer list. The Object Manager initializes the Q register with a pointer to the free buffer list by the Object Manager. The Cantor Engine writes messages directly into the buffer at the front of the free list. After the message is completely written, the Cantor Engine advances the pointer stored in the Q register. The Object Manager can send all messages between the front of the Output Queue and the pointer stored in the Q register. The Object Manager can also add more free buffers to the end of the free list.

5.6 Fault Tolerance

For a given node architecture, the probability that a node computer will fail increases

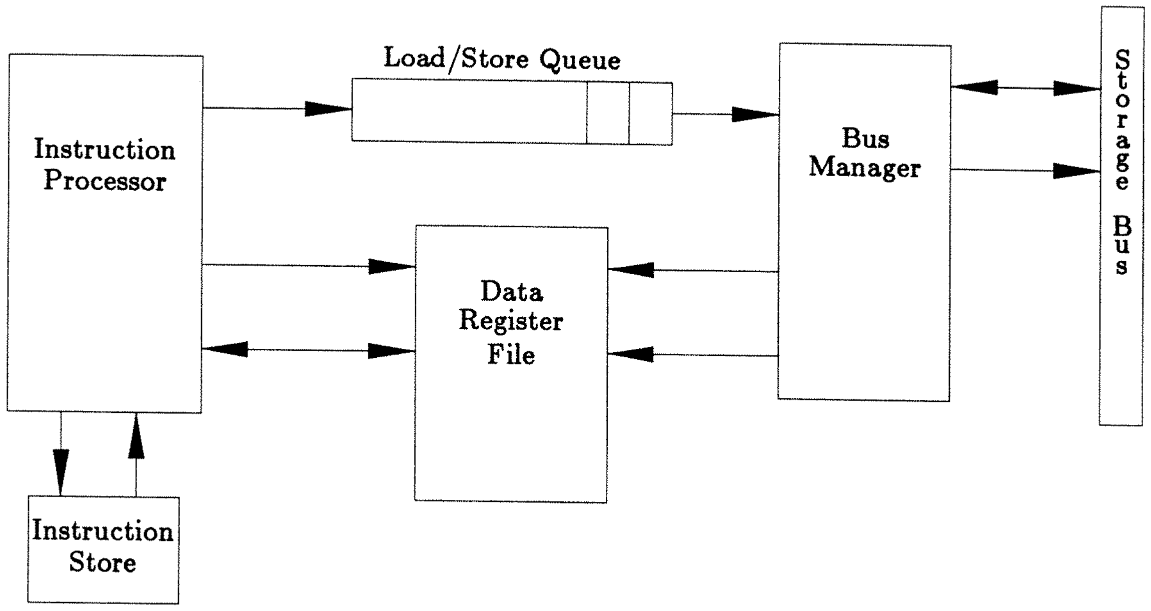


Figure 5.5: The Cantor Engine

Channels and Network Interface	12,500
Datapath	7,000
Microcontroller	2,500
RAM (4T design)	512,000

Table 5.2: Transistor Count (approximate) Breakdown for Mosaic C with 16KB

at least linearly with the number of nodes used by the object program. For applications where faults cannot be tolerated, or where the mean time between failures becomes impractically small, steps must be taken either to improve the reliability of the nodes or to find methods for allowing computations to continue in the presence of faults. All methods for fault tolerance require adding redundancy at some level in the system. The redundancy can be introduced over a wide range of levels, from machine words to objects and messages. An example of redundancy at the level of machine words is the use of error correcting codes to detect and correct bit errors in storage chips.

Adding redundancy at the message and object grain size is dependent upon where faults are expected to occur. In the block diagram of Figure 5.2, errors can be expected to occur in the message-passing network, Object Manager, Object Engine, and Backing Store. Table 5.2 shows the approximate transistor counts for the Mosaic C single chip computer. If the probability of failure is uniform over all the active devices, Table 5.2 clearly shows that most failures will occur in the RAM, i.e. the Backing Store. However, mechanical failures such as bad connector contacts have a much higher probability than transistor failures.

For single chip nodes, connector failures will primarily affect the message system. There are three types of errors that stem from the message system:

- The message is received at the wrong destination.
- The contents of the message are corrupted.
- The message is destroyed in transmission.

A message received at the wrong destination cannot simply be forwarded to the correct destination because the property of message order could be violated. The one exception is when the destination is a history-insensitive object. For the general case, a sequence number included with each message can be used to reconstruct message order at the node level. The sequence numbers are assigned in a monotonically increasing order so that each node can calculate the expected next sequence number from every other node. Two tables are maintained by the Object Manager: a receive table and a send table. Before a message is sent, the destination node number for the message is used as an index into the send table. The result of the table look-up is the sequence number. This number is stamped onto the message, and the next sequence number is written back into the send table.

When the message is received, the node number of the sender is used as an index into the receive table. If the sequence number of the received message matches the sequence number stored in the receive table, then the message is delivered and the next sequence number is written back into the receive table. If the sequence numbers do not match, then the message was delivered out of the expected order. The destination Object Manager must place all out of sequence messages from a node into a special queue. These messages cannot be delivered until the missing messages are delivered.

If the Object Manager receives a message for a node other than itself, the wayward message is forwarded to the correct destination. Although the message may arrive out of sequence at its correct destination, the messages that are out of sequence will be placed in their proper order when they reach their destination.

The send and receive tables can be organized by object reference value instead of node number. This organization is complicated by the vacillating nature of the reference connectivity between objects. The connectivity of the nodes is static; thus, the set of possible senders and receivers for messages is limited to the set of nodes. To organize the ordering of messages by object, the send and receive tables are each replaced by a list and there is a send and receive list for each object reference value. When a message is sent by an object, the send list is searched for the appropriate sequence number. If the destination reference value is not in the list, then an entry is added to the list for the new destination. Likewise, when a message is received, the receive list is searched. If the sender reference value is not in the list, the sender is added to the receive list. The very first sequence number must be a global constnat so that all objects can recognize the first sequence number from a previously unknown object.

If each reference value has its own send and receive list, then the reference values can be recycled without affecting the sequence numbers. In principle, each reference value could have a send and receive table with a table entry for each of the possible reference values. In practice, these tables are too large, so a spare representation, viz. lists, are used and are incrementally updated to include additional entries.

The second type of message system failure is when a message arrives at the correct destination but the contents of the message have been corrupted. The first step is to detect that the message is corrupted. This step can be accomplished using either horizontal parity: adding check bits for each value in the message, or vertical parity: adding check bits for the entire message. Horizontal parity can be expanded so that the values in the message become codewords of an error correcting code. The codewords are decoded and corrected inside the message. Likewise, vertical parity can be expanded to include additional values for the message contents. For this scheme, the values contained in the message are the components of a codeword for an error correcting code, for example, Reed-Solomon codes [31]. The message is decoded and corrected in place.

If the message does not contain sufficient redundancy to correct an error, the necessary redundancy must be introduced by sending additional messages. One simple scheme is to acknowledge every message received. The sending node does not discard the original message contents until the acknowledge is received for the message. If a message is corrupted, a retransmit request message is replied instead of an acknowledge message. The sender must then send the message again and wait for the acknowledge. The acknowledge and retransmit messages are short, potentially only two bits long.

If the message received does not contain sufficient redundancy to detect an error, multiple copies of the message must be sent to increase the probability that the message is error free. This strategy is by far the simplest. Each message is sent n times and the contents of the n copies are compared. For $n = 2$, errors can be detected and for $n = 3$, a majority vote can be taken to decide upon the correct contents of the message. Many other schemes are possible by using encoded messages and multiple copies of the message to calculate a *best guess* for the message contents.

The third type of error is when a message is lost, never to be heard from again. This type of error is the most difficult since message passing is asynchronous in the object model. Because messages can be completely absorbed into the message-passing network, no node can claim ownership of the message until it exits the network. The probability of losing a message can be reduced by sending each message n times. The receiver expects to receive n copies of the message. After the first copy is received, the message is processed and all subsequent copies are discarded.

By explicitly acknowledging each message sent, the destination node for a message can request a second copy of a message. The sender can adopt a similar strategy for initiating the sending of duplicate copies of a message it suspects to have been lost. If the sending node does not receive an acknowledge or retransmit request for a message after a *reasonable* period of time, the sender retransmits. The destination node can also initiate a retransmit if a message arrives out of sequence. In either case, the destination node must be prepared to receive two copies of the same message, one copy from the original wayward message and the second from the retransmission. For messages that are truly lost, the first message will never arrive, but the nodes cannot determine this condition, unless all of the nodes involved in the computation are queried and the message-passing network is quiescent.

Faults inside of the Backing Store can potentially cause faults inside of the Object Manager and Object Engine. Assuming that a fault in the Backing Store can be detected, the Object Manager must rebuild the computation prior to the fault and then continue the computation. Rebuilding the computation state requires keeping duplicate copies of messages and objects.

A property of the object model is that an object will use finite resources to process a message. The maximum number of instructions executed, messages sent, and new objects created in response to processing a message is known before the message is processed. The node organization of Figure 5.6 exploits this property by providing the Object Engine with its own private Object Store. For an object to process a message, the contents of the message and object are copied into the Object Store. After the message has been processed, the new messages and the modified object are copied out of the Object Store and into the Backing Store.

If the Object Engine encounters a fault and the Object Manager can detect the failure, no harm is done to the computation state, because the object and message are copied into the Object Store prior to execution. If the Object Engine experiences a transient fault, the message and object contents are reloaded into the Object Store, and the Object Engine can attempt to process the message again. If the Object Engine is permanently broken, all of the objects and messages local to the node have to be relocated. Each object local to the node is relocated to another node. The location for each of the relocated objects is stored in the Object Table so that future messages are forwarded to the new destinations. To preserve message order, the messages local to the node must be first forwarded to their new destinations followed by all messages that subsequently arrive. If the indirection through the Object Table is circumvented, then message order cannot be preserved.

An Object Engine that fails to execute a switch instruction, e.g., the Object Engine is in an infinite loop, can be detected because the maximum number of instructions that an object interprets to process a message is bounded. The Object Manager monitors the number of instructions executed by the Object Engine and aborts the execution if the maximum number is exceeded.

If the Backing Store encounters a fault, and the Object Manager can detect the failure, and if the fault was transient or the node can continue in the presence of the fault, the Backing Store must have its contents restored. For the Backing Store to be rebuilt, the objects and messages of the computation must be systematically duplicated for every message processed. After a message is processed by an object, a copy of each message produced is sent to a different node. The contents of the modified object is also sent to another node as a back-up copy. After acknowledge messages are received confirming that the copy of the modified object has been saved and that the copies of the new messages have also been saved, the original message and old object contents are discarded. Two delete messages are sent to discard the non-local copies of the old object and original message.

To rebuild the state of the Backing Store, messages are sent to the nodes containing the back-up copies of the local objects. These messages request that the back-up copies be sent to the originating node. After all of the the back-up copies have been received and the message copies have been reordered, the computation can continue. If the Backing Store is permanently broken and the Object Manager cannot operate in the presence of the fault, the objects and messages have to be moved to other nodes. The nodes containing the back-up copies of the local objects are sent special messages requesting that the copies of the objects be changed into instances of the objects. The location of each relocated object must be stored in the Object Table of the originating node. The nodes that contain copies of the messages are sent messages requesting that the message copies be changed into messages and sent to the new destinations. Because message order is not preserved for this case, the Object Manager must support message reordering. A caveat is that the Object Table as well as the tables used to keep track of the back-up copies for the objects and messages must be kept in a *highly reliable* storage area that is separate from the Backing Store.

If the Backing Store or Object Engine faults and the Object Manager does not detect the error, or if the Object Manager faults, pandemonium may result. Faults in the Object Manager can be catastrophic. An Object Manager running amok can destroy objects and messages, forge reference values, and wantonly send messages. For environments where these types of faults can occur, the duplication of the state of the messages and objects is insufficient. The processing of the messages by the objects must also be duplicated.

The general idea is to duplicate the computation several times and compare the results. For example, three concurrent computers start with the same object program if two of the three computers agree on the result, then the disagreeing machine is considered to be faulty. This principle can be applied to the node computers at the granularity of objects and messages, providing there is a reliable agent that can compare the results of processing a message and render a judgement. For example, assume that the comparison is performed upon message delivery. The object program is performed in triplicate with each object having two identical counterparts in the two other program copies. Reference values are triple references so that each message is sent to three objects, one object per program copy.

Messages are tagged with sequence number, sender ID, and also the number of the program copy of the sender. Before a message is processed by an object, two of the three copies must be received. If the contents of the two messages agree, then the message can be processed and the third message is discarded when it arrives. If the two messages do not agree, then the third message is used to decide which of the two earlier messages is correct.

The treatment for new objects is identical, except that the comparison operation is performed by the Object Manager when it attempts to install the message as a new object. The overhead in redundancy for this scheme is that each message and object is replicated three times. The error correction capability covers the loss or corruption of a single message per object and the loss or corruption of a single object. The error correction capability can be expanded by using a more complex comparison operation so that a best guess for a message or object can be made when no two of the three message copies exactly agree.

Correcting faults on the fly when the object program has been triplicated and the three program copies are running concurrently is less complicated than the case where the object program has been triplicated but only one copy is running. Both cases require comparable amounts of storage, and the latter is less demanding on computational resources. The former, however, as is often the case with concurrent programs, has a less complicated formulation.

5.7 Garbage Collection

Hardware or software support for garbage collection must be included in the object model directed architecture. The primary difficulty with garbage collection is related to the problem of detecting messages that have been lost in transmission. From the algorithms of Section 3.6, the Object Manager on each node must be able to collectively determine when the number of white or black messages used by a program has reached zero. If the message system is capable of completely absorbing messages then, like lost messages, the presence or absence of messages cannot be determined exclusively by the nodes. For example, assume the system contains a single white message. After the message is placed into the message system, the sending node reports that its local white message count has reached zero. Meanwhile, the destination node also reports a zero white count. Until the white message arrives at the destination node, all of the nodes report that there are no white messages remaining.

The source of the problem is that the state of a computation is comprised of two parts: the state of the objects and the set of undelivered messages. If undelivered messages can be *competely* hidden inside the message system, then no guarantee can be made about the global state of the computation.

A software solution to this problem is to layer the network interface software with a request/acknowledge protocol. With this protocol, every message injected into the mes-

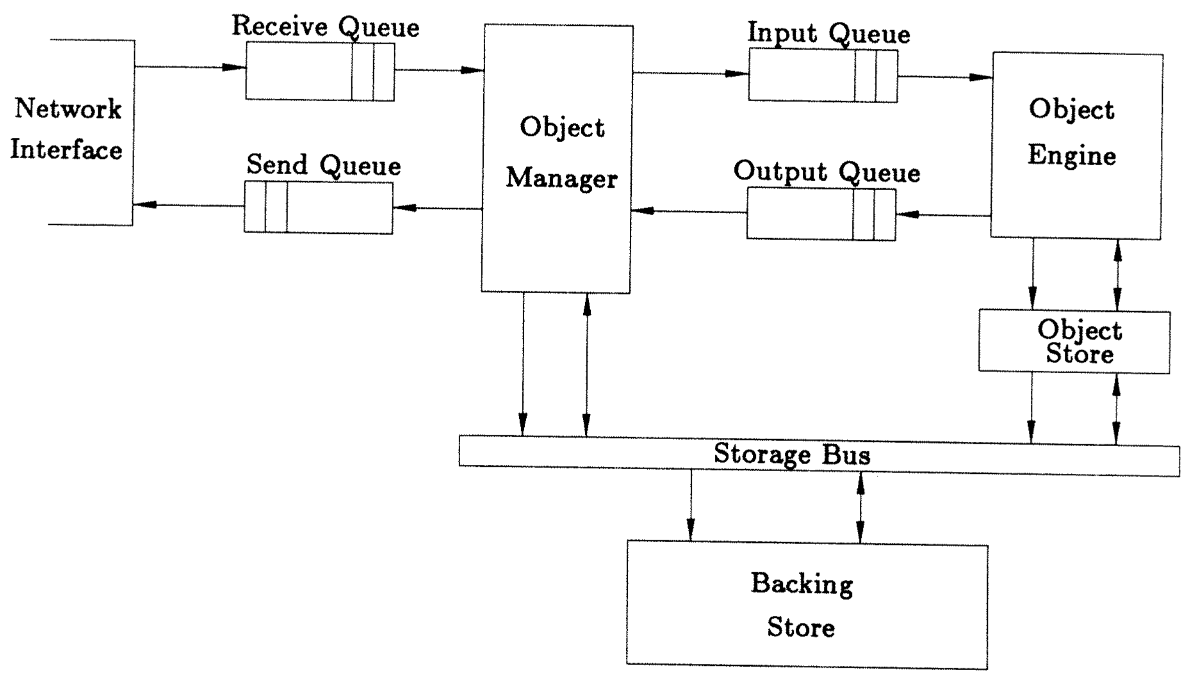


Figure 5.6: Fault Tolerant Node Organization

sage system is accounted for. The solution proposed is an adaptation of an algorithm for termination detection by E.W. Dijkstra and C.S. Scholten [17]. The following discussion applies equally to white and black messages. Messages are of two types: normal data messages and acknowledge messages. Each node maintains a count of the number of data messages sent but not yet acknowledged. Initially, all nodes have message counts of zero and are waiting for their very first message.

Receipt of the first data message by a node is *not* acknowledged; rather, the sender node number of the first message is kept in a special storage location. For each subsequent data message received by the node, the sender is immediately sent an acknowledge message. For each data message sent by the node, the local message count is incremented. For each acknowledge message received by the node, the message count is decremented. When the message count reaches zero and the node has no additional messages of the color to process, the acknowledge message is sent to the sender of the very first data message.

Initially the nodes are idle, waiting for their very first message. The environment external to the nodes initiates an object computation by sending a message to one of the nodes. The node that receives the message from the environment stores the environment ID as its very first sender. If the node does not send any messages, then an acknowledge message is sent to the environment; otherwise, the message count is positive and other nodes are sent their very first message. These nodes in turn save the sender ID, i.e., the node number that received the message from the environment. Once the production of messages is stopped, i.e., the computation terminates or the color of new messages and new objects is changed, the message counts will strictly decrease. When a message count for a node reaches zero, the node has accounted for all of the messages it has sent for the old color, and all of the nodes that it has engaged also have message counts of zero for the old color. The node can then send an acknowledge message to its very first sender.

Initially, the nodes and the environment must agree on the first color. After the environment sends the first data message to a node, the environment can broadcast a change of color message. When the acknowledge message is received by the environment, all messages for the previous color have been accounted for. The environment then broadcasts the reclamation message to each of the Object Managers to reclaim all of the unreachable objects.

If the message system could be probed to check for the presence of black or white messages, then the state of the computation relevant to garbage collection would no longer be hidden by the message system. For ensemble machines where each node contains a routing automaton to route messages through the node, the routing automaton must be able to interpret header information for the messages. Message color can be embedded in the message header.

A hardware solution that does not require the above software layering is to share a global wire between the routing automata and node computers. The wire is used to compute the "wired-AND" function. The wire is passively pulled high but actively pulled low. Each node pulls the wire low if the number of messages present on the node is greater than zero. Each routing automaton pulls the wire low while it is routing a message through the node.

The garbage collection algorithm requires two global wires: one for white messages and one for black messages. Both the nodes and routing automata can pull either wire low. The white wire is pulled low by a node if the white count is greater than zero, and the black wire is pulled low if the black count is greater than zero. For each routing automaton, the white wire is pulled low if a white message is being routed, and the black wire is pulled low if a black message is being routed. The environment senses the output of the two wires. When one of the wires becomes high, the environment broadcasts the reclamation message.

The use of the wired-AND function was originally proposed for use in ensemble machines by Dick Lang [25] for termination detection of the marking phase of his mark and sweep garbage collector.

Chapter 6

Epilogue

The chapters of this thesis have been organized to support the claim that fine grain ensemble machines can be used as general purpose computing engines. The thesis starts with an informal description of a new programming notation, Cantor, and a collection of six complete programs that employ up to thousands of fine grain objects. A formal semantics of Cantor is then developed to provide a rigorous basis for the object programs. The semantics provide formal definitions for the internal behavior and interaction of objects. The internal behavior of an object is represented by a finite automaton. The semantics of message-passing and object creation are made precise. The stepwise interpretation of object programs using the formal semantics is presented. From the interpretation, the pre-computing of object graphs via futures and garbage collection are explained.

The dynamics of the concurrent programs are revealed under several simulated environments. The results of the experiments show remarkable resilience of the programs under realistic constraints of finite resources. The positive results of the simulations, combined with the formal semantics, form a basis for proposing several machine architectures for supporting object programs. The properties of the computational model are used advantageously to reduce the complexity of the architectures for program execution, fault tolerance, and garbage collection.

6.1 What Has Been Learned

The scientific challenge to using fine grain ensemble machines for general purpose computing is in writing programs that are organized into small objects that use only uncomplicated sequences of message-passing. Cantor represents one such approach. For better or for worse, Cantor carefully draws a fine line between the responsibilities of the program writer and the responsibilities of the compiler and runtime system. Placement of objects to minimize contention among resources, i.e., load balancing and locality, are the purview of the compiler and runtime system. The implication for the programmer is that once the program is written, the programming task is completed.

Cantor programs are chameleons that can be hosted on a myriad of machine organizations. The compiler and runtime system cooperate to provide a good match between the program and the target machine. The compiler generates machine code, gathers ancillary information about object behavior, and also builds object graphs. Building accurate object graphs is a hopeless task in the general case; however, speculation on the shape of the object graph can be done efficiently and is harmless.

The experiments with program behavior under simulated conditions show that for load balancing, random placement is an excellent strategy. Centralized schedulers that support assignment of work loads to processors on demand, as would be used in a shared memory computer, are unnecessary provided the number of concurrent objects is much larger than the number of nodes, or is much smaller than the number of nodes. Optimizing object placement for load balancing is unnecessary and could increase program execution time because of the overhead of using an elaborate placement algorithm.

The results for locality are less decisive. The n-cube placement strategy of assigning objects to nodes in order of increasing geometric distances performs best overall for locality with minimal impact on load balancing. The compiler-assisted approach that uses both random and n-cube placement achieves better load balancing than n-cube placement, and achieves more locality than random placement.

The conclusion is that the object programs exhibit remarkable resilience to the limitation of both processor and message resources. For load balancing, the simple assignment strategy of random placement is efficient. The use of complex algorithms for load balancing is not justified and could lengthen program execution time because of the overhead for placement. Whether the complex algorithms are useful depends upon the performance of the message-passing networks. The importance of locality increases as the message-passing resources become scarcer relative to the number of nodes, or if the cost of using the message-passing resources is substantial relative to the compute time of the objects.

Once a program is expressed as a collection of fine grain objects, the exploitation of the concurrency is straightforward. The cognitive process of expressing a program as a collection of fine grain objects is less than straightforward and, unfortunately, is often obstructed by Cantor. The deficiencies of Cantor are addressable at three separate levels: the programming model, the programming notation, and the programming system.

Starting with the programming model, the inability of objects to exercise discretion in receiving messages often introduces complicated message flow. For example, the interpretation of functional or procedural abstraction as a sequence of message-passing actions is lacking. Viewing a procedure call as sending a message to a new object is acceptable. However, waiting for the reply from the called object is complicated because there is no assurance that the next message received by the object will be the value replied from the called object. The calling object cannot focus upon receiving a particular message unless the calling object buffers all the messages that are received out of sequence.

A second solution is to place a queue object between the calling object and all of the objects that can send messages to the calling object. The calling object sends and receives messages exclusively from the queue, and the queue object is responsible for buffering messages that are received in the wrong order. Both solutions introduce explicit queueing that would otherwise have to be included in the runtime system. The present runtime system only needs to introduce queueing in the case of messages arriving for an object faster than the object can process the messages. The decision to keep the buffering explicit was part of an experiment to determine whether a thorough understanding of the message flow could always be applied to solve the buffering problem more elegantly and efficiently than using a generalized buffering mechanism. For the six test programs, by cleverness or good fortune, a better way was always found, but the toll on the programmer was often severe.

As a programming notation, Cantor suffers mainly from deficiencies in control flow. The first deficiency is that objects cannot be created running; rather, each object is created waiting for a message. The problem is solved by immediately sending the new object a message. The second deficiency is that messages are only received at the beginning of a description. This problem is related to the first and often challenges the program writer to find a method for expressing the description without sending redundant messages. Interestingly, the lack of an iteration construct in the notation did not create much difficulty except for the case of interleaving iteration with receiving other messages.

Cantor as a programming system desperately needs a set of external objects for input and output. The two external objects of **console** and **chessboard** are only a beginning. The Gaussian elimination program slyly initialized the matrix as it built the matrix. In real life, the matrix would be loaded with real numbers read in from a file. Files, terminals, windows, etc. can be interpreted as custom objects. All of these objects do not have to be sent to the **main** object when the program is started; rather, only one special object reference need be included. This object is used as a gateway to introduce all additional external object references by allowing the gateway object to send messages to the internal objects.

6.2 Beyond Cantor

Since program writing is the key to fine grain concurrent computing, the next set of experiments should center on writing large scale programs. Some of the programming projects that have been proposed are a Cantor compiler, text editor, and a circuit simulator fashioned after the program *Concise* [27]. The starting point for these programming projects is Cantor and the rudimentary programming techniques developed in this thesis. The mettle of Cantor can be tested from these program experiments and its fate decided, namely whether Cantor should emerge as a programming system or should become submerged as the runtime kernel of another programming system. The latter approach is underway with the development of the Reactive Kernel, a small, layered operating system for the second generation "cubes."

A proof theory for reasoning formally about object programs and proving their correctness is paramount. The development of a proof theory is challenging because of the non-determinism that is intrinsic to the message-passing actions. Proving properties about the internal behavior of objects borders on the trivial because of the finite automata representation. The interesting dynamics of object programs occur in the interaction between objects. This interaction is subject to the non-determinism of the message-passing semantics.

The current strategy for reasoning about program correctness is to trace through all possible sequences of message delivery and show that all cases are properly accounted for. This approach is not systematic, and because the interaction complexity increases combinatorically with the number of objects, it is prone to oversight.

The simulation environments of Chapter 4 proved to be helpful during program development by profiling program execution. Future simulation tools of this type that can be used on both sequential and concurrent computers will most likely be mandatory for developing programs. The key observation is that the multi-dimensional dynamics of concurrent programs require that runtime resources be evaluated quantitatively so that unsuspected bottlenecks can be discovered. The application of a program profiler that identifies all available concurrency will also uncover the bottlenecks that limit concurrency.

In addition to program development, there are many more system ideas that can be tested on the object programs. This thesis experimented with simple placement strategies and the effects of limiting message throughput. A myriad of other experiments are ready to be conducted including experiments to measure the effects of different message latencies and experiments to use adaptive routing algorithms for delivering messages.

For improving program performance, the frameworks for flow analysis presented in Chapter 3 should prove useful for developing advanced compilers. The drawback to these high performance compilers is the necessity of the closed world assumption. For interactive programs where the input to the program is unknown, the compiler must assume the worst about the input. From the property lattices, these worst case assumptions tend to propagate through all of the program code. For applications where the closed world assumption can be made, viz. the program input is available at compile time, then the object definitions can be compiled down to finite state machines and the compiler can partially execute the programs to build an accurate object graph. This type of compile time analysis is undoubtedly costly. However, to spend one week compiling a program to achieve a 20% decrease in runtime is worthwhile if the program is expected to run for more than a month.

To improve program performance where locality of reference is important, more research into placement strategies is justified. The first step is to devise fast and efficient algorithms for embedding the static object graphs built by the compiler onto the target ensembles. For the general case, this problem is NP-complete. Embeddings within 20% of optimal, however, will probably be acceptable. Furthermore, the graphs that are being embedded are not random graphs. The compiler generated graphs will have some regular structure and the target ensemble will most likely have a completely regular structure.

Further experimentation with dynamic placement strategies should include using both compile time and runtime information to place new objects, and possibly the relocation of objects to improve locality. The overhead of the placement strategy should always be analyzed because all objects are not created equal. The priority or importance of an object is dependent upon program semantics; using a costly placement strategy for a sequential object that receives a single message and then self-destructs is difficult to justify. The application of a quick placement algorithm followed by optimization of the placement of objects that prove to be important is conjectured to be a basis for many efficient strategies.

Finally, the node architectures of Chapter 5 require a great deal more thought. The semantics of object programs requires both message processing and message management. The partition between processing and management is blurred in the semantics, but must be carefully defined when the semantics are cast into hardware. The program semantics provide only a vague definition for the object manager, namely that it must manage the flow of messages between the message-passing network and the object engine. Speed matching between the object engines and message-passing networks requires further research into the organization of the object managers.

If the large programming projects are successful, the need for fault tolerant node architectures will increase. Fortunately, the semantics of object programs simplify the requirements to support fault tolerance. The fault tolerant architecture proposed in Chapter 5 exploited the advantages of the program semantics. However, the proposal is far from a complete solution.

In conclusion, this thesis has investigated fine grain concurrent computing as the basis for a model of computation that meshes well with the future of ensemble machines. The experimental results show that non-trivial programs can be written as collections of small objects and that once the programs are written, exploiting the concurrency is not difficult. Exotic hardware to manage the concurrency is therefore not justified. Further research into fine grain concurrent computing should concentrate on the methods for expressing programs. The presence of application programs will motivate and be useful for evaluating the other research topics related to fine grain ensemble machines.

Bibliography

[1] H. Abelson and G. J. Sussuman, *Structure and Interpretation of Computer Programs*, The MIT Press, Cambridge, Mass., 1985.

[2] G. A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Artificial Intelligence Laboratory, Technical Report 844, June 1985.

[3] A. V. Aho, J. E. Hopcroft, & J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Mass., 1974.

[4] A. V. Aho & J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Mass., 1977.

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

[6] H. G. Baker, *Actor Systems for Real-Time Computation*, MIT Artificial Intelligence Laboratory, Technical Report 197, March 1978.

[7] H. G. Baker, Jr., *List Processing in Real Time on a Serial Computer*, Communications of the ACM, Vol. 21, No. 4, pp. 280-293, Apr. 1978.

[8] E. R. Barnes, *An Algorithm for Partitioning the Nodes of a Graph*, SIAM Journal of Alg. Disc. Meth., Vol. 3, No. 4, pp. 541-550, Dec. 1982.

[9] C. Berge, *The Theory of Graphs and its Applications*, John Wiley & Sons, New York, 1962.

[10] W. D. Clinger, *Foundations of Actor Semantics*, MIT Artificial Intelligence Laboratory, Technical Report 633, May 1981.

[11] W. J. Dally & C. L. Seitz, *The Balanced Cube: A Concurrent Data Structure*, Dept. of Computer Science, California Institute of Technology, Technical Report 5174, May 1985.

[12] W. J. Dally & C. L. Seitz, *The Torus Routing Chip*, Journal of Distributed Computing, Vol. 1, No. 4, 1986.

[13] A. J .T. Davie and R. Morrison, *Recursive Descent Compiling*, Ellis Horwood Limited, Chichester, 1982.

[14] P. J. Denning, *Virtual Memory*, Computing Surveys, Vol. 2, No. 3, pp. 153-189, Sept. 1970.

[15] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall Inc., Englewood Cliffs, N.J., 1976.

[16] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, & E. F. M. Steffens, *On-The-Fly Garbage Collection: An Exercise in Cooperation*, Communications of the ACM, Vol. 21, No. 11, pp. 967-975, Nov. 1978.

[17] E. W. Dijkstra and C. S. Scholten, *Termination Detection for Diffusing Computations*, Philips Research Laboratories, EWD687, Eindhoven, The Netherlands, Oct. 1978.

[18] C. C. Elgot and A. Robinson, *Random Access Stored Program Machines*, Journal of the ACM, Vol. 11, No. 4, pp. 365-399, 1964.

[19] R. K. Guy, *How to Factor a Number*, Proceedings of the Fifth Manitoba Conference on Numerical Mathematics, Utilitas Mathematics Publishing Inc., Winnipeg, Oct. 1975.

[20] Intel Scientific Computers, *iPSC User's Guide*, Order No. 175455-001, 15201 N.W. Greenbrier Parkway, Beaverton, Oregon, Aug. 1985.

[21] K. Jensen and N. Wirth, *Pascal User Manual and Report*, Springer-Verlag, New York, 1974.

[22] R. M. Karp and R. E. Miller, *Parallel Program Schemata*, Journal of Computer System System Sciences, Vol. 3, No. 2, pp. 143-147, 1965.

[23] B. W. Kernighan & S. Lin, *An Efficient Heuristic Procedure for Partitioning Graphs*, The Bell System Technical Journal, February, 1970, pp. 291-307.

[24] D. E. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, Volume 1, Addison-Wesley, Mass., 1969.

[25] C. R. Lang, Jr. *The Extension of Object-Oriented Languages to a Homogeneous, Concurrent Architecture*, Dept. of Computer Science, California Institute of Technology, Technical Report 5014, May 1982.

[26] B. R. Locanthi, *The Homogeneous Machine*, Dept. of Computer Science, California Institute of Technology, Technical Report 3759, Jan. 1980.

[27] S. Mattison, *Concise, a Concurrent Circuit Simulator*, Dept. of Applied Electronics, University of Lund, Aug. 1986.

[28] S. S. Muchnick & N. D. Jones, *Program Flow Analysis: Theory and Applications*, Prentice-Hall, N. J. 1981.

[29] S. S. Muchnick & N. D. Jones, *A Flexible Approach to Interprocedural Data Flow Analysis and Programs with Recursive Data Structures*, 9th Annual Symposium on the Principles of Programming Languages, pp. 66-74, Jan. 1982.

[30] R. H. Halstead., Jr. *Multilisp: A Language for Concurrent Symbolic Computation*, MIT Laboratory for Computer Science, Cambridge, Mass., July 1985.

[31] R. J. McEliece, *The Theory of Information and Coding*, Addison-Wesley, Mass. 1977.

[32] E. I. Organick, *Computer System Organization: The B5700/6700*, The Academic Press, San Francisco, 1973.

[33] C. L. Seitz, *Concurrent VLSI Architectures*, IEEE Transactions on Computers, Vol. C-33, No. 12, pp. 1247 1265, Dec. 1985.

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

[35] C. L. Seitz, *Experiments with VLSI Ensemble Machines*, Dept. of Computer Science, California Institute of Technology, Technical Report 5102, Oct. 1983.

[36] C. S. Steele, *Placement of Communicating Processes on Multiprocessor Networks*, Dept. of Computer Science, California Institute of Technology, Technical Report 5184, Apr. 1985.

[37] W-K. Su, R. Faucette and C. L. Seitz, *The C Programmer's Guide to the Cosmic Cube*, Dept. of Computer Science, California Institute of Technology, Technical Report 5203, Sept. 1985.

[38] D. G. Theriault, *Issues in the Design and Implementation of Act2*, MIT Artificial Intelligence Laboratory, Technical Report 728, June 1983.

[39] *UNIX Programmer's Manual*, Computer Science Division, Dept. of Electrical Engineering and Computer Sceience, University of California, Berkeley, Calif. , Aug. 1983.

[40] W. W. Ware, *The Ultimate Computer*, IEEE Spectrum, pp. 84-91, Mar. 1972.

[41] N. Wirth & H. Weber, *EULER: A Generalization of ALGOL, and its Formal Definition: Part I*, Communications of the ACM, Vol. 9, No. 1, pp. 13-25, Jan. 1966.