THE USE OF SEMANTIC INFORMATION IN A DISTRIBUTED DATA STRUCTURE

by

Douglas Bruce Bailey
B.Sc., University of British Columbia, 1986

THESIS SUBMITTED IN PARTIAL FULLFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

© Douglas B. Bailey 1988
SIMON FRASER UNIVERSITY
August 1988

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without permission of the author

APPROVAL

Name : Douglas Bruce Bailey

Degree : M.Sc. Computing Science

Title of Thesis : The Use of Semantic Information in a Distributed Data Structure

Examining Committee:

Chairperson: Dr. W. S. Luk

Dr. M. S. Atkins
Senior Supervisor

Dr. A. L. Liestman

Dr. J. G. Peters
External Examiner
School of Computing Science
Simon Fraser University

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

The Use of Semantic Information in a Distributed Data Structure

Author:

(signature)

Douglas Bailey

(name)

Aug 16/88

(date)

ABSTRACT

It is considered that an algorithm which solves a general class of problems will not be as efficient as an algorithm which solves a subset of the class of problems. Similarly, generalized distributed data structures (and their manipulative operations) are less efficient than particular algorithms for distributing data for a specific application.

The efficiency limitation of generalized data structures can be partially defeated by using semantic information. This thesis presents a distributed data structure which makes use of information provided by the application to implement an efficient distribution of the data. This information typically includes which nodes in a network are going to produce data, which nodes are going to read data, and how the data is to be distributed and replicated. An implementation of the distributed data structure is given, along with two examples. Analysis of these examples demonstrates the efficiency and flexibility of the data structure.

Acknowledgements

I would like to thank all those who helped me with this thesis: my supervisors, Stella Atkins and Art Liestman, for their constant attention; Joe Peters, Ada Fu and a few others for reading and criticizing my thesis proposal; and my wife and friends for their moral support.

Table of Contents

Abstract....iii
Acknowledgements....iv
Table of Contents....v
List of Figures....vi
1. Introduction....1
  1.1. Introduction to MOOSE....1
  1.2. Previous Work....4
2. MOOSE Definition....10
  2.1. MOOSE Semantics....10
  2.2. System Model....12
  2.3. Customization Features....13
    2.3.1. Interface Server Update Modes....14
    2.3.2. Interface Server Activities....16
    2.3.3. Correctness of Operation....16
  2.4. Summary of Customization Features....17
3. Implementation....19
  3.1. Additions to the object space - PUT....20
  3.2. Looking at the object space - READ....21
  3.3. Removing objects from the object space - GET....23
  3.4. Support Mechanisms....25
  3.5. Performance Analysis....26
4. An Example: Steiner Tree Problem....29
  4.1. Problem Definition....29
  4.2. Computational Model....31
  4.3. Configuring the MOOSE....32
5. An Example: Bitonic Merge....38
  5.1. Problem Definition....38
  5.2. Computational Model....40
  5.3. Configuring the MOOSE....40
6. Alternative Methods for Improving Efficiency....43
  6.1. Consistency Requirements....43
  6.2. Granularity of Data Distribution....44
    6.2.1. Limitations to the Prototype MOOSE....44
    6.2.2. Reducing the Granularity of the Distribution of Data....45
    6.2.3. Server Mapping Functions....46
    6.2.4. Example of Server Mapping Functions....50
  6.3. Exploiting Piggybacking....53
  6.4. Automating Selection of Update Modes....55
7. Conclusions....57
  7.1. Conclusions....57
  7.2. Further Research....57
Appendix A : Overview of the SR Programming Language....59
  A.1. Operations and Invocations:....59
  A.2. Sequential statements....61
Appendix B: MOOSE Implementation....63
Appendix C: Steiner Tree Example Source Code....92
References....99

List of Figures

Figure 2.1 : update modes 15

Figure 3.1 : Pseudocode of the `GET` operation 21

Figure 3.2 : Pseudocode of the `READ` operation 23

Figure 3.3 : Pseudocode of the `PUT` operation 24

Figure 4.1 : instance of the Steiner Tree Problem 29

Figure 4.2 : solution to the Steiner Tree Problem 30

Figure 4.3 : time required to solve the Steiner Tree Problem vs. number of worker processes used 37

Figure 5.1 : a comparator 38

Figure 5.2 : a bitonic sort network for 8 elements 39

Figure 6.1 : matrix decomposition for Jacobi method 51

Figure 6.2 : example problem decomposition 52

List of Tables

Table 3.1 : cost of MOOSE operations 27

Table 6.1 : server mapping function 53

1. Introduction

1.1. Introduction to MOOSE

The paradigm of shared memory, in which multiple active processes can access the same physical memory concurrently, has proved to be useful in many different fields of science. Computer systems which implement shared memory are usually uni- or multi-processor systems because the communications overhead between physically distributed components is very high. Consequently, shared physical memory is not feasible in computer systems distributed across a local area network (LAN). The obvious utility of shared memory remains, however.

One of the major research areas in computer science today is distributed programming. A distributed program consists of more than one process — often with different processes executing on different machines across a LAN — communicating through an exchange of messages. The exchanged messages carry state information and data among processes which do not necessarily share physical memory. An obvious alternative to this potentially chaotic collection of processes and messages is to provide a means of logically sharing memory or (as the proposed research will focus on) sharing the data which all processes need to access. If an efficient shared data structure is provided, the programmer will not have to build one. The resulting program will be more modular and less bug-prone than if the programmer had to build his own distributed data structure.

It is usually the case, however, that a data structure which satisfies the needs of a great many applications does not provide an efficient service: generality leads to inefficiency. The primary reason for this is that a programmer, in custom designing

a data structure, can make use of information on how the data is to be used by the application to improve efficiency.

A generalized data structure can be made more efficient by making use of some semantic information provided by the application. This thesis describes the design of such a generalized data structure, a MOOSE (MOdifiable Object StructurE), where information provided by the application is used to determine which of several different algorithms should be used to distribute the data. Many applications can use a MOOSE (almost) as efficiently as a custom designed data structure. The MOOSE system was designed to operate on a LAN such as an Ethernet where message communication costs are significant compared with calculation costs.

The following information is provided by the application to a MOOSE at the start of execution:

- Which nodes of the network are going to produce data, and which are going to use the data produced?
- How is data to be distributed? Should a piece of data be distributed as soon as it is produced, should the data be batched up and sent in one large block, or should a process have to ask if a given piece of data is available?
- How is the MOOSE going to be used? Is more than one process likely to attempt to grab the same piece of data at the same time? Does it matter if they both succeed, even though they should not, according to the definition of the data structure?

In effect, when the application gives this information it is agreeing that in using the MOOSE, the application will work within certain restrictions (e.g., if the applica-

tion specifies that a certain network node will not produce any data, then that node is restricted to not producing any data). If these restrictions are not met (i.e. the information provided by the application was incorrect) then the MOOSE might operate incorrectly. It is the programmer's task to make sure that the information given to the MOOSE is accurate. It is also the programmer's task to make sure that the information provided makes the program execute as efficiently as possible.

The MOOSE structure is aimed at a loosely coupled distributed system in which several uni- or multi-processors are connected across a LAN. MOOSE has been implemented in the distributed programming language SR ([Andrews and Olsson, 1987], [Olsson 1986], and [Andrews et al, 1988]), and runs on several SUN workstations running the Unix operating system connected across an Ethernet LAN. The language SR was chosen because it provides a very high level environment for distributed programming, along with a wide variety of communication and synchronization mechanisms. In addition, SR provides a consistent and intuitive syntax which makes the task of programming both easier and more enjoyable than programming in lower level languages like C.

In the remainder of this section, the work of other authors towards improving data structure efficiency by using application dependent information will be examined. In Section 2 and 3 following, the design and implementation of MOOSE will be described in detail, and some performance figures will be given. Sections 4 and 5 each give an example on the use of a MOOSE, along with some performance analysis. Section 6 discusses some alternative methods for improving efficiency through use of semantic information. Section 7 gives a brief conclusion.

1.2. Previous Work

A number of different authors have examined means by which the efficiency of a data structure can be improved in an application-dependent way. The best example of this can be found in [Cheriton, 1985 and 1986] which deal with Cheriton's "Problem-Oriented Shared Memory." In these papers, Cheriton examines a number of different distributed components of the V System, and the means by which their data structures have been made more efficient. Generally, he is interested in improving efficiency through relaxation of consistency.

Cheriton describes the relaxation of consistency through *relaxed store* and *relaxed fetch* operations. A relaxed store operation is not guaranteed to store the new information. A relaxed fetch operation may return stale or incorrect data, or might even indicate that the data is not known. He indicates several ways of dealing with the inconsistencies brought about by relaxed operations:

- **Detection on use** : the user of the data can detect that the data returned by a relaxed fetch operation is not correct.
- **Sufficient Accuracy** : the data returned may not be accurate or up-to-date, but it may be *sufficiently accurate* for the use it is intended.
- **Optional Data** : an application might be able to continue operation without a requested piece of data. The application might be able to perform correctly without the data by extrapolating or substituting other data. Alternatively, the application could make the fetch request again on a different machine which has the data. Cheriton refers to this as *function shipping* where the execution of the function is moved, as opposed to *data shipping* where the data is moved.
- **Discardable Updates** : an application might be able to tolerate the effects of a lost update. For example, the data might be regularly updated so that if a

store operation fails, the stored data is stale until another store operation succeeds.

The first example that Cheriton gives is the V name service. In this service, a primary copy of the information associated with a name is kept at one machine, with caches kept at other machines. Stale or incorrect data is detected *when it is used* . On a cache miss, the correct information is determined through multicast communication, and the cache is updated.

The V time-of-day clock is an example of shared memory with sufficient accuracy. Each machine's local clock is periodically corrected with a message from a network time server, and so the local clock never differs from the network time by more than a small amount.

Another example of sufficient accuracy of information and detection of stale data on use is in global scheduling across a LAN. In the V System a user can indicate that a command is to be executed on any machine in the network cluster. Thus, the scheduler will want up-to-date information about the resources available to different machines in order to best schedule the job. The servers could broadcast their current status periodically, or when there is a significant change. Other servers would then make use of sufficiently accurate information. When a scheduler tries to start a job on a different machine, it could include a description of the presumed load. If this description is too inaccurate, the request can be refused — inaccurate data is thus detected on use. These methods of global scheduling are currently being implemented in the V System.

In [Ravindran 1987], "Application Driven Shared Variables" (ADSVs) are defined. Ravindran provides a thorough definition of his ADSVs, which are simply shared variables which only provide a weak form of consistency. He shows how his ADSVs might be used in three examples: management of leadership in a server group, management of the printer in a spooler group, and management of the name space of machines in a distributed system.

Terry, in [Terry 1987] considers the problem of maintaining cache consistency in a distributed system. He notes that maintaining full consistency is very expensive, and a possible alternative is to loosen consistency requirements and consider cached data as "hints" rather than accurate information. This is an acceptable alternative for a variety of applications. For example, in the distributed mail service Grapevine, servers cache information about mailboxes not stored locally. A server will detect the inaccuracy of its cache when it attempts to forward mail to a mailbox which has been moved. Then, a global registration service is consulted to update the cached information.

The work done by Schwarz and Spector in [Schwarz and Spector, 1984] is interesting but not quite as relevant to the topic under discussion. Schwarz and Spector examine the consistency requirements of shared abstract types from a more theoretical point of view. The primary purpose of their paper is to study a method of notation for dependencies between different operations on an abstract type; however they also describe a locking technique which makes use of type-specific information provided by the programmer to improve availability.

In all of these works, solutions have been proposed for specific problems. Efficiency has been improved at the expense of consistency. However, a MOOSE

attempts to provide a general solution to a wide variety of problems. As such, a number of methods for increasing efficiency had to be considered which were just taken for granted in the works described above (in implementing a specific algorithm, the pattern of message passing can easily be designed so that the minimum number of messages are required. In a generalized data structure, achieving optimal message passing is difficult).

The work which inspired the form of the MOOSE data structure was [Carriero and Gelernter, 1985] — "The S/Net's Linda Kernel." Linda is a distributed data structure implemented on the S/Net multicomputer (built by AT&T Bell and based on a fast, word-parallel bus interconnect). Rather than providing operations for assigning values to labels and for reading the value associated with a label (such as the standard read and write operations), Linda provides a tuple space. The Linda tuple space is simply a replicated, shared set of tuples, where a tuple is an ordered list of values. The tuple space is manipulated with three operations: `OUT`, `READ`, and `IN`.

The `OUT` operation adds a tuple to the tuple space. If an `OUT` operation is performed with a tuple already existing in the tuple space, the tuple space remains unchanged.

`READ`(X) takes a "tuple template", X (an object which defines a tuples form and optionally fills in some of the tuples values) and returns a tuple which "matches" the template X. A tuple is said to match a template if the forms are equivalent and any values given in the template are matched by those in the tuple. If more than one matching tuple is available, one is chosen non-deterministically. If no match-

ing tuple exists in the tuple space, the READ operation blocks until such a tuple is available.

The IN operation is similar to the READ operation, except that the IN operation also removes the tuple from the tuple space. If two IN operations attempt to remove the same tuple, only one will succeed and the other will be forced to try again with a different tuple, or block until a matching tuple is available.

While the Linda data structure is quite interesting, it is not appropriate for a loosely coupled distributed system. In the Linda implementation, a copy of the entire tuple space is kept at each node in the network. Because of the slow message transfer times (relative to the S/Net bus) between two Unix processes connected over an Ethernet, such an implementation results in far too much communications overhead to be feasible for most applications. Even if a reliable broadcast mechanism is available to reduce the cost of transmitting a piece of data to every node in the network, the cost of processing that data at each node would still be too high.

A further point of interest in [Carriero and Gelernter, 1985] concerns their "token/worker" model of computation. In this model, a set of workers compete over work tokens. When a worker successfully grabs a work token, the token is removed from the data space so that no other workers will attempt to grab it. The worker then does the work represented by the token, and goes back to grab another token. This model is interesting for a number of reasons: it scales transparently as more workers are added, it automatically balances the work load among the workers, and it can be fairly resilient to node failure among the workers. Because the MOOSE data structure is so similar to the Linda data structure, implementation of the token/worker model with a MOOSE is quite simple.

In [Cheriton and Stumm, 1987], a master-slave approach to distributed programs is introduced and called a *multi-satellite star*. This structure is in fact nearly identical to the token/worker model explained above, with a *star central* allocating subtasks to *satellite modules*. The purpose of their study was to examine methods of realizing the computational potential of workstation clusters. They concluded that their multi-satellite star configuration of a workstation cluster provides a usable parallel machine for certain classes of problems. The master/slave approach is not the best solution paradigm for every problem, however, and thus the multi-satellite star is limited. A MOOSE can provide the functionality of the multi-satellite star without the limitations present in that computation model.

2. MOOSE Definition

2.1. MOOSE Semantics

A MOOSE is logically a separate entity from the processes which use it. Processes communicate with the MOOSE through interface servers. The interface servers implement the basic operations on the shared objects. Ideally, one interface server will exist on each node in the network. Interface servers can process operations concurrently.

The basic data type is an object. The form of an object consists of a name and an ordered list of simple types (integer, boolean and character string). The value of an object is an ordered list of values, with types to match the form of the object. For example, an object might have the form ('cell', integer, integer, character string). An object of the form 'cell' might have a value (5,10,'hello').

An object template consists of a partially defined object. The form of the template is defined fully, but some or all of the values are undefined. For example, ('cell',integer,10,character string) is a valid object template. An object is said to match a template if the forms are equivalent, and the all of the defined values of the template match the corresponding values of the object. For example, the object ('cell',5,10,'hello') matches the template ('cell',integer,10,character string) but not the template ('dif',integer, 10, character string) or ('cell',10,integer,character string).

The operations which the interface servers implement are `READ`, `GET` and `PUT`. These operations correspond to Linda's `READ`, `IN` and `OUT`, but some of the names were changed to avoid confusion.

The READ operation takes an object template for input, and returns a matching object. If more than a single object in the object space matches the object template, one is chosen non-deterministically to be returned. If no matching object is available, the operation blocks until one becomes available. The object space remains unaffected by a READ operation.

The GET operation is identical to the READ operation except that when the operation has been completed, the object has been removed from the object space. Several GET operations might be invoked with templates which match the same object (such operations are said to be competing). If this happens, only one GET for each object matching the template should succeed at once. The rest should block until more objects matching the template become available. If two GET operations are competing for one object, only one should succeed immediately while the other blocks.

The PUT operation takes an object for a parameter and places the object into the object space. If the object is already in the object space, it is duplicated — multiple copies of an object are allowed (unlike in Linda).

For example, consider the piece of pseudo-code given below:

```
do (true) ->
   GET('multi',integer a, integer b, integer c)
   d := a*b*c
   PUT('product',d)
od
```

It GETs from the object space an object of the form:

```
('multi',integer,integer,integer).
```

It then multiplies the three integers together and places the result back into the object space with a PUT operation. This action is repeated indefinitely. When the object space is emptied of objects which match the GET template, the algorithm will block on the GET.

Throughout the rest of this section, an object template will be specified for the GET and READ operations by specifying the object name as the first parameter, and the appropriate types or values as subsequent parameters. Values will be given explicitly, while types will be given in the form 'type name' (such as 'int n') where *type* specifies the field type, and *name* specifies the name of the variable to instantiate with the corresponding value of the matching object. For example, the operation GET('result', 1, 5, int result) would execute the GET operation on an object with the form ('result', int, int, int). The first and second integers of the matched object must equal 1 and 5 respectively, and the value of the third integer will be returned and assigned to the variable named *result* .

2.2. System Model

A program using a MOOSE consists of several processes distributed across a LAN. Each machine which executes one of these processes also has a MOOSE interface server executing on it. Thus each process in the program has access to a local interface server. The interface servers communicate through the LAN to implement the MOOSE. Note that if several MOOSE-using programs are executing at once, their respective MOOSEs are completely independent — a MOOSE is associated with a program not with a system, so several might exist at one time.

In a LAN, the communications costs are high but not so high as to be prohibitive (as would be the case in a wide-area network). The total cost of messages sent among machines (including protocol overhead and transmission time) will still constitute the majority of the total cost of implementing a MOOSE. It is with the goal of minimizing the number of messages sent across the LAN that the analyses in Sections 4, 5 and 6 are made.

Some assumptions are made. The most important assumption is that an underlying guaranteed delivery mechanism is in place so that no messages are lost: this is reasonable for a system composed of Sun-3 workstations running Unix which have many Ethernet interface buffers so that even unreliable datagram messages are unlikely to be lost. It is also assumed that message costs between machines are constant, regardless of the size of the message or where it is being sent. This assumption is reasonable as long as no message exceeds the size of an ethernet packet (of ≈ 1500 bytes). This will certainly not be the case in general (consider a very large object, or just a cache update (see Section 2.3. following) of many small objects), but should be a reasonable assumption for the examples given in Sections 4 and 5.

2.3. Customization Features

Up to this point, MOOSE has been very similar to Linda. It is through the customizable features of MOOSE that the two data structures differ. It is through these features that the application can give the MOOSE application-specific information so that the execution of the MOOSE will be as close to optimally efficient as possible.

2.3.1. Interface Server Update Modes

Consider a system in which ten different computers gather data from their environment while a single computer processes that data. With the Linda system, the ten producer computers would not only have to send their data to all ten other computers, they would also have to receive data from nine other data producers — data that they do not need. All updates made to the tuple space are immediately communicated to all Linda servers. This method will always work, but it is far from efficient.

In this case, the overhead can be avoided by the use of the first customization feature of a MOOSE: the update modes. Each interface server has a different "add update mode" and a different "delete update mode" for each object form. The modes might be 'eager', 'cached' or 'lazy'. Each server knows the modes of all other servers.

When a server has an add update mode for a particular object form set to eager, then that server is immediately informed by all other interface servers of all additions of objects of that form. Similarly, if a server has a delete update mode of eager, all other servers immediately inform it of deletions from the object space.

Lazy update mode means that a server does not want to be informed of updates to the object space. A server in lazy add update mode must ask other servers if a given object is in the object space. A server in lazy delete object mode must ask other servers if a given object is still in the object space, even if the object is still present locally.

Eager and lazy update modes represent two extremes. Between the two extremes is the cached update mode — the updates are cached by the other servers before being sent. A cache of updates is sent when the size limit of a cache is reached or after a certain time limit has expired (the application defines both the cache size and the time limit). Eager update mode is equivalent to cached update mode with a cache size of 1. Lazy update mode is equivalent to cached update mode with a cache size and delay between sending of the cache of infinity.

Thus the update modes provide nine distinct states for each server to be in, for each object form in use (although not every combination makes sense: if the add update mode is lazy, using an eager delete update mode is pointless). The update mode space is shown graphically in Figure 2.1 below.

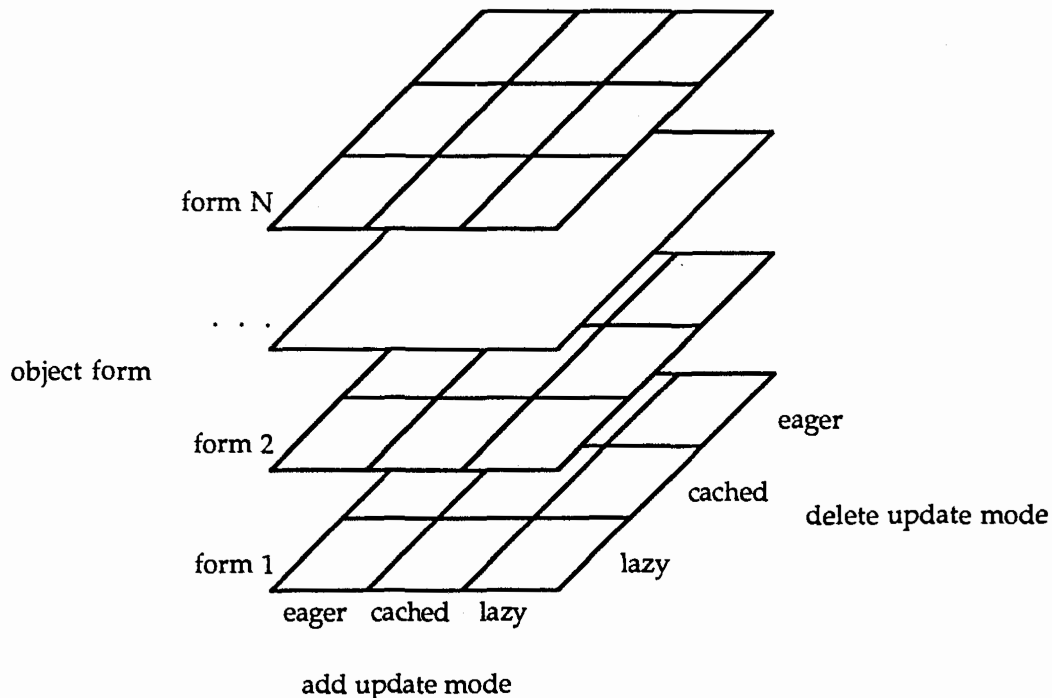

Figure 2.1: there are nine update modes per object form

2.3.2. Interface Server Activities

The update modes solve some of the inefficiencies of the Linda system, but not all of them. The customized update modes make execution efficient in situations in which there are a small number of data *consumers* (interface servers through which at least one READ or GET is executed). On the other hand, there may be a large number of consumers but a small number of *producers* (interface servers through which at least one PUT is executed. A server can be both a consumer and a producer). One possibility is to make all of the consumers eager with respect to additions to the object space. This works fine if each consumer accesses most of the data. If each consumer only accesses a small quantity of the data, however, message exchanges are wasted in sending the data which will not be accessed.

If it is known which servers are producers, then the consumers can be lazy with respect to additions, and poll the producers only when a piece of data is required. It can be shown that this latter method is more efficient than giving the consumers an eager add update mode if each consumer READs less than 1/(2M) of the data, where M is the number of producers[1]. Thus an application must specify the activity of each interface server for each object form — consumer, producer or both.

2.3.3. Correctness of Operation

As is usually the case in distributed systems, ensuring correct operation in all contingencies requires a great deal of effort. Because of the simple semantics of a

[1]If a server is eager-add with respect to a particular object form, it will receive notification of all additions of objects of that form. If N objects are PUT, this requires N messages, regardless of how many of those objects are eventually READ. If α objects are READ, then the cost per object READ is $N/_a$. If the server is lazy-add, then the PUTs do not require any messages, but each READ requires 2M messages, where M is the number of producer servers. Thus the cost per READ operation for an eager-add server ($N/_a$) is less than the cost per READ operation for a lazy-add server (2M) iff ($N/_a > 2M$), or ($^a/_N < {}^1/_{2M}$), i.e. the proportion of objects READ is less than $^1/_{2M}$.

MOOSE, the only case where correctness is difficult to ensure is when different processes are all attempting `GET`s with matching templates. These situations are quite specialized, however. There are a great many algorithms which do not present these problems (consider a situation in which many machines collect data and place it into the object space, and one server consumes the data and analyzes it) and these should not have to pay the penalty for correct operation in every possible situation.

There are two 'correctness' modes which apply to an object form — *safe* and *risky* . If an object form is safe, all `GET` operations are guaranteed to execute correctly. If an object form is risky, competing `GET` operations might not execute correctly; however, all non-competitive `GET` operations will execute correctly. Thus, if a situation where two processes are trying to grab the same object is never going to arise, or if correct operation in that case is not important (e.g. if each object represents a little work to be done, and taking the chance on having a task done more than once to save a few messages is worth while), a risky correctness mode can be used to improve efficiency. Unlike other customization features, the correctness mode for an object form is the same for every server in the network.

2.4. Summary of Customization Features

There are three customization features of MOOSE: update modes, server activities and correctness. The update modes and server activities apply to particular servers with respect to particular forms. There are two update modes: add update mode and delete update mode. Each mode might be eager, cached or lazy. There are two server activities: producer and consumer. Each server is assigned either or both of the activities of producing and consuming. Correctness mode applies to a partic-

ular object form: all servers must implement the same level of correctness. There are two correctness modes: safe and risky.

Thus there are 72 different possible states for a given server with respect to a particular object form. There are 3 possible add update modes, 3 possible delete update modes, 2 possible states regarding production, and 2 possible states regarding consumption. In addition, there are 2 possible correctness modes with respect to a given object form which apply to all servers. While all combinations are permitted, not all make sense. For example, if a server is not a consumer, sending object space updates to that server is just a waste of resources. Thus specifying eager add or delete update mode is pointless.

Note that each of the variables pertains to a particular object form. Each server must know the values of the different variables for every server. Because of this, it is necessary to explicitly create an object space (or perhaps "sub-space") for a particular object form, before an object of that form can be used. In this implementation we assume that this initialization data is reliably distributed before the application starts executing.

3. Implementation

As mentioned previously, MOOSE has been implemented on a group of Sun-3 workstations connected on an Ethernet LAN, running the Unix 4.2 BSD operating system in the distributed programming language SR. Because SR is a strongly typed language, implementation of objects as described in the previous section of this thesis would have proved to be very difficult and quite messy. Since such a task would not have served any useful research purpose, a prototype version of an object was implemented where an object is simply a list (of any size up to a system-defined maximum) of integers.

For a complete listing of all of the source code for the implementation, refer to Appendix B. For those not familiar with the SR programming language, a brief summary of the basic features of SR is given in Appendix A.

Before a description of the algorithms for handling different operations is given, an overview of the possible situations a MOOSE server might be in will be useful. Remember that there are 4 pieces of application-dependent information given to the MOOSE servers about each object form: the add-update modes of all of the MOOSE servers; the delete-update modes of all of the MOOSE servers; which of the MOOSE servers are producers and which are consumers; and the consistency mode of the object form.

Of this information, only the update modes affect the accuracy and completeness of the local database stored with each server. Each server must know its own add and delete update modes to know how to perform the READ and GET operations correctly. Each server must know the add update modes of every other server to exe-

cute a PUT operation correctly. Lastly, each server must know the delete update modes of every other server in order to correctly remove an object from the object space (as part of the GET operation). The server activity information provided by the application is used to optimize the operations in special cases (such as the GET operation executed from a lazy-add server when there is only one producer, where the operation is shipped to the producer server).

The server which PUT an object into the object space is the owner of that object. In safe correctness mode, it is this server's responsibility to ensure that only one attempt to delete the object succeeds. Even if the correctness mode is risky, the owner of an object is informed immediately if that object is deleted, even if the owner has a delete update mode of lazy or cached. Thus, the owner of an object is the final authority in deciding whether or not a given object has been removed from the object space.

3.1. Additions to the object space - PUT

Pseudocode for the PUT operation is given in Figure 3.1. The PUT procedure simply calls the global_add procedure which in turn is responsible for immediately sending off the addition to all eager add update servers and for caching the addition for all cached update servers[2].

[2]Note that the addition is neither sent nor cached to servers which are not consumers, regardless of their add update mode.

```
proc PUT(X)
   call global_add(X)
end

proc global_add(X)
   fa ({all servers S such that (eager_add(S,X) and
               consumer(S,X)) or (S=me)}) ->
      send S.local_add(X)
   af
   co ({all servers S such that cached_add(S,X) and
         consumer(S,X)}) ->
      send cache_add(S,X)
   oc
end

process a
   do (true) ->
      in cache_add(S,X) ->
         { add X to S's cache C }
         if ({C full}) ->
            call fire_off_add(S)
         fi
      ni
   od
end

proc fire_off_add(S)
   /* this proc is invoked on timeout or cache full conditions */
   { if S's cache not empty send it off }
end

proc local_add(X)
   { add X to the local database }
end
```

Figure 3.1 : Pseudocode of the PUT operation

3.2. Looking at the object space - READ

Pseudocode for the READ operation is given in Figure 3.2. READing the object space is considerably more complicated than putting an object into the object space, because of the various update modes that the server performing the READ might be in. If the server is in eager add update mode, it can be sure that if the object exists, it has it stored locally. If the server is in cached add update mode, it can be sure that if the object exists, it has or will shortly have it in local storage. However, if the server is in lazy add update mode, it may or may not have the object stored locally (if the

object was produced locally, it will be stored locally, but if it was produced elsewhere, it will not be stored locally).

Even if an object has been found stored locally, it may not exist in the object space — it may have been removed earlier. This is possible if the server is in lazy or cached delete update mode.

Thus, altogether there are three different possibilities that must be handled differently.

1. If the server performing the READ is in lazy add mode (with any delete mode) then the object may or may not be stored locally so the reader cannot depend on locally stored information. In the general case, the READ request is forwarded on to all producers of the given object form. However, if a server exists in eager-add update mode (and thus has all objects of the given form in its local store), the function is forwarded to that server. Here, *function shipping* is used instead of *data shipping*.
2. If the server performing the READ is in eager or cached add mode and eager delete mode, then the object, if it has ever been or ever will be produced, will eventually be stored locally. The reader can depend on the local stores so need not check elsewhere.
3. If the server performing the READ is in eager or cached add mode and cached or lazy delete mode, then if the object exists or will exist, it will eventually be stored locally. However, when a matching object is found locally, it may be an object that has since been deleted from the object space so the reader must check with the owner of the object to see if indeed it still exists.

```
proc READ(X)
   if (lazy_add(me)) ->
      /* operate same for all delete modes */
      if (num_producers > 1 and ∃ some server S such that
                                                   eager_add(S,X)) ->
         S.READ(X)
      [] else ->
         if ({X not present locally}) ->
            co ({all producers S})
               call S.owner_read(X) -> exit
            oc
         fi
      fi
      return X
   /* eager or cached add mode */
   [] eager_delete(me) ->
      call checkwait(X)
      return X
   [] else ->     /* lazy or cached delete */
      do (true) ->
         checkwait(X)
         if owner(X).still_there(X) ->
            return X
         [] else ->
            call me.local_delete(X)
         fi
      od
   fi
end

/* the local_delete operation is described under the Get operation */
```

Figure 3.2 : Pseudocode for the READ operation

3.3. Removing objects from the object space - GET

Pseudocode for the GET operation is given in Figure 3.3. GET is much simplified by the fact that most of its action is duplicated by the READ operation. A GET is effectively a READ followed by an attempt to delete the object once the READ has returned. This scheme is complicated slightly by the different correctness modes which affect the GET operation, and by the different delete update modes of the other servers. Recall that to maintain correct operation and consistency, the owner of an object is responsible for deleting it from the object space. In the case of risky correctness mode, however, the server performing the GET operation also invokes the delete operation (although it must do so in such a way as to make sure that the

object is deleted from the owner's local database immediately, even if the owner is in cached or lazy delete update mode). Thus, in risky correctness mode, it is possible for two servers to attempt to GET the same object at the same time and, since they do not check with a central authority, both might succeed.

The GET operation is an example of how the server activities of producer/-consumer can be exploited. Consider a case in which only one server produces objects of a given form, and a different server which is in lazy add update mode performs a GET operation on a object of that form. The cheapest way of performing the GET operation (in terms of messages sent) is to forward the request on to the one producer of that object form.

```
proc GET(X)
   if (correct_mode = safe) ->
      if (num_producers > 1) or Iamproducer(X) ->
         do (true) ->
            call READ(X)
            if owner(X).owner_delete(X) ->
               return X
            fi
         od
      [] else ->
         call producer(X).GET(X)
      fi
   [] (correct_mode = risky) ->
      if (num_producers > 1) or Iamproducer(X) ->
         call READ(X)
         send owner(X).global_delete(X)
      [] else ->
         call producer(X).GET(X)
      fi
   fi
end
```

```
process a
   do (true) ->
      in owner_delete(X) returns success ->
         if find(X) and (owner(X) = me) ->
            call global_delete(X)
            success = true
         [] else ->
            success = false
         fi
      ni
   od
end

proc global_delete(X)
   /* delete object locally, regardless of delete update mode */
   local_delete(X)
   /* inform all eager delete servers */
   fa ({all servers S such that (S!=me) and eager_delete(S,X)
                                and consumer(S,X)})
      send S.local_delete(X)
   af
   /* inform all cached delete servers */
   co ({all servers S such that (S!=me) and cached_delete(S,X)
                                and consumer(S,X)})
      call cache_delete(S,X)
   oc
end

proc cache_delete(S,X) ->
   { add delete(X) command to S's cache }
   if ({C full}) ->
      call fire_off_delete(S)
   fi
end

proc fire_off_delete(S)
   /* this proc is invoked on timeout or cache full conditions */
   { if S's cache not empty send it off }
end

proc local_add(X)
   { add X to the local database }
end
```

Figure 3.3 : Pseudocode for the GET operation

3.4. Support Mechanisms

Obviously a great deal of support mechanism is missing from this description. However the details that are missing are primarily concerned with managing the local databases (such as local_add and local_delete) and are not considered relevant to the discussion. The more obscure operations are described here:

`checkwait`(X) : this procedure checks the local store to see if the an object matching the template X is present. If so, it returns the object. If not, it waits until such an object is added and then returns it.

`producer`(X) : this function is called only if there is just a single producer of the object form of X. The capability for that producer resource is returned by the function

`owner_read`(X) : this operation is present to allow a lazy add update mode server to forward READ operations on to producer servers. If this operation is invoked on a server, it starts up a process which watches all objects that that server adds or has added. The process replies to the original READ operation when an object matching the template X is found. The READ operation may thus be effectively partitioned between the producer servers.

`still_there`(X) : this operation is invoked on the server which owns X and is used to confirm that X has not been deleted yet.

3.5. Performance Analysis

Studying this implementation allows us to derive Table 3.1 below which gives the cost of different MOOSE operations (READ, GET, PUT) in terms of messages sent across the LAN. An entry in the table gives the cost of the operations in a system with δ moose servers, all in the add update mode indicated at the top of the column, and in the delete update mode indicated at the head of the row, and all both producing and consuming data. There are two figures given for the GET operation, indicating the cost of the operation in safe or risky correctness mode (marked 's' and 'r' respectively). The figures given for the GET operation are based on the assumption that the owner of the object returned by the GET operation is a different server from the server which initiated the GET (the most common situation). For operations

working in a cached update mode, add-cache sizes are assumed to be C_a, and delete-cache sizes are assumed to be C_d. Lastly, the figures given for GET and READ operations with cached or lazy delete mode are based on the assumption that there are no "cache misses" — the mechanism used to select one of several objects in local memory always manages to select one that has not been deleted yet (this will not necessarily be the case in cached or lazy delete mode).

Table 3.1: cost of READ, GET and PUT operations with multiple producers and consumers

	eager add	cached add	lazy add
eager delete	PUT: $\delta - 1$ READ: 0 GET (s): $\delta + 1$ GET (r): $\delta - 1$	PUT: $(\delta - 1)/C_a$ READ: 0 GET (s): $\delta + 1$ GET (r): $\delta - 1$	n/a
cached delete	PUT: $\delta - 1$ READ: 2 GET (s): $4 + (\delta - 1)/C_d$ GET (r): $3 + (\delta - 2)/C_d$	PUT: $(\delta - 1)/C_a$ READ: 2 GET (s): $4 + (\delta - 1)/C_d$ GET (r): $3 + (\delta - 2)/C_d$	n/a
lazy delete	PUT: $\delta - 1$ READ: 2 GET (s): 4 GET (r): 3	PUT: $(\delta - 1)/C_a$ READ: 2 GET (s): 4 GET (r): 3	PUT: 0 READ: $2\delta - 2$ GET (s): 2δ GET (r): 3

In this efficiency analysis (and those found in the following sections), the local processing times of the various operations (READ, GET, PUT) are ignored. The local processing time results from maintenance of a local database. This thesis is not con-

cerned with issues concerning this local database, so no attempt has been made to optimize it. Thus in the analysis, efficiency has been measured by counting the number of messages sent between hosts in the network. In comparing this analysis to benchmark tests, an implicit assumption is made: that the local processing time for an operation is approximately proportional to the number of messages sent between workstations. Without this assumption, measuring the cost of an operation by counting the number of messages is not valid. This is a reasonable assumption as long as databases remain fairly small (since the underlying database mechanism is a binary tree, the cost of local database operations only increases with the $\log_2$ of the database size).

4. An Example: Steiner Tree Problem

4.1. Problem Definition

The first example presented here is the Steiner Tree Problem in Networks. The Steiner Tree Problem is an NP-complete problem taken from graph theory. The problem may be stated as:

GIVEN: a graph G=(V, E, c) with n vertices ($|V| = n$) and m edges ($|E| = m$) and a cost function c ($c : E \rightarrow \Re$) and a subset of the nodes of the graph, Z

FIND: the minimum weight tree which spans Z.

The graph shown below (in Figure 4.1) contains 23 nodes. The nodes in the set Z are shown shaded to differentiate them from the other nodes. All of the edges of this graph are of equal weight (c=1). The solution to the Steiner Tree Problem applied to this example is shown in Figure 4.2.

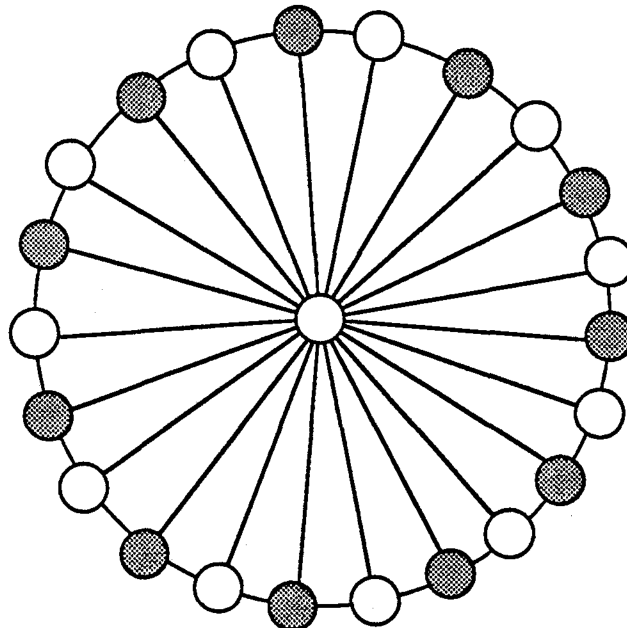

Figure 4.1 : problem instance for Steiner Tree Problem with 23 nodes. Nodes in the set Z are shown filled.

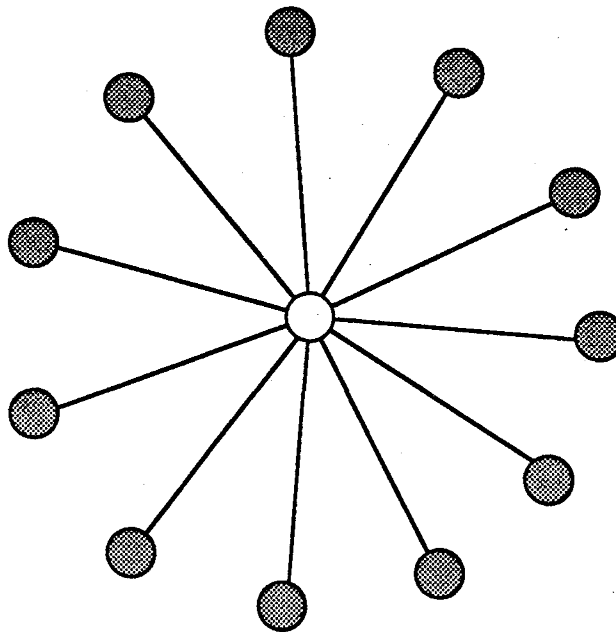

Figure 4.2 : solution to Steiner Tree Problem applied to graph shown in Figure 4.1.

For a survey of different solutions to the Steiner Tree Problem, see [Winter, 1987]. The solution adopted in this thesis is a brute force approach: determine the minimum weight spanning tree of every subgraph of G which contains Z. The spanning tree with the least weight is also the least weight tree which spans Z.

Since there is an exponential number of subgraphs for which the minimum weight spanning tree must be determined, the algorithm is exponential in nature. The algorithm used to determine the minimum weight spanning tree of each subgraph of G (known as Prim's algorithm) is requires time polynomial in $|V|$. For details on Prim's algorithm, consult [Horowitz and Sahni, 1978].

In order to distribute this algorithm, the set of subgraphs of G must be partitioned into a number of subsets. The minimum weight spanning tree is calculated for each graph in the subset, and the least weight of all of the trees in the subset is

offered as a tentative solution. The least weight of all of the tentative solutions is taken as the solution to the Steiner Tree Problem.

The subsets define the granularity of the distribution — the spanning trees of different subsets might be calculated on different machines, but the spanning trees of two graphs within the same subset will be calculated on the same machine. Thus each of the subsets of the set of subgraphs of G constitutes a single *task* to perform.

4.2. Computational Model

The token/worker model of computation will be used. For each task to perform (i.e. for each partition of the set of subgraphs of G), a token will be placed into a token space. The token will contain enough information to transform the original graph G into each of the subgraphs within the partition. An arbitrary number of worker processes will exist. Upon creation, the workers will read the data for the original graph, and then compete for work tokens. When a work token has been successfully grabbed by a worker, that worker will perform the task represented by the token (i.e. compute the minimum weight spanning tree of every subgraph in the partition and offer the least weight tree as a potential solution) and then go back to get another token.

A single master process exists which is responsible for creating the data and tokens, and for gathering all of the results computed by the workers. The master decides which of these is the least weight and thus determines the solution. Since the number of worker processes is arbitrary, any number of workers may be used. With more workers, the time required for a computation should decrease.

The algorithms for the master and worker processes are quite trivial:

master process:

```
PUT the original matrix
PUT the tokens
READ the results
```

worker processes:

```
READ the input graph
do (true) ->
   GET a task token
   fa (subgraphs specified by the token) ->
      build the subgraph from the original graph
      determine the minimum weight spanning tree
   af
   PUT least of all minimum weight spanning trees
od
```

4.3. Configuring the MOOSE

In implementing this solution using a MOOSE, three object forms are needed: a form for the input graph; a form for the tokens; and a form for the results (these forms have been named 'data', 'token' and 'result', respectively). The input data and the tokens are produced by the master process and consumed by the worker processes. The results are produced by the workers and consumed by the master. Using Table 3.1 and some knowledge of the algorithms given in Section 3, the best update modes for all of the servers and all of the object forms can be determined.

Objects of the form 'data' are PUT by the master process and READ by the worker processes. Thus the best update modes to use are eager add and eager delete for all servers, giving a total cost of distributing the 'data' objects to ω workers of ω messages per object (in this example it is necessary to differentiate between the number of worker processes, ω, and the total number of processes, δ. Since only the master process is not a worker, $\delta = \omega+1$).

The master process PUTs objects of the form 'token' while the worker processes GET these objects. Referring to Table 3.1, it seems that the best update modes for this object form would be cached add/lazy delete. However, Table 3.1 is based on the assumption that several processes produce the objects, while in this case only the master process produces the objects. By referring to the algorithms given in Section 3, one can see that the GET operation is optimized for the case where only one process produces the data, by function shipping the GET request from a lazy add server to the producer server. Thus the least expensive update modes are eager add/eager delete for the master process, and lazy add/lazy delete for the worker processes.

For objects of the type 'result', the objects should be sent to the master process, since this process will READ the objects, but not to the other worker processes. This may be accomplished by assigning the master process an add update mode of eager and the worker processes an add update mode of lazy. Since the objects are never removed from the object space, the delete update modes for objects of type 'result' do not matter, but for completeness they may arbitrarily be assigned eager delete update mode for the master and lazy delete update mode for the workers.

object form	contains	master update (add/delete)	worker update (add/delete)
'data'	original matrix	eager/eager	eager/eager
'token'	work tokens	eager/eager	lazy/lazy
'result'	task results	eager/eager	lazy/lazy

Since the workers compete over the task tokens, and allowing more than a single worker to successfully grab the same token would result in wasted effort, the correctness mode for the object form 'token' must be safe. Otherwise, if the correctness mode were risky, more than a single worker could grab the same work token, and thus work would be wasted. Since the cost of performing the work associated with a

token is greater than the cost of grabbing the token, it is not advisable to allow extra work to be done to improve concurrency by using a risky correctness mode. Since neither the data nor the results need ever be deleted from the object space, the correctness mode for the forms 'data' and 'result' do not matter.

Analyzing this algorithm for efficiency is quite simple. Consider a serial algorithm which solves the Steiner Tree Problem using the same brute force method. Assume that such an algorithm requires a time period C to solve a particular problem instance. This token/worker distributed implementation of the brute force method, using ω worker processes, should then require C/ω + {communication cost} time to solve the same problem. For a problem of size n (i.e. a graph of n nodes), $C \in O(2^n)$. As long as the communications cost is small, ω is reasonably small (as it will be on a LAN), and the problem size is reasonably large ($n > 10$), the speedup as workers are added should be effectively linear.

There are two components of the communications cost: the one-time event of distributing the data, and the per token cost of assigning a task to a worker and having the worker return the results. Assuming that the graph of n nodes is represented by its adjacency matrix (an n×n matrix), and each row of the adjacency matrix is represented by a single object of the 'data' form, the cost of distributing the matrix to ω workers is $O(n^2 * \omega)$ in the general case[3].

[3] Since the messages sent between MOOSE servers for the distribution of the adjacency matrix might be larger than an Ethernet message packet, simply counting messages is not sufficient, and the cost of a single message must be assumed to be O(n) — however, in the problem instance given in figure 4.1, each message should be small enough to fit within a single network packet and so the cost of distributing the matrix to ω workers is $O(n\omega)$.

The cost of distributing the work tokens among the workers is a little more complicated to determine. Recall that the master process which will PUT the tokens into the object space is eager-add and eager-delete with respect to the 'token' object form. The workers which will GET the tokens are lazy-add and lazy-delete with respect to the 'token' object form. When the master process PUTs a token into the object space, no messages are sent across the network. When a worker process GETs a token, since the only 'token' producer is the master process, the GET operation is forwarded to the master, hence requiring two messages (one sent to the master, and a reply back from the master). Performing the GET at the master requires no messages, since the master is the owner of the object, and all other MOOSE servers are lazy-delete. Thus it costs two messages for the master to produce a token and for a worker to grab it.

When a worker has completed the work specified by one token, the results must be sent back to the master process. This entails a PUT of a 'result' form object. Since the master process is eager-add but all other processes are lazy-add with respect to the 'result' form, the PUT operation only requires a single message. When the master READs the result, no messages are required since the master is in eager-add mode and thus the object is already in its local memory.

Thus, the total cost of distributing τ tasks among ω workers, including the cost of distributing the input graph initially, is: $O(n^2 * \omega) + 3\tau$. The total cost of solving the Steiner Tree Problem with this algorithm is thus $C/_{\omega} + O(n^2 * \omega) + 3\tau$.

Appendix C gives the program for solving the Steiner Tree Problem. In this program, graphs are represented by their adjacency matrices. The subgraphs for which

the minimum weight spanning tree must be calculated are enumerated[4] (it is assumed here that an integer is big enough to enumerate the subgraphs: in particular, $|V-Z| \leq 31$). Thus task tokens simply contain two integers which specify a range. The task represented by such a token consists of computing the minimum weight spanning tree for each subgraph within that range.

This program was run on the example shown in Figure 4.1, with 1, 2, 3, 4 and 5 workers and $\tau = 16$ (the value $\tau = 16$ was chosen because it led to the best results for these test runs. A different value of τ might prove to be best for test runs with different numbers of machines). The speedup with the number of workers is plotted in Figure 4.3.

In Figure 4.3, the ×'s show the best results obtained from several tests run when the system load was small (in the system on which tests were run, the system was never completely free of other user tasks. The best results of several tests were chosen because they best reflected the results that would be obtained from a system running only the steiner tree program). The line drawn on the graph shows optimal linear speedup from the time required for computation with one worker. The timing results obtained from running the program given in Appendix C with different numbers of workers show that the speedup is good, with the speedup degenerating as the number of workers increases.

[4] In order to enumerate all of the subgraphs of a graph G=(V,E) which contain a given set of nodes Z, first order all of the nodes not contained in Z. Associate with each node not in Z a bit value: 1 if the node is in the subgraph, 0 if it is not. Order the bits in the same order as the nodes not in Z. These bits may be interpretted as an integer between 0 and $2^{|V-Z|}-1$, with each different subgraph being represented by a different integer. Thus the subgraphs are enumerated.

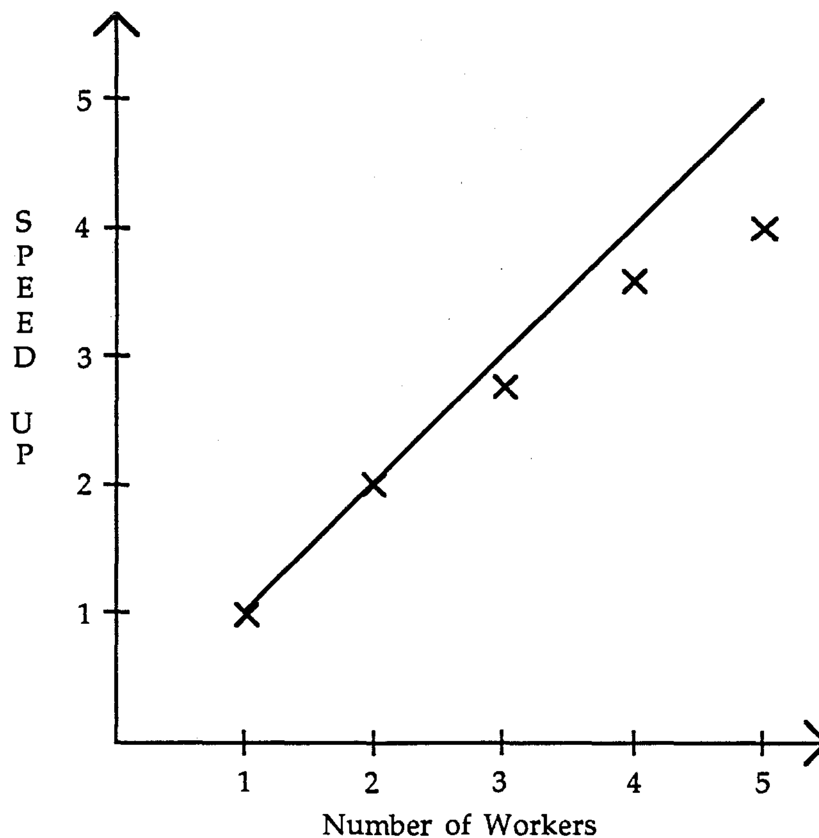

Figure 4.3: Time required to solve Steiner Tree Problem shown in Figure 4.1 vs. number of worker processes used.

One factor which breaks down the speedup is the distribution of data at the start of the algorithm which is proportional to the number of worker processes. Thus, as more worker processes are added and therefore the time required for the computation decreases, the time required for distributing the data increases. As workers are added, a point is eventually reached where the increase in cost of distributing the data to another worker is more than the decrease in computation brought by the extra worker[5].

[5] Note that if a reliable broadcast mechanism were available, the cost of distributing the data would be effectively independent of the number of workers, so the speedup would not degenerate so significantly as workers are added.

5. An Example: Bitonic Merge

5.1. Problem Definition

As was shown in the previous section, a MOOSE can be used to achieve efficient data sharing. A MOOSE is also quite versatile, as illustrated in this section where a MOOSE is used to implement a simple parallel sorting algorithm using a bitonic merge.

A list of numbers is said to be 'bitonic' if some rotation of the list may be split into two pieces - one increasing and one decreasing. For example, the list (4,2,1,3,5,7,8,6) is bitonic, since it may be split into two pieces — (1,3,5,7) and (8,6,4,2) — one increasing and one decreasing.

The algorithm given below takes a bitonic list and sorts it into ascending order. The algorithm can be generalized to handle any list of integers by first ordering the input list into a bitonic list, but for the sake of brevity this will not be shown.

The merge algorithm is best described with a picture. Before the picture is presented, some notation used in it must be explained. The device shown in Figure 5.1 is called a 'comparator'. It takes two inputs and swaps them if necessary so that the second output is not less than the first.

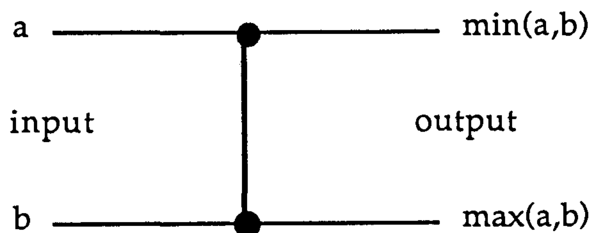

Figure 5.1: a comparator

A set of these comparators can be used to create a network which performs a bitonic merge for N elements, where N is a power of 2. A network for N=8 is shown in Figure 5.2. In order to sort N elements of a list, the algorithm requires $\log_2 N$ stages, with each stage consisting of $N/2$ parallel comparison/switch operations. Certainly the bitonic merge algorithm is not the most efficient parallel sorting algorithm ever devised, requiring $\log_2 N$ rounds and $N/2$ switch processes, but it serves quite well to demonstrate the versatility of a MOOSE.

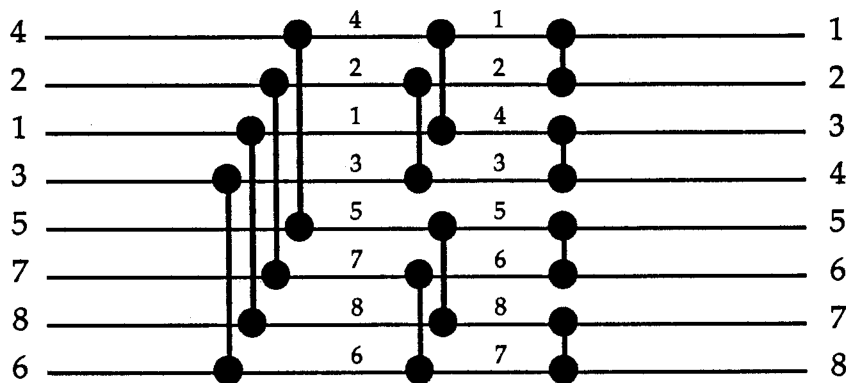

Figure 5.2: a bitonic merge network for 8 elements

The most efficient implementation of the bitonic merge using a MOOSE, in terms of the number of messages sent, does not allow an in-place sort (an 'in-place' sorting algorithm sorts a given list without requiring more memory than that necessary to store the list — perhaps necessary if memory is a critical resource). When the sort is completed, a copy of the list for each stage will exist at each node in the network. Implementing an in-place sort requires deleting objects from the object space, thus requiring considerably more messages than the algorithm which does not conserve memory; at any time during the execution of the in-place sort, only a single copy of the list will exist.

5.2. Computational Model

The data objects used in the examples have the form:

('element', integer R; integer I; integer X)

where R is the round of the sort algorithm, I is the index of the element in the list, and X is the value of the element.

The algorithm for the sort which does not operate in-place is given below:

```
proc bitonic_merge(r,n,m)
   if m>n ->
      co (i := n to (m+n-1)/2) switch(r,i,i+(m-n+1)/2)
      oc

      co bitonic_merge(r+1,n,(n+m-1)/2)
      // bitonic_merge(r+1,(n+m+1)/2,m)
      oc
   fi
end

proc switch(r,i,j)
   READ('element',r,i,integer X)
   READ('element',r,j,integer Y)
   if X<Y ->
      PUT('element',r+1,i,X)
      PUT('element',r+1,j,Y)
   [] else ->     /* X>Y */
      PUT('element',r+1,i,Y)
      PUT('element',r+1,j,X)
   fi
end
```

Note that although it is not shown in the code above, the concurrent invocations of `switch` and `bitonic`_merge should be performed on different machines. The in-place version of this algorithm is identical to the version given above, except that the `READ` operations in the `switch` proc are replaced by `GET` operations. Thus the data generated in each round is removed as the data for the next round is produced.

5.3. Configuring the MOOSE

The analysis of the algorithms requires specification of the different update modes of the servers. First note that all of the work is done in the `switch` opera-

tion. The `bitonic_merge` operation is present only to invoke the `switch` operations in the correct order. Also, the `bitonic_merge` operation is dormant while the switch operations it invokes are in progress — thus one of the invoked switch operations can take place on the same processor that the `bitonic_merge` operation is executing on, without slowing either process down. Thus, every node in the network will at one point have a `switch` operation executing on it, and this means that all nodes will have the same configuration regarding update modes and activities. Furthermore, the `switch` operation both produces and consumes, so the configuration must include both activities.

Second, note that arbitrary cached updates are out of the question, since waiting in one round for the cached results of the previous round to propagate is not feasible. In the in-place sort, cached deletes might be considered, except that the sort would then not be "in-place" (if at round 12 the deletes sent out in round 6 have still not propagated, a large amount of memory may be wasted). The application using a MOOSE structure has control over the caching of updates, however, so caching is not 'arbitrary'. Notice that in each round, each switch operation requires two `PUT`s and two `READ`s (`GET`s if the sort is in-place). If the cache sizes are set to two for both cached add update mode and cached delete update mode, then we are guaranteed that the caches will be flushed after each switch operation.

Third, note that each element of the list produced in one round is the object of only one `READ` (`GET`) operation in the next round. This means that if the algorithm operates in lazy or cached delete mode, there can be no 'cache misses'. The object selected by the `READ` (`GET`) operation can not have been deleted by another server, so a minimal amount of work is required (in the general case, an arbitrary number of cache misses can occur before an object still present in the object space is found).

Also, because of the one-to-one correspondence between PUT and GET operations in the in-place sort, 'risky' correctness will suffice (remember that using risky correctness saves 2 messages on each GET operation).

To determine the best settings for the update modes, refer back to Table 3.1 which gives the cost in messages of the three operations (READ, GET and PUT) executing on a network of δ MOOSE servers, in each of the nine different combinations of update modes. In this case the cache sizes C_a and C_d will be 2. Note that for a list of M elements, $^M/_2$ switch operations execute concurrently. Thus if $M>2\delta$, it will be necessary to have MOOSE servers serve more than a single switch operation (i.e. more than one switch operation will be executing on the same machine concurrently).

For a list of M elements, there are $\log_2 M$ rounds in the bitonic merge algorithm, with each round requiring M PUTs and M READs (or GETs for an in-place sort). In total, therefore, there are $M\log_2 M$ PUTs and $M\log_2 M$ READs ($M\log_2 M$ PUTs and $M\log_2 M$ GETs for the in-place algorithm). Thus, the best update mode combination for the algorithm which does not sort in-place is the cached add/eager delete combination which requires a total of $^{(\delta-1)}/_2 * M\log_2 M$ messages. The optimal settings for the in-place algorithm depend on δ. If $\delta > 6$, then the best algorithm uses cached add/cached delete update modes, requiring $(\delta + {^3/_2})M\log_2 M$ messages. If $\delta < 6$, then the best algorithm uses cached add/eager delete update modes, requiring $({^3/_2}\delta - {^3/_2})M\log_2 M$ messages (note that cached add/lazy delete update mode gives the best message count, but does not provide an in-place sort since objects are not deleted from local memory stores from one stage to the next).

6. Alternative Methods for Improving Efficiency

6.1. Consistency Requirements

In [Cheriton, 1985 and 1986], the possibilities for improved efficiency through relaxing consistency were discussed. In a distributed data structure, a piece of data might be produced at one host in the network, be propagated to the other hosts, and eventually be consulted at a different host from which it was produced. The primary way of relaxing consistency in such a system is to relax the requirements on whatever fetch operation is used to consult the data. The strongest form of consistency would require that a piece of data be locked before it is modified so that no other host can read the data until the task of propagating the update to the other hosts is complete.

A slightly less strict form of consistency would allow fetch operations at any time, but would attempt to propagate updates as quickly as possible. In this situation, fetch operations performed after the update was made might return stale data, but much improvement in efficiency is gained by not having to synchronize all data structure update operations on all hosts.

Further relaxation of consistency could be made by allowing fetch operations at any time, and storing up data structure updates until several have been made before propagating the updates to other hosts. Using this propagation method, it might take several seconds for an update to propagate to the other hosts in the network. This method improves performance by propagating several data structure updates to a host with a single message. If the cache of updates does not occupy more LAN packets than a single update, then the single message used to propagate several

updates will not be significantly more expensive than one of the messages needed to propagate a single update.

In the design of MOOSE, consistency was not very important because the operations used to access the MOOSE data structure do not include an in-place write operation (Cheriton's *store* operation). Hence consistency is easier to maintain. Loosening the consistency constraints on the MOOSE operations does not result in a significant gain in efficiency. The "correctness modes" used in MOOSE are in effect a relaxation of consistency requirements on the `GET` operation. Using risky correctness mode saves, at most, two message transfers per `GET` operation. Saving two messages could be important in a system with a small number of producers, but if a large number of MOOSE servers produce data, the relative savings are not significant.

6.2. Granularity of Data Distribution

6.2.1. Limitations to the Prototype MOOSE

In the MOOSE system, the information provided by the application is primarily used to increase efficiency through control of data distribution and replication. While the information that an application provides to a MOOSE gives a fair degree of flexibility concerning how data is to be distributed and replicated, the granularity of the distribution is at the object form level — a given MOOSE server's local store for an object form contains either all of the objects of that form in the object space, or just those produced locally. This method of distributing data is simply too coarse grained for some applications.

For example, in the Bitonic Merge example, the control provided by the MOOSE was not sufficient to allow a good solution. In the solution to the Bitonic Merge, all of the data was distributed to every node in the network, despite the fact that it would be simple for a process to algorithmically determine where the data it has just produced will be needed. By distributing the data to all servers in the network, a great deal of memory and communications resources are wasted.

To reduce the granularity of the distribution, if the application can provide information as to which objects of a given form will be required at which host site, the programmer could set up a different object form for each different server. Then, given an object to be placed into the object space, a MOOSE server could place it into the object form for each server which will later READ or GET the object. This is not a reasonable solution, however — it is too complicated, especially if more than one server will want to GET a given object.

6.2.2. Reducing the Granularity of the Distribution of Data

A better solution to reduce the granularity is to give the application more direct control over data distribution. Rather than providing the data structure servers with information pertaining to each other's update modes, the application could, with each PUT operation, describe to which servers the update is to be forwarded. This would allow the application complete control over data distribution, but would complicate application programming and the implementation of the other operations.

Consider the work necessary to complete a READ operation if the granularity of distribution were at the object level instead of at the object form level. Since the application controls the data distribution, there can be no guarantees as to whether

or not a given object will be stored locally. If checking the local store fails, the READ operation would have to be forwarded to all producers of the given object form (as is currently done with lazy add servers[6]); forwarding the operation in this manner is expensive if several servers produce the objects.

The application might be able to provide more information, however. The READ and GET operations could be modified so that the application specifies whether the operation is to search for the object locally, or to forward the search onto other servers if necessary (and to which servers the operation should be forwarded, if necessary). Using a MOOSE modified in this fashion, the Bitonic Merge algorithm could be written so that no messages or memory are wasted — all READs and GETs could be performed locally, and the data could be forwarded only to that server which would need it in the next round of computation.

6.2.3. Server Mapping Functions

To achieve this functionality, the application defines a function which maps an object to the set of servers which require immediate ("eager") notification when the object is PUT into the object space (a one-to-many relation). A different function could be defined for each object form (although object forms are not really necessary any longer). These functions would have to be defined locally at every server so that they would not need to be invoked remotely (costing at least 2 messages). With these functions available, distributing the data as it is produced is quite simple — the functions map the object which has been PUT to all of the servers to which the object must be sent. Performing a READ or GET is a little more complicated, but the

[6] unless there is an eager-add server in which case the operation is optimized through function shipping to the eager-add server.

same mapping function could be used; the application does not have to provide any more information.

The READ operation would have to be implemented in a manner similar to that shown below, where the function mapping objects to servers is named send_to_set.

```
READ(X)
   if me ∈ send_to_set(X) →
      perform READ locally
   [] send_to_set(X) ≠ ∅ →
      choose S ∈ send_to_set(X)
      forward READ to server S
   [] else → /* send_to_set(X) = ∅ */
      fa (S st S produces objects of X's form) →
         S.owner_read(X)
      af
   fi
```

This suggested implementation of MOOSE operations provides increased data distribution control over the prototype implementation by giving the application explicit control over where individual objects are sent on a PUT operation (data shipping) and whether or not READ and GET operations are forwarded to all or some producers (function shipping). This can lead to increased efficiency.

As with the existing MOOSE implementation, information pertaining to the distribution of data is an implicit part of the object form, defined when the form is defined (through update modes in the prototype implementation, and through the server mapping functions in this suggested implementation).

If several servers are producing objects of a given form, and a server performing a READ operation is not getting immediate updates of those objects, the READ may be forwarded to any set of servers which among the set will receive immediate notification should an object matching the template given in the READ operation be PUT

into the object space (function shipping). In the prototype, the READ is forwarded to all of the producer servers with one crude form of this optimization: if any server is eager-add with respect to an object form, servers which are lazy-add with respect to that object form may ship their READ operations to the eager-add server.

While this suggested implementation brings a great deal of improvement over the prototype MOOSE implementation, there is one major complication: the server mapping functions map a given object to a set of servers, but the parameters to the READ and GET operations are not necessarily fully defined objects.

Consider a system consisting of multiple servers. Several servers produce objects with the form ('data', int, int). The first server receives all of these objects with value (1,1). The second server receives all objects with value (1,2). The third server consumes these objects. Assume that the third server starts a READ operation:

```
READ('data',1,1)
```

This operation may be forwarded to the first server, since the first server has a complete store of all objects which match the object template of the READ operation. However, if the READ operation were:

```
READ('data',1,int i)
```

the first or the second server, or both together will not necessarily have a complete store of all objects which match this template. The operation would have to be forwarded to all producers. But if it were known that only objects with a '1' or a '2' as the second value would be PUT then the command could be forwarded to just the first and second servers. How could the application inform the MOOSE system of that fact?

The server mapping functions might be considered to define the set (possibly infinite) of objects which will be sent to a given server, if they are produced (the *object set* of the mapping function). The object template also defines a set: the set of objects which could match the template (the object set of the template). Ideally, the READ and GET operations could look at the mapping functions, determine the minimal set Ψ of servers such that the object set of the template is a subset of the union of the object sets of Ψ, and forward the operation to the set of servers Ψ. This problem is certainly non-trivial, and solving it could cost more than could possibly be saved by reducing the number of messages.

There are several solutions. The READ and GET operations could simply not make full use of the information provided by the mapping functions — if possible, the operation would be performed locally, otherwise it would be forwarded to all of the producer servers (as is done in the general case by the prototype implementation). The problem is now reduced in magnitude, although it is still necessary to determine if the object set of the mapping function of the server executing the READ or GET spans the object set of the object template in order to determine if the operation may be executed locally or must be forwarded to the producers of the object form. This problem is not difficult, and the solution could be supplied by the application (perhaps as a function similar to the mapping function). As an extension, the same method could be used to determine if the object set of the template is spanned by the object set of any one server, and thus determine if there is any single server to which the operation could be forwarded.

Consider the example given above. The object set of the mapping function for server 1 only contains the object with value (1,1), while the object set of the mapping function for server 2 only contains the object with value (1,2). The object set of the

object template ('data',1,int i) contains an infinite number of objects, such as (1,1), (1,2), (1,3), etc. Obviously, no single server has a mapping function with an object set which is a superset of the object set of the template, so the operation cannot be forwarded to any one server — the operation must be shipped to all of the producer servers.

Alternately, the mapping functions could be restricted in such a way that the problem of finding the minimal set Ψ of servers would not be difficult. One way to do this would be to restrict the mapping functions in the following manner: a server would immediately be sent all objects with a particular value in a particular position as they are PUT into the object space (for example, a server might receive all objects with a 3 as the first value). With such a restriction, the problem of determining the minimal set Ψ would not be difficult for a small number of servers. In this solution, each MOOSE server would need to know the details of each mapping function (the mapping functions could not be "black boxes").

6.2.4. Example of Server Mapping Functions

For an example, consider the Jacobi method for solving the two-dimensional Laplace's Equation. Laplace's Equation leads to a matrix manipulation problem which the Jacobi method solves with an iterative technique. The trial value for the ith cell of matrix Φ in the kth iteration of the algorithm (denoted $\phi_i^{(k)}$) may be obtained by taking the average of the values of the neighboring cells from the previous iteration. This may be written as follows:

$$\phi_i^{(k)} = \frac{1}{4}\left[\phi_{i-x}^{(k-1)} + \phi_{i-y}^{(k-1)} + \phi_{i+x}^{(k-1)} + \phi_{i+y}^{(k-1)}\right]$$

In the Jacobi method, in a given iteration each cell can be updated independently of all other updates. Consider partitioning the matrix Φ between processors as shown in Figure 6.1.

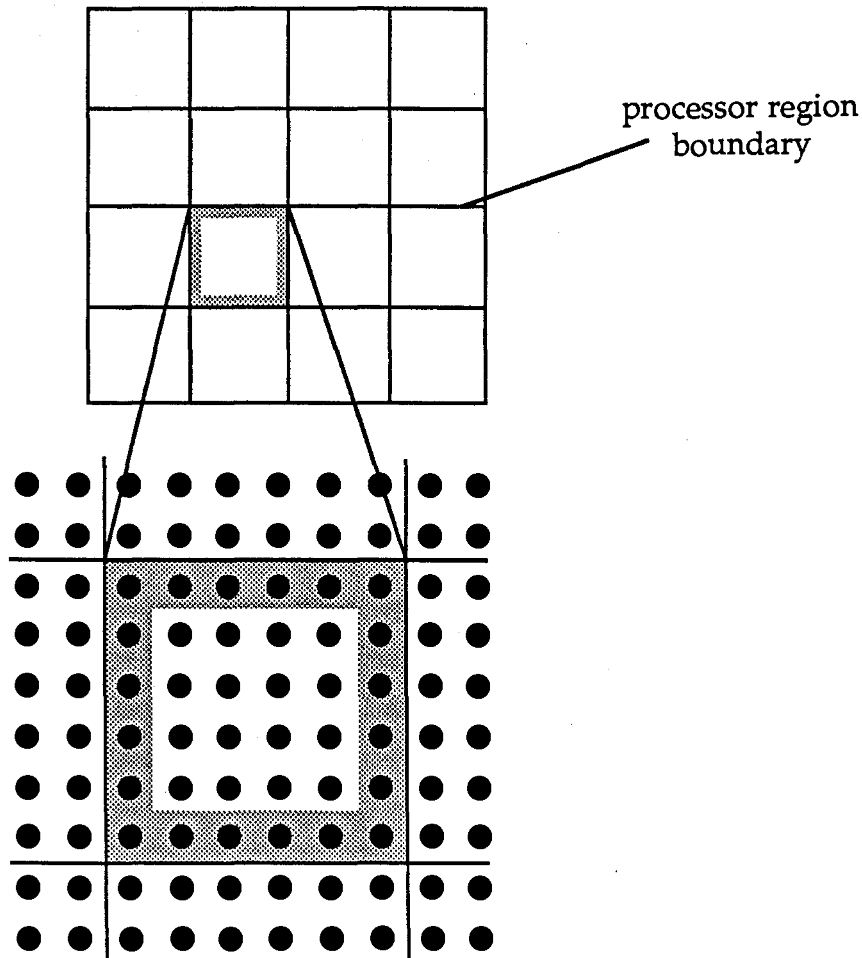

Figure 6.1 : partitioning of matrix between processors. Updates of cells within shaded regions require information from neighboring processors.

Using the problem decomposition implied by the partitioning of the matrix shown in Figure 6.1, each processor must, at each stage of the computation, compute the new values of the matrix cells and then send the edge values to the neighboring processors. An optimal implementation of this algorithm will obviously send the edge values as soon as they have been computed, and will send them only to the processor which needs those edge values for its own computation.

Using the prototype MOOSE, such fine-grained distribution of the data would be difficult to achieve. Consider instead how it would be done using server mapping functions. Enumerate the edges of each of the regions (4 edges per region). When an edge is `PUT` into the object space, attach the number of the edge to the start of the list of values of the object. The server mapping functions would be defined so that an edge `PUT` into the object space will be forwarded to the MOOSE server which serves the process that will require that edge in the next stage of computation. For example, assume that 4 processors are working on the problem. Assume that the problem has been decomposed and that the processors and edges are numbered as shown in Figure 6.2.

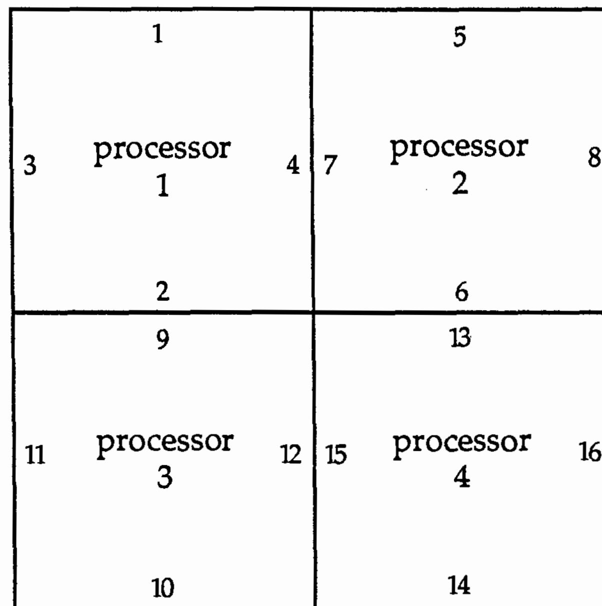

Figure 6.2 : example problem decomposition and numbering of processors and edges.

For the decomposition and numbering shown in Figure 6.2, and assuming that the edge number is attached to the start of the edge object, the server mapping function

for the edge object form is given in Table 6.1. With this server mapping function, edges are propagated only to where they are needed.

first value of object	send-to set
1	Ø
2	{3}
3	Ø
4	{2}
5	Ø
6	{4}
7	{1}
8	Ø
9	{1}
10	Ø
11	Ø
12	{4}
13	{2}
14	Ø
15	{3}
16	Ø

Table 6.1 : server mapping function for a program implementing the Jacobi method and using the problem decomposition and processor/edge enumeration given in Figure 6.2.

Although, for the Jacobi method, a very simple server mapping function was required (send the object to server α if the first value of the object equals β), this form of server mapping function is quite limited. Further research on possible restrictions to the server mapping functions is necessary.

6.3. Exploiting Piggybacking

One method of reducing communication costs is to send two or more logically different messages in a single network packet ("piggybacking" the messages). A MOOSE provides one way of piggybacking messages — through cached update modes where several data structure updates are sent together in one message. The cached update modes are a very restricted form of piggybacking, however. The diffi-

culty in providing for more general piggybacking comes from a lack of information: how is it possible to determine whether a message needs to be sent immediately or whether it can be held and sent later? To answer this question, information on the future actions of the application are necessary.

A restricted form of piggybacking may be provided by allowing the application to specify whether or not a message needs to be send immediately. A distributed data structure which provided for piggybacking messages could implement a DELAY operation. The purpose of the DELAY operation would be to tell the data structure server that any message generated from the next operation need not be sent immediately, perhaps because another operation would follow shortly which would also generate messages. Thus the data structure server could piggyback the messages.

Consider the worker process in the Steiner Tree example given in Section 4. The worker executes the following loop:

```
do (true) ->
   GET a task token
   fa (subgraphs specified by the token) ->
      build the subgraph from the original graph
      determine the minimum weight spanning tree
   af
   PUT least of all minimum weight spanning trees
od
```

For each iteration of this loop, three messages are sent across the LAN: a message forwarding the GET operation to the master server (the server which produced the tokens), a message from the master server returning a token, and a message to the master server returning the result of the computation. Of these three messages, the two going to the master always occur one after another — the data from a computation is sent, and then a GET for another token is forwarded. If a DELAY operation were implemented in MOOSE, then these two messages could be piggybacked,

resulting in a savings of 33% of the messages generated in the work phase of the computation!

6.4. Automating Selection of Update Modes

In order for a MOOSE to provide an efficient data structure, the various parameters must be correctly specified. While the task of determining the parameters which give best performance is not necessarily difficult, it is certainly not trivial. This leads one to consider the possibility of automating the process.

Presently, the application specifies which nodes in a network are going to produce data, and which are going to consume data. If the application also specified what percentage of the data a consumer was going to consume or a producer was going to produce, it would be very easy to determine an optimal assignment of eager or lazy mode to the add and delete update modes of each server. It was shown earlier that if a node consumes a fraction of the objects of an object form greater than $1/2P$, where P is the number of producers, then it is more efficient to make that server eager-add. If the server consumes less than $1/2P$ of the data, then it should be made lazy-add.

The use of cached update modes complicates this simple scheme, however. When is it appropriate to use a cache? How big should the cache be? How long should the timeout be between flushing of the cache? These questions cannot be answered simply. Consider the Bitonic Merge example. A cache size of two was used because each stage of the computation caused two objects to be generated by each worker process. This is a property of the algorithm used, and is readily apparent to the programmer, but how can it be automatically determined?

One possibility is to collect statistics on how frequently objects are produced, and how frequently they are consumed. If the variance of the time between consumption of objects by a particular server is small, then the cache size for that server could be set large with the timeout set equal to the mean time between object consumption. This would possibly result in near optimal efficiency, but there are several problems. The program execution must be closely observed — this is not a trivial task in itself. The assumption must be made that statistics gathered accurately reflect future action of the program — this is not necessarily the case. The method is only useful for programs which demonstrate a low variance in the mean time between object consumption — this is a rather restricted class of problems.

While automating the task of determining the update modes may be possible, doing so would conflict with two of the basic principles of the MOOSE system: efficiency and generality. The automated process would not, in most cases, be as effective in determining the optimal values for the update modes as would the programmer who builds the application. Also, the automated process would be effective only for problems which demonstrated those properties mentioned above.

7. Conclusions

7.1. Conclusions

A data structure general enough to serve a wide variety of applications, but independent of those applications, will not provide an efficient service. In designing a data structure for a particular application, a programmer makes use of information specific to that application. An efficient data structure cannot be independent of the application it serves.

In this thesis it has been shown that by using semantic information provided by the application, a data structure can be both versatile and efficient. To demonstrate this principle, the MOOSE data structure was designed and a prototype was implemented. An application which makes use of a MOOSE must provide information on where the data is going to be produced and consumed, and how the data is to be replicated. The MOOSE servers make use of this information to select between a variety of algorithms to implement the data structure, and thus an efficient service is provided. The efficiency and versatility of the MOOSE structure was demonstrated through two applications: a brute force approach to solving the Steiner Tree Problem and a simple parallel sorting algorithm.

7.2. Further Research

The MOOSE data structure has proved its value in the implementation of a few different algorithms. In particular, the token/worker model of computation is particularly easy to implement using a MOOSE. The token/worker model is applicable to any problem in which the task as a whole may be partitioned into a large number of independent smaller tasks. One aspect of this computational model which

deserves a closer look is the granularity of the task partitioning. The automatic load balancing of the token/worker model improves as the number of subtasks increases. If the subtasks are too small, however, the communications overhead of distributing the task will be too high. The optimal granularity for the distribution of a problem could be determined through experimentation.

While a MOOSE succeeds in providing an efficient distributed data structure for many applications, certain problems cannot be solved with optimal efficiency using a MOOSE. The coarse-grained approach to data distribution and replication used by the MOOSE system is not sufficient for some applications. More finely grained approaches were looked at, but all led to complications of the MOOSE operations. The most promising of these alternate means of controlling data distribution used the server mapping functions. The problem with this approach is that the `READ` and `GET` operations are complicated tremendously if a general mapping function is used. An interesting course of study would be to look at restrictions to the server mapping functions which would allow a feasible implementation of the `GET` and `READ` operations.

The most important aspect of this thesis has been the use of semantic information to improve efficiency in a distributed data structure. The MOOSE data structure requires that the application provide information pertaining to where the data is going to be produced or consumed, and how the data is to be distributed. A few very important questions must be raised: is there any other information that an application could provide that would be useful in improving efficiency? If further information could be provided, how could it be used?

Appendix A : Overview of the SR Programming Language

This appendix will describe enough of the SR programming language to help someone unfamiliar with the language understand the code and pseudo-code given throughout this proposal. For a complete description of the language, see [Andrews 87].

A.1. Operations and Invocations:

SR provides operations, which are generalizations of procedures. There are two ways of implementing an operation: with a `proc` or an `in` statement. The `proc` is equivalent to standard procedures. Each time the `proc` is invoked, a new process may be created (although some optimization is done — local invocations of procs are implemented as standard procedure calls). An `in` statement is in effect an instruction for a process to block and wait for an invocation of the named operation. The `in` statement unblocks when such an invocation is received.

In the description of the implementation, `in` statements were used with an enclosing infinite loop surrounding the statement. Thus the process will repeatedly service invocations of the operation named in the `in` statement. With this scheme, only one invocation can be executing at any one time. Thus `in` is used when, for reasons of concurrency control, only one invocation of the operation can exist at one time, forcing other invocations to wait. A `proc` is used when many invocations can execute concurrently.

Operations may be invoked in either of two manners: the `call` or the `send` statement. The `call` statement provides synchronous communications — the `call`

statement invokes the operation and then blocks the caller until the operation is complete. The `send` statement provides asynchronous communications — a message to start the operation is sent, and then the sender continues without waiting for the operation to complete (or necessarily even start, if the operation is implemented with an `in` statement).

The various combinations of `call/send` and `proc/in` provide four different mechanisms for invoking operations:

invocation	service	effect
call	proc	procedure call
call	in	rendezvous
send	proc	dynamic process creation
send	in	message passing

The granularity of distribution in the SR language is at the resource level. Operations and data are contained within resources. A resource instance may be created on any virtual machine in the system — virtual machines are mapped to real machines at run time, not necessarily on a one-to-one basis. In order to invoke an operation remotely (i.e. in a different resource), it is necessary to have the capability for the resource instance implementing that operation. The capability may be prefixed to the name of the operation in the invocation statement. For example, the invocation:

```
call cap_a.foo(. . .)
```

invokes the `foo` operation in the resource instance defined by the capability variable `cap_a`. If no capability is specified in an invocation, the operation is invoked in the same resource as the invoking statement.

In addition to basic communication facilities, SR provides the `co` statement to invoke several different operations concurrently. The syntax of the co statement is as follows:

```
co (quantifier) invocation [-> statement block]
// (quantifier) invocation [-> statement block]
   . . .
oc
```

Each operation specified by the invocation parts of the statement (over the different values of the quantifiers) are invoked concurrently with call semantics. As each invocation returns, the statement block following the invocation portion is executed. The `co` statement terminates when all invocations have returned, and all statement blocks have finished executing.

A.2. Sequential statements

There are many different sequential mechanisms in the language used throughout this proposal. They will only be mentioned briefly, as their semantics are quite straightforward.

The `fa` statement is a generalization of the standard "loop a fixed number of times" mechanism (such as the `for` statement in pascal). The `fa` statement has the form :

```
fa (quantifier) st (boolean expression) ->
    statement block
af
```

where the statement block is executed for each value of quantifier which satisfies the (optional) boolean expression.

The `if` statement has the form:

```
if condition -> statement block
[] condition -> statement block
  . . .
fi
```

where the last condition can optionally be 'else' which is taken as the conjunction of the negations of all the other conditions. The conditions are evaluated in a nondeterministic order. If one is found to be true, its statement block is executed and the statement terminates.

The `do` statement has the form:

```
do condition -> statement block
[] condition -> statement block
  . . .
od
```

Each condition is evaluated (the order is non-deterministic). If one is found to be true, its associated statement block is executed and the operation repeats. The `do` statement terminates when all conditions are false.

Appendix B: MOOSE Implementation

```
global types

   /* this global resource is imported to all other resources */

   /* maximum number of values in an object */
   const max_obj_size := 30

   /* various system constants */
   const infinity := 1000
   const undefined := -1
   const safe := 1
   const risky := 2

   /* object form name type */
   type nametype = string(10)

   /* record of information for an addition to a local database */
   type add_op = rec(
      name : nametype
      values[1:max_obj_size] : int
      o_id : int)

   /* record of information for a deletion from a local database */
   type del_op = rec(
      name : nametype
      values[1:max_obj_size] : int
      o_id : int)

end
```

```
resource moose

   /* this is the main resource which the application imports */

   import types


   op init_moose(servers[1:*] : cap moose; whoami : int)

   op put(form:types.nametype; values[1:*] : int) returns b:bool
   op get(form:types.nametype; var values[1:*]:int) returns b:bool
   op read(form:types.nametype; var values[1:*]:int) returns b:bool

   op def_form(f:types.nametype; s:int; c:int; pr[1:*]:bool;
           con[1:*]:bool; as[1:*] : int; ad[1:*] : int; ds[1:*] : int;
           dd[1:*] : int) returns b:bool


   op def_form_sub(f:types.nametype; s,c:int; pr[1:*]:bool;
        con[1:*]:bool; as[1:*]:int; ad[1:*]:int; ds[1:*]:int;
        dd[1:*]:int)

   op local_add(name:types.nametype; values[1:*]:int; o_id:int)
   op cache_update(c[1:*] : types.add_op)
   op cache_delete(c[1:*] : types.del_op)
   op owner_delete(o_id:int; name:types.nametype; values[1:*]:int)
                                                   returns ok:bool
   op global_delete(f:int; o:int; n:types.nametype; v[1:*]:int)
   op local_delete(name:types.nametype; v[1:*]:int; o_id:int)
   op still_there(name:types.nametype; v[1:*] : int; o_id:int)
                                                   returns b:bool

   op read_work(n:types.nametype; var v[1:*]:int; res o_id:int)
                                                   returns b:bool
   op start_owner_read(name:types.nametype; var v[1:*]:int;
           res o_id:int)
   op owner_read(name:types.nametype; v[1:*]:int; r:cap moose; i:int)
                                                   returns t:int
   op return_owner_read(rn:int;v[1:*]:int; o_id:int; f_index:int)
   op cancel_owner_read(f_index:int)


body moose() separate
```

```
resource ticker

   /* a resource which ticks off seconds...  in order for this resource
      to work properly, it must be the only resource in a virtual
      machine.  Since the sleep operation forces the entire unix
      process (i.e. the entire virtual machine) to wait, creating this
      resource within an active virtual machine will no work properly */

   external sleep(t:int)
   op tick()

body ticker()

   proc tick()
      sleep(1)
   end

end
```

```
resource timer

   /* this resource is used by the cache-handling resources to time
      the waits between cache firings.  This resource creates another
      virtual machine on the same physical machine, and creates a
      ticker resource within that virtual machine                      */

   import ticker

   op snooze(t:int)

body timer()

   var v : cap vm
   var tick_res : cap ticker

   initial
      v := create vm()
      tick_res := create ticker() on v
   end


   var gen : int := 0
   var num_waiting : int := 0
   op tickmark(g:int)


   /* this process uses the ticker resource to tick off seconds, and
      informs any snooze procs of the ticks through invocations of
      the tickmark operation.  The variable 'num_waiting' keeps a count
      of how many snooze procs are waiting for ticks.  The tickmark
      invocations keep a generation number with them so that one
      snooze process can't grab all of the tickmarks from one
      clock tick                                                    */

   process t
      do (true) ->
         gen ++
         fa i := 1 to num_waiting ->
            send tickmark(gen)
         af
         num_waiting := 0
         tick_res.tick()
      od
   end


   /* this proc acts like a C sleep function for SR processes.  Since
      using the C sleep function would sleep the entire virtual machine,
      this proc and all of its support mechanisms had to be built     */

   proc snooze(x)
      var my_gen : int

      my_gen := gen
      do (x>0) ->
         num_waiting ++
```

```
            in tickmark(g) and g>my_gen ->
               x := x - (g-my_gen)
               my_gen := g
            ni
         od
      end


   end
```

```
resource addcachehandler

   /* this resource is used to handle a cache of add operations.  When
      a new object form is created, each moose server creates a
      different addcachehandler resource for each cached add server.
      The addcachehandler resource is given the capability to the
      moose server for which it is caching updates, the capability for
      the local timer resource, and the size and delay of the cache
      it must look after.  The only operation it exports is the
      cache_add operation which the moose server may call to add
      an entry into the cache.  When the cache is full, or the timeout
      expires, the cache is automatically fired off.                  */

   import types
   import moose
   import timer

   op cache_add(name:types.nametype; values[1:*]:int; o_id:int)

body addcachehandler(mres:cap moose; tres:cap timer; size,delay:int)

   /* the cache is simply an array of type add_op */
   var cache[1:size] : types.add_op
   var next_free : int := 1

   op fire_off_cache()
   op foc()


   /* this process implements the timeout.  Every 'delay' seconds, it
      causes the cache to be sent.                                    */

   process ticker
      do (true) ->
         tres.snooze(delay)
         send foc()
      od
   end


   /* this process is responsible for handling cache_add operations,
      and commands by the timer to send off the cache.  It is
      implemented with an in statement so that cache_add operations
      can't interfere with each other or with the sending of the
      cache.                                                        */

   process p1
      do (true) ->
         in cache_add(name,values,o_id) by 1 ->
            cache[next_free].o_id := o_id
            cache[next_free].name := name
            cache[next_free].values[1:ub(values)] := values
            next_free++
            if (next_free > size) ->
               fire_off_cache()
            fi
```

```
            [] foc() by 2 ->
               if (next_free > 1) ->
                  fire_off_cache()
               fi
            ni
         od
      end


      /* this proc is called to send and reset the cache */

      proc fire_off_cache()
         send mres.cache_update(cache[1:next_free-1])
         next_free := 1
      end


   end
```

```
resource delcachehandler

   /* this resource looks after caches of delete operations in the same
      manner that addcachehandler resources look after caches of add
      operations.  The two resources are in all manner parallel.     */

   import types
   import moose
   import timer

   op cache_del(name:types.nametype; values[1:*]:int; o_id:int)

body delcachehandler(mres:cap moose; tres:cap timer; size,delay:int)

   var cache[1:size] : types.del_op
   var next_free : int := 1

   op fire_off_cache()
   op foc()



   process ticker
      do (true) ->
         tres.snooze(delay)
         foc()
      od
   end



   process p1
      do (true) ->
         in cache_del(name,values,o_id) by 1 ->
            cache[next_free].o_id := o_id
            cache[next_free].name := name
            cache[next_free].values[1:ub(values)] := values
            next_free++
            if (next_free > size) ->
                  fire_off_cache()
            fi
         [] foc() by 2 ->
            if (next_free > 1) ->
               fire_off_cache()
            fi
         ni
      od
   end


   proc fire_off_cache()
      send mres.cache_delete(cache[1:next_free-1])
      next_free := 1
   end


end
```

```
resource formhandler

   /* each moose server creates a single formhandler resource to manage
      a local database of information about object forms.  Through the
      formhandler resource, every form is assigned a unique integer
      identifier (although different servers may assign different
      identifiers due to race conditions).                           */

   import types

   op new_form(f:types.nametype; size:int; c:int; pr[1:*]:bool;
         con[1:*]:bool; as[1:*] : int; ad[1:*] : int; ds[1:*] : int;
         dd[1:*] : int)

   op form_id(name:types.nametype) returns n:int
   op form_size(f_id:int) returns s:int

   op eager_add(form_id:int; server_id:int) returns b:bool
   op cached_add(form_id:int; server_id:int) returns b:bool
   op lazy_add(form_id:int; server_id:int) returns b:bool
   op eager_del(form_id:int; server_id:int) returns b:bool
   op cached_del(form_id:int; server_id:int) returns b:bool
   op lazy_del(form_id:int; server_id:int) returns b:bool
   op producer(form_id:int; server_id:int) returns b:bool
   op consumer(form_id:int; server_id:int) returns b:bool
   op correctness(form_id:int) returns c:int
   op num_producers(f:int) returns n:int
   op prodid(f:int) returns c:int

   external gethostname(res s:string(*); n:int)


body formhandler(me : int; max_servers:int; max_forms:int)

   type form_type = rec(
                        name : types.nametype
                        size : int
                        correctness : int
                        producers[1:max_servers] : bool
                        consumers[1:max_servers] : bool
                        add_cache_sizes[1:max_servers] : int
                        add_cache_delay[1:max_servers] : int
                        del_cache_sizes[1:max_servers] : int
                        del_cache_delay[1:max_servers] : int
                        num_prod : int
                        prodid : int
                        )
   var form[1:max_forms] : ptr form_type
   var next_form :int := 1


   /* given a form name, return its identifier */

   proc form_id(name) returns id
      id := 0
      fa i:=1 to next_form-1 st (name=form[i]^.name) ->
            id := i
```

```
        exit
   af
end


/* given a form identifier, return the size of the tuple of values */

proc form_size(f_id) returns size
   size := form[f_id]^.size
end


/* return true if the specified server is eager-add with respect to
   the specified object form, false otherwise.                     */

proc eager_add(form_id,server_id) returns b
   b := (form[form_id]^.add_cache_sizes[server_id] = 1)
end


/* return true if the specified server is cached-add with respect to
   the specified object form, false otherwise.                      */

proc cached_add(form_id,server_id) returns b
   b := (form[form_id]^.add_cache_sizes[server_id] != 1) and
      (form[form_id]^.add_cache_sizes[server_id] != types.infinity)
end


/* return true if the specified server is lazy-add with respect to
   the specified object form, false otherwise.                    */

proc lazy_add(form_id,server_id) returns b
   b := (form[form_id]^.add_cache_sizes[server_id] = types.infinity)
end


/* return true if the specified server is eager-del with respect to
   the specified object form, false otherwise.                     */

proc eager_del(form_id,server_id) returns b
   b := (form[form_id]^.del_cache_sizes[server_id] = 1)
end


/* return true if the specified server is cached-del with respect to
   the specified object form, false otherwise.                      */

proc cached_del(form_id,server_id) returns b
   b := (form[form_id]^.del_cache_sizes[server_id] != 1) and
      (form[form_id]^.del_cache_sizes[server_id] != types.infinity)
end


/* return true if the specified server is lazy-del with respect to
   the specified object form, false otherwise.                    */

proc lazy_del(form_id,server_id) returns b
```

```
   b := (form[form_id]^.del_cache_sizes[server_id] = types.infinity)
end


/* return true if the specified server is a consumer of the specified
   object form, false otherwise.                                    */

proc consumer(form_id,server_id) returns b
   b := form[form_id]^.consumers[server_id]
end


/* return true if the specified server is a producer of the specified
   object form, false otherwise.                                    */

proc producer(form_id,server_id) returns b
   b := form[form_id]^.producers[server_id]
end


/* return the correctness mode of the specified object form */

proc correctness(form_id) returns cor
   cor := form[form_id]^.correctness
end


/* return the number of producers of the specified object form */

proc num_producers(f_id) returns n
   n := form[f_id]^.num_prod
end


/* return the index of a moose server which produces the specified
   object form                                                   */

proc prodid(f_id) returns id
   id := form[f_id]^.prodid
end



/* define a new object form and all of its parameters */

proc new_form(name,size,corr,prod,con,addsize,adddelay,delsize,
            deldelay)
   form[next_form] := new(form_type)
   form[next_form]^.name := name
   form[next_form]^.size := size
   form[next_form]^.correctness := corr
   form[next_form]^.producers[1:ub(prod)] := prod
   form[next_form]^.consumers[1:ub(con)] := con
   form[next_form]^.add_cache_sizes[1:ub(addsize)] := addsize
   form[next_form]^.add_cache_delay[1:ub(adddelay)] := adddelay
   form[next_form]^.del_cache_sizes[1:ub(delsize)] := delsize
   form[next_form]^.del_cache_delay[1:ub(deldelay)] := deldelay
   form[next_form]^.num_prod := 0
```

```
        fa i := 1 to ub(prod) st prod[i] ->
           form[next_form]^.num_prod ++
           form[next_form]^.prodid := i
        af
        next_form := next_form + 1
     end

  end
```

```
resource dbhandler

   /* each moose server creates a single dbhandler resource to handle
      all of the local database.  The local database consists of a
      set of binary trees - one for each object form.  A semaphore
      exists for each binary tree.  Before any modification can be
      made to the tree, the modifying process must grab the semaphore.

      A wait_on_add operation exists for each binary tree.  A process
      can wait for an addition to an object form by executing a receive
      on the associated trees wait_on_add operation.  When an addition
      is made, the operation is invoked once for each process executing
      the receive operation.                                          */

   import types
   import moose
   import formhandler

   op local_add(n:types.nametype; v[1:*]:int; o:int)
   op local_delete(name:types.nametype; v[1:*]:int; o_id:int)
   op cache_update(c[1:*] : types.add_op)
   op cache_delete(c[1:*]:types.del_op)
   op checkwait(name:types.nametype; var values[1:*]:int; res o_id:int)
                                             returns b : bool
   op check(f_id:int; var values[1:*] : int; res o_id:int)
                                             returns b:bool
   op unique_check(f_id:int; v[1:*]:int; o_id:int) returns found:bool
   op invoke_ocw(n:types.nametype; v[1:*]:int; r:cap moose; rep:int)
                                             returns i:int
   op cancel_ocw(m:int)


body dbhandler(fres:cap formhandler; me:int; max_forms : int)

   type tnode = rec(
                    o_id : int
                    values[1:types.max_obj_size] : int
                    left : ptr tnode
                    right : ptr tnode)

   op sem[1:max_forms]()
   op wait_on_add[1:max_forms](g:int)
   var num_wait_on_add[1:max_forms] : int := ([max_forms] 0)
   var generation[1:max_forms] : int := ([max_forms] 0)

   op bt_add(var n : ptr tnode; values[1:*] : int; o_id:int)
   op obj_gt(v1[1:*] : int;  v2[1:*] : int) returns b:bool
   op cw(f_id:int; var values[1:*] : int; res o_id:int; p:ptr tnode)
                                                   returns b:bool
   op match(v1[1:*]:int; v2[1:*]:int) returns b:bool
   op ult(v1[1:*] : int; v2[1:*]:int) returns b:bool
   op ucw(v[1:*]:int; o_id:int; p:ptr tnode) returns found:bool
   op bt_del(var p:ptr tnode; v[1:*]:int; o_id:int)
   op prop_del(var p:ptr tnode)
   op attach(loose_p:ptr tnode; onto_p:ptr tnode)
```

```
var root[1:max_forms] : ptr tnode


initial
   /* initialize the semaphores */
   fa i:=1 to max_forms ->
      send sem[i]()
      root[i] := null
   af
end



/* add an object to the local database */

proc local_add(name,values,o_id)
   var f_id : int

   f_id := fres.form_id(name)
   /* grab the semaphore */
   receive sem[f_id]()
   /* add the object to the binary tree */
   bt_add(root[f_id],values,o_id)
   /* update the generation of the binary tree */
   generation[f_id] ++
   /* release the semaphore */
   send sem[f_id]()
   /* inform any waiting processes of the change */
   fa i:=1 to num_wait_on_add[f_id] ->
      send wait_on_add[f_id](generation)
   af
   num_wait_on_add[f_id] := 0
end


/* this proc is responsible for doing all of the work in adding
   an object to a tree                                          */

proc bt_add(p,v,o_id)
   if p=null ->
      /* we've found where to add it */
      p := new(tnode)
      p^.o_id := o_id
      p^.values[1:ub(v)] := v
      p^.left := null
      p^.right := null
   [] else ->
      /* descend down the tree recursively */
      if obj_gt(p^.values[1:ub(v)],v) ->
         /* descend down the left branch */
         bt_add(p^.left,v,o_id)
      [] else ->
         /* descend down the right branch */
         bt_add(p^.right,v,o_id)
      fi
```

```
      fi
   end



   /* define a total ordering over the values of objects */

   proc obj_gt(v1,v2) returns b
      fa i:=1 to ub(v1) ->
         if v1[i] > v2[i] ->
            b := true
            return
         [] v1[i] < v2[i] ->
            b := false
            return
         fi
      af
      b := false
   end



   /* receive and process a cache of add operations */

   proc cache_update(cache)
      fa i:=1 to ub(cache) ->
         send local_add(cache[i].name,cache[i].values,cache[i].o_id)
      af
   end


   /* receive and process a cache of delete operations */

   proc cache_delete(cache)
      fa i:=1 to ub(cache) ->
         send local_delete(cache[i].name,cache[i].values,cache[i].o_id)
      af
   end


   /* check for an object matching the specified template in the local
      database.  If a matching object is found, return it immediately.
      Otherwise, wait until such an object is found and then return
      it.                                                              */

   proc checkwait(name,values,o_id) returns b
      var f_id : int := fres.form_id(name)
      var last_gen : int

      if (f_id = 0) ->
         b := false
         return
      [] else ->
         b := true
      fi
```

```
    do (true) ->
       last_gen := generation[f_id]
       if check(f_id,values,o_id) ->
          return
       [] else ->
          num_wait_on_add[f_id]++
          in wait_on_add[f_id](g) & g>last_gen ->
             skip
          ni
       fi
    od
end


/* this proc starts up an operation similar to checkwait, except that
   only objects produced locally (i.e. this resource is the owner of
   the object) can be returned.  The proc is given a capability to
   a moose server to which it must return results.  This proc might
   receive a cancellation message before it completes (it may never
   complete short of being cancelled).                             */

var ocw_ctr : int := 1

proc invoke_ocw (name,values,rep_cap,rep_num) returns i
   var f_id : int := fres.form_id(name)
   var last_gen,my_ocw_ctr,o_id : int

   my_ocw_ctr := ocw_ctr
   ocw_ctr++
   i := my_ocw_ctr
   reply

   last_gen := generation[f_id]
   do (true) ->
      if check(f_id,values,o_id) ->
         /* we've found an object.  Send a reply and exit */
         send rep_cap.return_owner_read(rep_num,values,o_id,me)
         exit
      [] else ->
         num_wait_on_add[f_id] ++
         /* wait for either an addition to the tree or a cancellation
            message                                                  */
         in wait_on_add[f_id](g) & g>last_gen ->
            last_gen := g
         [] cancel_ocw(m) & m=my_ocw_ctr ->
            num_wait_on_add[f_id]--
            exit
         ni
      fi
   od
end


/* check the local database for an object matching the specified
```

```
   object template.                                                    */

proc check(f_id,values,o_id) returns found
   found := cw(f_id,values,o_id,root[f_id])
end


/* this proc does all of the work of the check proc - the check proc
   calls this proc with a pointer to the root of a tree.  This proc
   recursively descends the tree.                                         */

proc cw(f_id,values,o_id,p) returns found

   if(p=null) ->
      found := false
      return
   fi
   if match(values,p^.values[1:ub(values)]) ->
      found := true
      values := p^.values[1:ub(values)]
      o_id := p^.o_id
      return
   fi

   /* note that because of the possible undefined values in
      the object to be searched for, it may be necessary to search
      both children of the current node                                 */
   if ult(values,p^.values) ->
      found := cw(f_id,values,o_id,p^.left)
      if (found) ->
         return
      fi
   fi
   if ult(p^.values,values) ->
      found := cw(f_id,values,o_id,p^.right)
   fi
end



/* this proc returns true if the values of two objects match. The
   values may not be fully defined, in the case of an object
   template.                                                           */

proc match(v1,v2) returns b
   fa i:=1 to ub(v1) st (v1[i]!=v2[i] and v1[i]!=types.undefined and
                v2[i]!=types.undefined) ->
      b:=false
      return
   af
   b := true
end


/* determine if the first set of values might be less than the second
   set, based on the ordering defined by the obj_gt proc.  The set
```

```
   of values might be partially undefined (in the case of an object
   template)                                                        */

proc ult(v1,v2) returns b
   fa i:=1 to ub(v1) ->
      if (v1[i]=types.undefined or v2[i]=types.undefined or
                                              v1[i]<v2[i]) ->
         b := true
         return
      fi
      if v1[i] > v2[i] ->
         b := false
         return
      fi
   af
   b := false
end


/* this proc checks to see if the given fully defined object exists
   in the tree                                                      */

proc unique_check(f_id,values,o_id) returns found
   found := ucw(values,o_id,root[f_id])
end


/* this proc does all of the work for the unique_check proc.  The
   unique_check proc calls this proc with the root of a tree, and
   this proc recursively descends the tree looking for the object */

proc ucw(values,o_id,p) returns found
   if (p=null) ->
      found := false
   [] else ->
      if (p^.o_id = o_id) ->
         found := true
      [] obj_gt(p^.values[1:ub(values)],values) ->
         found := ucw(values,o_id,p^.left)
      [] else ->
         found := ucw(values,o_id,p^.right)
      fi
   fi
end


/* this proc removes the specified object from the local database */

proc local_delete(name,values,o_id)
   var f_id : int := fres.form_id(name)
   var last_gen : int

   /* make sure the thing is there (the delete may have arrived
      before the add)                                               */
   last_gen := generation[f_id]
   do (not unique_check(f_id,values,o_id)) ->
```

```
        num_wait_on_add[f_id]++
        in wait_on_add[f_id](g) & g>last_gen ->
           last_gen := g
        ni
     od

     receive sem[f_id]()
     bt_del(root[f_id],values,o_id)
     send sem[f_id]()
  end



  /* this proc removes an object from a binary tree */

  proc bt_del(p,v,o_id)
     if (p!=null) ->
        if (p^.o_id = o_id) ->
           /* the object has been found, delete the object and
              restructure the tree which was below the object */
           prop_del(p)
        [] obj_gt(p^.values[1:ub(v)],v) ->
           bt_del(p^.left,v,o_id)
        [] else ->
           bt_del(p^.right,v,o_id)
        fi
     fi
  end



  /* this proc is responsible for deleting the specified node from
     a binary tree. The tree below the deleted object must be
     restructured.                                               */

  proc prop_del(p)
     var pl : ptr tnode := p^.left
     var pr : ptr tnode := p^.right

     if (pl != null) ->
        /* there was a subtree to the left of the deleted node.
           Attach it where the deleted node was, and attach any
           subtree to the right of the deleted node to this new
           structure.                                         */
        p := pl
        if (pr != null) ->
           attach(pr,pl)
        fi
     [] else ->
        /* there was no subtree to the left of the deleted node,
           so the subtree to the right of the deleted node (if any
           is attached in place of the deleted node)              */
        p := pr
     fi
  end


  /* this proc attaches one subtree onto another, maintaining the
```

```
      correct ordering of the binary tree.                                 */

   proc attach(loose_p,onto_p)
      if (onto_p^.left != null) ->
         attach(loose_p,onto_p^.left)
      fi
      onto_p^.left := loose_p
   end



end
```

```
body moose

   /* this is the body for the moose resource, responsible for
      implementing all of the moose operations.  When this resource
      is created and initialized, it creates formhandler, dbhandler
      and timer resources to serve it.                              */

   import addcachehandler
   import delcachehandler
   import formhandler
   import dbhandler
   import timer



   const max_servers := 15
   const max_forms := 25
   var moose_server[1:max_servers] : cap moose
   var num_servers,me : int

   var formhan : cap formhandler
   var dbhan : cap dbhandler
   var timer_res : cap timer
   type chs = rec(server[1:max_servers] : cap addcachehandler)
   type dhs = rec(server[1:max_servers] : cap delcachehandler)
   var addcachehan[1:max_forms] : ptr chs
   var delcachehan[1:max_forms] : ptr dhs

   op od_sem[1:max_forms]()   /* owner_delete has to be mutually
                                 exclusive for each form so need
                                 to use a semaphore...            */



   initial
      fa i:=1 to max_forms ->
         send od_sem[i]()
      af
   end



   proc init_moose(servers, whoami)
      me := whoami
      num_servers := ub(servers);
      fa i:=1 to num_servers ->
         moose_server[i] := servers[i]
      af
      formhan := create formhandler(me,num_servers,max_forms)
      dbhan := create dbhandler(formhan,me,max_forms)
      timer_res := create timer()
   end



   op new_obj_id() returns o_id:int {call}
   op owner(o_id:int) returns owner:int
```

```
    op global_add(f:int; n:types.nametype; v[1:*] : int)



 /*********************************************************************/
 /* all of the following procs are invoked by other moose servers.  They
    simply forward a task on to the local dbhandler resource          */

    proc local_add(name,values,o_id)
       dbhan.local_add(name,values,o_id)
    end

    proc cache_update(c)
       dbhan.cache_update(c)
    end

    proc cache_delete(c)
       dbhan.cache_delete(c)
    end

    proc local_delete(name,values,o_id)
       dbhan.local_delete(name,values,o_id)
    end

    proc still_there(name,values,o_id) returns b
       b := dbhan.unique_check(formhan.form_id(name),values,o_id)
    end




    /* this proc defines unique identifiers for new objects created
       locally.  The id of the local moose server is coded into the
       object id                                                    */

    var cnt : int := 1

    proc new_obj_id() returns o_id
       o_id := me*100000+cnt
       cnt++
    end


    /* given an object id, this proc returns the id of the moose server
       which created and owns it                                         */

    proc owner(o_id) returns owner
       owner := o_id / 100000
    end



    /* this proc is the application interface to the procs which
       define new object forms.  This proc may be called from any
       moose server, and the new object form is created on all moose
```

```
      servers.                                                    */

   proc def_form(name,size,corr,con,prod,addsize,adddelay,delsize,
                                               deldelay) returns b

      /* this conditional is set up a little funny to avoid a bug in
         the compiler...                                         */
      if (size > types.max_obj_size) ->
         b := false
         return
      fi
      if (formhan.form_id(name) != 0) ->
         b := false
      [] else ->
         co (i:=1 to num_servers)
            moose_server[i].def_form_sub(name,size,corr,con,prod,
                        addsize,adddelay,delsize,deldelay)
         oc
         b := true
      fi
   end


   /* this is the proc which actually defines the new object form
      locally, and creates any addcachehandler and delcachehandler
      resources which need to be created.                      */

   proc def_form_sub(name,size,corr,con,prod,addsize,adddelay,delsize,
                  deldelay)
      var f_id : int

      formhan.new_form(name,size,corr,con,prod,addsize,adddelay,
                                    delsize,deldelay)
      f_id := formhan.form_id(name)
      addcachehan[f_id] := new(chs)    /* create the capabilities for the
                                 new cache handlers */
      /* create the new add-cache handlers */
      fa i:=1 to num_servers st
                  (addsize[i] != 1) and (addsize[i]!=infinity)
                     and (i!=me) ->
         addcachehan[f_id]^.server[i] := create addcachehandler(
                  moose_server[i],timer_res,addsize[i],adddelay[i])
      af
      /* create new del-cache handlers */
      fa i:=1 to num_servers st
            (delsize[i] != 1) and (delsize[i] != types.infinity) and
                                 (i!=me) ->
         delcachehan[f_id]^.server[i] := create delcachehandler(
                  moose_server[i],timer_res,delsize[i],deldelay[i])
      af
   end


/*********************************************************************/
/* this proc implements the get operation                            */
```

```
proc get(name,v) returns b
   var o_id : int
   var f_id : int := formhan.form_id(name)

   if f_id = 0 ->
      b := false
      return
   [] else ->
      if ub(v) != formhan.form_size(f_id) ->
         b := false
         return
      fi
   fi

   /* different actions for different correctness modes */
   if formhan.correctness(f_id) = types.safe ->

      /* have to make this test a little funny because of compiler
                                                              bugs */
      var Imaproducer : bool := formhan.producer(f_id,me)
      if (formhan.num_producers(f_id) > 1) or Imaproducer ->
         /* if there is more than one producer, or these objects
            are produced locally, perform the operation locally */
         var values[1:ub(v)] : int
         do (true) ->
            /* in safe correctness mode, we have to attempt to delete
               the object through the object's owner, with the
               owner_delete operation.  If the owner_delete fails, we
               have to try with a different object.                */
            values := v
            if (not read_work(name,values,o_id)) ->
               b := false
               return
            fi
            if moose_server[owner(o_id)].owner_delete(o_id,name,
                     values) ->
               v := values
               b := true
               return
            fi
         od
      [] else ->
         /* if there is only one producer, forward the call to that
            server                                                 */
         moose_server[formhan.prodid(f_id)].get(name,v)
      fi
   [] else ->     /* correctness = risky */
      if (formhan.num_producers(f_id) > 1) or
                 (formhan.producer(f_id,me)) ->
         /* if there is more than one producer, or these objects
            are produced locally, perform the operation locally */
         if (not read_work(name,v,o_id)) ->
            b := false
            return
         fi
         /* in risky correctness mode, we do not need permission of
            the owner to delete the object.                        */
         /* by causing the owner to execute a global delete, we
```

```
                guarantee that the object is removed from the owner's
                local db, regardless of his delete update mode.  This
                doesn't cost any extra messages than if we had done
                it all from here, either                                  */
             send moose_server[owner(o_id)].global_delete(f_id,
                                                          o_id,name,v)
          [] else ->
             /* if there is only one producer, forward the call to that
                server                                                    */
             moose_server[formhan.prodid(f_id)].get(name,v)
          fi
       fi
       b := true
       return
    end



/***********************************************************************/
/* this proc implements the read operation                              */

    proc read(name,values) returns b
       var o_id : int

       b := read_work(name,values,o_id)
    end


    /* this proc does all of the work for the read operation.  */

    proc read_work(name,values,o_id) returns b
       var f_id : int := formhan.form_id(name)

       if f_id = 0 ->
          b := false
          return
       [] else ->
          if ub(values) != formhan.form_size(f_id) ->
             b := false
             return
          fi
       fi

       b := true
       if (formhan.lazy_add(f_id,me)) ->
          /* lazy add and any delete mode - we ship the function to
             somebody else to perform                                  */
          var keepgoing : bool := true
          var bestserver : int := 0

          /* determine the best server, if any, to ship the function to.
             An eager add/eager delete server is best of all, followed
             by any eager add server.                                   */
          fa i := 1 to num_servers st keepgoing ->
             if formhan.eager_add(f_id,i) ->
                bestserver := i
                keepgoing := not formhan.eager_del(f_id,i)
```

```
            fi
         af

         if (bestserver != 0) and (formhan.num_producers(f_id) > 1) ->
            /* if possible and > 1 producer, ship the function */
            moose_server[bestserver].read_work(name,values,o_id)
         [] else ->
            /* otherwise, poll the producers for the answer */
            start_owner_read(name,values,o_id)
         fi

      [] (not formhan.lazy_add(f_id,me)
                  and formhan.eager_del(f_id,me)) ->
         /* eager delete and eager or cached add mode - we simply call
            the checkwait function and wait for the object to show up
            in the local database                                      */
         b := dbhan.checkwait(name,values,o_id)

      [] else ->
         /* lazy or cached delete mode and eager or cached add mode -
            when the object gets produced it will be sent here, but
            the deletion of the object might be delayed, so we have
            to check with the owner after we find an object to make
            sure that it hasn't been deleted yet.                   */
         var v[1:ub(values)] : int

         do (true) ->
            v := values
            /* find an object */
            if not dbhan.checkwait(name,v,o_id) ->
               b := false
               return
            fi
            /* check with the owner to see if it's still present */
            if (moose_server[owner(o_id)].still_there(name,v,o_id)) ->
               /* if it's still present, return it */
               b := true
               values := v
               return
            [] else ->
               /* if it's been deleted from the object space, delete
                  it locally and try again                          */
               dbhan.local_delete(name,v,o_id)
            fi
         od
      fi
   end


/************************************************************************/
/* this proc implements the put operation                               */

   proc put(name,values) returns b
      var f_id : int

      /* simply check to make sure it's a valid object, then call
```

```
      the global_add procedure                                    */
   f_id := formhan.form_id(name)
   if (f_id = 0) ->
      b := false
      return
   [] else ->
      if (ub(values) != formhan.form_size(f_id)) ->
         b := false
         return
      [] else ->
         b := true
      fi
   fi
   global_add(f_id,name,values)
end


/* send the addition immediately to all eager-add servers, and place
   it in the cache of all cached-add servers.  Ignore any servers
   which are lazy-add.  Also, ignore any servers which are not
   consumers, even if they are eager- or cached-add.             */

proc global_add(f_id,name,values)
   var o_id : int

   o_id := new_obj_id()
   dbhan.local_add(name,values,o_id)
   fa i:=1 to num_servers st (i!=me and formhan.consumer(f_id,i)
                              and formhan.eager_add(f_id,i)) ->
      send moose_server[i].local_add(name,values,o_id)
   af
   fa i:=1 to num_servers st (i!=me and formhan.consumer(f_id,i)
                              and formhan.cached_add(f_id,i)) ->
      send addcachehan[f_id]^.server[i].cache_add(name,values,o_id)
   af
end




/* send the deletion immediately to all eager-del servers, and place
   it in the cache of all cached-del servers.  Ignore any servers
   which are lazy-del.  Also, ignore any servers which are not
   consumers, even if they are eager- or cached-del.             */

proc global_delete(f_id,o_id,name,values)
   dbhan.local_delete(name,values,o_id)
   fa i:=1 to num_servers st (i!=me and formhan.consumer(f_id,i)
                              and formhan.eager_del(f_id,i)) ->
      send moose_server[i].local_delete(name,values,o_id)
   af
   fa i:=1 to num_servers st (i!=me and formhan.consumer(f_id,i)
                              and formhan.cached_del(f_id,i)) ->
      send delcachehan[f_id]^.server[i].cache_del(name,values,o_id)
   af
end
```

```
/* this proc deletes an object which was produced locally if it
   hasn't already been deleted.  Return true if the deletion was
   successful, false if the object has already been deleted.
   A semaphore for each object form is used so that two owner_delete
   operation trying to delete two objects of the same form will not
   interfere                                                          */

proc owner_delete(o_id,name,values) returns ok
   var f_id : int := formhan.form_id(name)

   receive od_sem[f_id]()
   if dbhan.unique_check(f_id,values,o_id) ->
      ok:= true
      reply
      global_delete(f_id,o_id,name,values)
      send od_sem[f_id]()
   [] else ->
      ok := false
      send od_sem[f_id]()
   fi
end



/*********************************************************************/
/* the following procs are used for the owner_read operation.  This
   operation is used to forward a read operation from a lazy-add server
   onto all of the servers which produce the objects of that form.
   The operation is complicated by the need to cancel the processes
   operating on other servers when one process returns successfully */

   var or_ctr : int := 1

   /* start up the owner_read process on all producer servers, and wait
      for one to respond with success.  When one returns successful,
      send cancellation messages to all of the others.               */

   proc start_owner_read(name,values,o_id)
      var f_id : int := formhan.form_id(name)
      var f_index : int
      var can_ids[1:num_servers] : int
      var my_ctr : int

      my_ctr := or_ctr
      or_ctr++

      co (i:=1 to num_servers st formhan.producer(f_id,i))
         can_ids[i] := moose_server[i].owner_read(name,values,
                                              myresource(),my_ctr)
      oc

      in return_owner_read(rn,v,o,f) & rn=my_ctr ->
            values := v
            o_id := o
```

```
            f_index := f
      ni

      fa i:=1 to num_servers st formhan.producer(f_id,i) ->
         if (i != f_index) ->
               send moose_server[i].cancel_owner_read(can_ids[i])
         fi
      af
   end


   /* this process is used to forward owner_read and cancel_ocw
      invocations on to the dbhandler resource where they will be
      processed.  The owner_read operation in the dbhandler (started
      with an invocation of invoke_ocw) will reply directly to the
      start_owner_read process above.                              */

   process orp
      do (true) ->
         in owner_read(name,values,rep_cap,rep_num) returns can_id ->
            can_id := dbhan.invoke_ocw(name,values,rep_cap,rep_num)
         [] cancel_owner_read(can_id) ->
            send dbhan.cancel_ocw(can_id)
         ni
      od
   end


end
```

Appendix C: Steiner Tree Example Source Code

```
resource stein_master

   /* this resource is the master resource in the computation.  Only one
      such resource will exist.  It is responsible for putting the data
      and tokens into the object space, and for collecting the results
      once they have been computed by the workers.  This resource also
      times the whole process.                                         */

   external mscounter() returns t:int

   import types
   import moose

   op go()

body stein_master(num_tokens,num_nodes,num_critical:int; ms:cap moose)


   proc go()
      var t[1:num_nodes+3] : int
      var min,id : int
      var max_range, range_size : int
      var in_graph[1:num_nodes,1:num_nodes] : int
      var stime,etime : int

      /* read the input graph from stdin */
      fa i := 1 to num_nodes, j := 1 to num_nodes ->
         read(in_graph[i,j])
      af

      write("graph size :",num_nodes,"nodes     tokens :",num_tokens)
      write("Starting timing...")
      stime := mscounter()

      /* PUT the data */
      fa i := 1 to num_nodes ->
         t[1] := i
         t[2:num_nodes+1] := in_graph[i,1:num_nodes]
         ms.put("DATA",t[1:num_nodes+1])
      af

      /* PUT the tokens - each token is an integer range */
      max_range := (1 << (num_nodes - num_critical)) - 1
      range_size := (max_range+1)/num_tokens
      fa i := 0 to num_tokens - 1 ->
         t[1] := i
         t[2] := range_size * i
         t[3] := range_size * (i+1) - 1
         ms.put("TOKEN",t[1:3])
      af
      if (range_size*num_tokens != max_range + 1) ->
         t[1] := num_tokens
         t[2] := range_size * num_tokens
         t[3] := max_range
```

```
            ms.put("TOKEN",t[1:3])
            num_tokens++
         fi

         /* collect the results */
         min := 10000
         fa i := 0 to num_tokens-1 ->
            t[1:2] := (i, types.undefined)
            ms.read("RESULT",t[1:2])
            if (t[2] < min) ->
               min := t[2]
               id := t[1]
            fi
         af

         etime := mscounter()
         write("Computation completed...")
         write("\n\nTotal time for computation :",etime-stime)

      end

   end
```

```
resource stein_worker

   /* this resource is the worker resource.  An arbitrary number of
      resources will exist (1 or more).  These resource compete over
      the work tokens, performing the work specified when they
      successfully grab a token.                                   */

   import types
   import moose

body stein_worker(num_nodes:int; moose_res:cap moose)

   const infinity := 10000

   type graph_type = rec(
         adj_mat[1:num_nodes,1:num_nodes] : int
         present[1:num_nodes] : bool)

   op modify(ing:graph_type; n:int; res out:graph_type)
   op min_span(g:graph_type; res t:graph_type) returns w:int


   /* given the input graph, modify it according to the enumerating
      integer.  Each bit in the integer corresponds with a node in the
      input graph, with a bit value of 1 indicating that the node should
      be removed for this particular subgraph.                          */

   proc modify(in_graph,n,out_graph)
      out_graph := in_graph
      fa i := 1 to num_nodes ->         /* for each bit in n ... */
         if (n%2 = 1) ->                /*     if the bit is set... */
            out_graph.present[i] := false  /*     remove the node... */
            fa j := 1 to num_nodes ->
               out_graph.adj_mat[i,j] := infinity
               out_graph.adj_mat[j,i] := infinity
            af
         fi
         n := n >> 1
      af
   end



   /* given an input graph, apply Prim's algorithm to compute the
      minimum weight spanning tree.  Return the tree and its weight */

   proc min_span(g,t) returns weight
      var iterations, min_weight, min_from, min_to : int

      /* initialize everything...  choose a node as the starting node
         for the tree.                                                */
      t.adj_mat := ([num_nodes*num_nodes] infinity)
      t.present := ([num_nodes] false)
      iterations := 0
      fa i := 1 to num_nodes st g.present[i] ->
         iterations ++
      af
      iterations --
```

```
    weight := 0
    fa i := 1 to num_nodes st g.present[i] ->
       t.present[i] := true
       exit
    af

    /* for each iteration, add the least weight arc from the current
       tree to a node not in the current tree - this is the greedy
       approach.                                                    */
    fa q := 1 to iterations ->
       min_weight := infinity
       fa i := 1 to num_nodes st t.present[i],
                j := 1 to num_nodes st (not t.present[j]) ->
          if (g.adj_mat[i,j] < min_weight) ->
             min_weight := g.adj_mat[i,j]
             min_from := i
             min_to := j
          fi
       af
       if (min_weight = infinity) ->
          weight := infinity
          return
       [] else ->
          weight := weight + min_weight
          t.adj_mat[min_to,min_from] := min_weight
          t.adj_mat[min_from,min_to] := min_weight
          t.present[min_to] := true
       fi
    af
end



/* this is the main process executed by a worker process.  It GETs
   a work token, uses the previous two procs to perform the work
   specified by that token, and then goes back for another.     */

var input_graph, work_graph, tree, min_tree : graph_type
var start,finish,min,weight : int
var t[1:num_nodes+3] : int
var id : int

process main
   /* collect the data (the input graph) */
   fa i := 1 to num_nodes ->
      t[1] := i
      t[2:num_nodes+1] := ([num_nodes] types.undefined)
      moose_res.read("DATA",t[1:num_nodes+1])
      input_graph.adj_mat[i,1:num_nodes] := t[2:num_nodes+1]
   af
   input_graph.present := ([num_nodes] true)

   do (true) ->
      /* grab a token... */
      t[1:3] := ([3] types.undefined)
      moose_res.get("TOKEN",t[1:3])
```

```
          id := t[1]
          start := t[2]
          finish := t[3]
          min := infinity
          /* for each subgraph specified by the token, apply Prim's
             algorithm.  Take note of the least weight.              */
          fa i := start to finish ->
             modify(input_graph,i,work_graph)
             weight := min_span(work_graph,tree)
             if (weight < min) ->
                min_tree := tree
                min := weight
             fi
          af
          /* PUT the result (i.e. the least weight) */
          t[1:2] := (id,min)
          moose_res.put("RESULT",t[1:2])
       od
    end

end
```

```
resource steiner

   /* this is the main resource of the steiner program.  This resource
      is responsible for creating the virtual machines, creating and
      initializing the moose servers, defining the object forms,
      creating the worker and master resources, and then starting things
      moving                                                            */

   import types
   import moose
   import stein_worker
   import stein_master

   external gethostname(res n : string(*); i:int)

body steiner()

   const max_nodes := 10

   var vms[1:max_nodes-1] : cap vm
   var ms[1:max_nodes] : cap moose

   var num_nodes : int
   var num_tokens : int

   initial
      var n : string(50)

      /* the parameters to this program should consist of an integer
         followed by a list of physical machine names.  The integer
         specifies the number of work tokens that the task will be
         split into.  The list of machines specifies the machines
         which the worker processes will be created on.  The master
         resource is created on the machine from which the program
         is executed.                                                  */
      if (numargs() < 2) ->
         write("form: steiner <num tokens> <worker1> [<worker2> ",
                     "[<worker3>...]]")
         stop
      fi
      getarg(1,num_tokens)
      num_nodes := numargs()
      writes("\n\nRunning Steiner Tree problem on ",num_nodes,
                " servers...\n\n")

      gethostname(n,40)
      write("   locating master node on",n)
      ms[1] := create moose()
      fa i := 1 to numargs()-1 ->
         getarg(i+1,n)
         write("   locating worker",i,"on",n)
         locate(i,n)
         vms[i] := create vm() on i
         ms[i+1] := create moose() on vms[i]
      af

      /* initialize moose servers */
      write("Initializing moose servers...")
```

```
      fa i:=1 to num_nodes ->
         ms[i].init_moose(ms[1:num_nodes],i)
      af

   end


   process main
      var mres : cap stein_master
      var wres : cap stein_worker
      var graph_size,num_critical : int

      read(graph_size,num_critical)

      var t[1:num_nodes] : bool := ([num_nodes] true)
      var tf[1:num_nodes] : bool := (true, [num_nodes-1] false)
      var ft[1:num_nodes] : bool := (false, [num_nodes-1] true)
      var one[1:num_nodes] : int := ([num_nodes] 1)
      var of[1:num_nodes] : int := (1,[num_nodes-1] infinity)

      ms[1].def_form("DATA",graph_size+1,types.safe,tf,t,one,one,
                                                            one,one)
      ms[1].def_form("TOKEN",3,types.safe,tf,t,of,one,of,one)
      ms[1].def_form("RESULT",2,types.safe,ft,tf,of,one,of,one)

      mres := create stein_master(num_tokens,graph_size,num_critical,
                                                            ms[1])
      fa i := 1 to num_nodes - 1 ->
         wres := create stein_worker(graph_size,ms[i+1]) on vms[i]
      af
      mres.go()

      stop
   end

end
```

References

Selim G. Akl, *Parallel Sorting Algorithms*, Academic Press Inc., Orlando Florida, 1985.

Gregory R. Andrews et al., "An Overview of the SR Language and Implementation" in *ACM Transactions on Programming Languages and Systems*, **10** (1), January 1988.

Gregory R. Andrews and Ronald A. Olsson, "Revised Report on the SR Programming Language," University of Arizona TR 87-27, 1987.

M. Stella Atkins, "Experiments in SR with different Upcall Program Structures" in *ACM Transactions on Computer Systems*, November 1988.

Philip A. Bernstein, Nathan Goodman and Ming-Yee Lai, "Analyzing Concurrency Control Algorithms When User and System Operations Differ" in *IEEE Transactions on Software Engineering*, **9** (3), 1983.

Kenneth P. Birman and Thomas A. Joseph, "Reliable Communication in the Presence of Failures" in *ACM Transactions on Computer Systems*, **5** (1), February 1987.

F. W. Burton, "Functional Programming for Concurrent and Distributed Computing" in *The Computer Journal*, **30** (5), 1987.

Nicholas Carriero and David Gelernter, "The S/Net's Linda Kernel" in *Proceedings of the Symposium on Operating System Principles*, December 1985.

Nicholas Carriero, David Gelernter and Jerry Leichter, "Distributed Data Structures in Linda" in *Proceedings of the Principles of Programming Languages Symposium*, 1986.

David R. Cheriton, "Preliminary Thoughts on Problem-oriented Shared Memory: A Decentralized Approach to Distributed Systems" in *ACM Operating Systems Review*, **19** (4), October 1985.

David R. Cheriton, "Problem-oriented Shared Memory: A Decentralized Approach to Distributed System Design" in *6th International Conference on Distributed Computer Systems*, May 1986.

David R. Cheriton and Michael Stumm, "The Multi-Satellite Star: Structuring Parallel Computations for a Workstation Cluster," to appear in *Distributed Computing*, 1988.

G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker, *Solving Problems on Concurrent Processors*, Prentice-Hall Inc., Englewood Cliffs New Jersey, 1988.

Armen Gabrielian and Douglas B. Tyler, "Optimal Object Allocation in Distributed Computer Systems" in *4th International Conference on Distributed Computer Systems*, May 1984

Maurice Herlihy, "Optimistic Concurrency Control for Abstract Data Types" in *Proceedings of the Principles of Distributed Computing Conference*, 1986.

Ellis Horowitz and Sartaj Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press Inc., Rockville Maryland, 1978.

Sigurd L. Lillevik and John L. Easterday, "Throughput of multiprocessors with replicated shared memories" in *AFIPS Conference Proceedings*, 1984.

Bruce G. Lindsay, Laura M. Haas, C. Mohan, Paul F. Wilms, and Robert A. Yost, "Computation and Communication in R*: A Distributed Database Manager" in *ACM Transactions on Computer Systems*, **2** (1), February 1984.

Ronald A. Olsson, "Issues in Distributed Programming Languages: the Evolution of SR," University of Arizona TR 86-21 (Ph.D. Dissertation), 1987.

K. Ravindran, "Reliable Client-Server Communication in Distributed Programs," Ph.D. Dissertation, University of British Columbia, 1987.

Peter M. Schwarz and Alfred Z. Spector, "Synchronizing Shared Abstract Types" in *ACM Transactions on Computer Systems*, **2** (3), August 1984.

Michael L. Scott, "Language Support for Loosely Coupled Distributed Programs" in *IEEE Transactions on Software Engineering*, **13** (1), January 1987.

Douglas B. Terry, "Caching Hints in Distributed Systems" in *IEEE Transactions on Software Engineering*, **13** (1), January 1987.

Arthur H. Veen, "Dataflow Machine Architecture" in *ACM Computing Surveys*, **18** (4), December 1986.

Scott J. Warren and Joan M. Franscioni, "Reduction of Communication Delays in Hypercube Programs Based on Run Time Statistics" in *6th International Conference on Distributed Computing Systems*, June 1988

Pawel Winter, "Steiner Problem in Networks: A Survey" in *Networks*, **17**, 1987.