T-4255

INVESTIGATION OF PARALLEL PROGRAMMING METHODS

by

John D. Crabtree

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

T-4255

A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Master of Science (Mathematics).

Golden, Colorado
Date 5/18/92

Signed: _____
John D. Crabtree

Approved: _____
Dr. Aaron Gordon
Thesis Advisor

Golden, Colorado
Date 18 May 1992

Approved: _____
Dr. Ardel J. Boes
Professor and Head,
Department of Mathematics
and Computer Science

ABSTRACT

This thesis compares the attributes of the client/server and Godzilla paradigms used within several distributed processing environments and further compares these paradigms to Linda. The application is a seismic migration program which graphically displays the migrated seismic section through the use of Xmovie.

Using the client/server model, the client hands out frequency vectors to each server which extrapolates an output array that is returned to the client for display.

The Godzilla paradigm uses a slightly different approach. The client starts the servers as in a normal client/server fashion, but then each server is free to decide what resources or actions it may request from the client. This process has the effect of freeing the manager to satisfy more server requests.

Linda was the third paradigm used. The manager releases the data into an area of memory called the tuple space. Each worker is then able to select data tuples from this area.

The parallel processing environments investigated were PVM public-domain software developed at Oak Ridge National Laboratory, NCS (Network Computing System) under licence to IBM from Hewlett-Packard Inc., IMCS (Inter Machine Communication Services) from IBM, and C-Linda from Scientific Computing Associates Inc.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I wish to thank my advisor, Dr. Aaron Gordon, who was of great assistance when circumstances necessitated a change in focus of this thesis. I also wish to thank the Center for Geoscience Computing at CSM and Gina Boice in particular for her invaluable technical assistance. I would also like to thank IBM for funding this project.

Thanks also goes to my wife Lynne for her understanding and ability to cope with my late night sessions at the Computer Center. Finally, I would also recognize my parents and my sister for their support throughout my academic career.

Chapter 1

INTRODUCTION

For years scientists have known that some problems, due to their size and complexity, cannot be solved even on the fastest supercomputers. Others can only be solved using a supercomputer. Another larger class of problems exists which can be solved with lower performance hardware, but only at high cost. Resources, such as computers and employees, are often tied up waiting for output from these computationally-intensive programs, sometimes to the extent that solving the problem is no longer economical for the company.

These are some of the roadblocks that distributed processing hopes to clear in the future. Recently, the Lawrence Livermore National Laboratory in California replaced a Cray X/MP with 14 IBM RS/6000 workstations. They found that the combined power of the networked workstations was equal to that of the Cray at 1/20th of the cost. [1]

This investigation will use a network of loosely coupled processors (more specifically, IBM RS/6000 workstations) to investigate the advantages and disadvantages of the use of different parallel processing paradigms (client/server, Godzilla, and Linda) for the distributed implementation of a seismic migration application. The application will be used to compare the relative run-times using each of the parallel processing environments (PVM, NCS, libimcs, Linda). The environments will also be contrasted in terms of functionality, ease of use, and ideas for future enhancements. I use the term "environment" here to describe a set of library routines and/or system calls, delivered as a distinct software package, which enables the programmer to parallelize programs. Table 1.1 shows how the environments relate to particular paradigms and interfaces (physical networks) for this investigation.

Table 1.1 Environment and Paradigm Relationships

Environment	Paradigms	Interfaces
PVM	Client/Server, Godzilla	Token Ring
NCS	Client/Server, Godzilla	Token Ring
libimcs	Client/Server, Godzilla	Fiber Optics
Linda	Linda	Token Ring

A seismic record, or seismic section, is a graphical representation of the results of a seismic survey. The section is created by using a string of geophones, usually in a straight line, to record the vibrations created by a wavefront produced by a seismic source (e.g. dynamite). A reflection of the wavefront is recorded by the geophones whenever a discontinuity in the subsurface (e.g., a change in rock type and thus velocity) is encountered.

If the layering in the subsurface is flat (has zero dip), the record produced by a single geophone (single trace) can be correlated to a single point in the subsurface. That point

lies midway between the source and receiver and the depth can be calculated if the velocity of the medium is known. (The velocity is usually determined by a different type of geophysical survey or by the use of core samples.) The occurrence of such flat layering of beds is rare.

A reflection from a dipping geologic bed can come from any one of a locus of points that form an ellipse, the foci being the source (the dynamite) and the receiver (the geophone).

With one trace, there is no way to deduce where the actual source of the reflection lies in space; however, a geophysicist can produce a self-consistent seismic record by using velocity information and the change in reflection time between adjacent traces (apparent dip) to migrate the source of the reflection to its true position. This process is called seismic migration.[2]

In the geophysical industry, seismic processing has long been performed on mainframes at a high cost. Due to this cost, many small oil companies have chosen to contract out

their processing to geophysical companies or larger oil companies. One of the most important and time-consuming procedures is the migration of seismic data.

During processing, the geophysicist may migrate the section several times until the desired result is achieved. In most processing centers, the geophysicist must run the migration, which may take many hours depending on the amount of data, and then send the output to be plotted. Once the plot is received, it may take only minutes to decide to make some slight changes and remigrate the data.

Obviously, faster migration techniques are desirable as is graphical output both of which save time and money.

The migration application presented distributes the processing over many workstations and sends graphical output to the workstation terminal using Xmovie. Besides the advantages of graphical output and less execution time the ability to monitor intermediate output is also advantageous. Output is reported back to the managing process as often as

needed through the use of a run-time frequency parameter and can be displayed immediately. The user can start a migration, notice problems with the output, and stop the processing even though the computation is not complete.

Chapter 2

THE SEISMIC MIGRATION APPLICATION

A seismic section is a graphical representation of the subsurface produced by the recording of seismic waves (Fig. 2.1 (a)). These sections are usually displayed as distance (along the surface) vs. time (in the y direction, the time required for the signal to reach the surface), although they can be shifted to distance vs. depth. Historically, seismic sections have been displayed on large paper records.

Unmigrated seismic sections are useful only when there are no discontinuities or steeply dipping surfaces in the seismic section. When geologic features such as faults, steeply dipping beds, anticlines, and synclines are encountered, reflections are deflected producing distortion that makes interpretation difficult, if not impossible. [3][4]

There are several methods of seismic migration. Due to the increased use of computers, the two most popular are the finite difference method and f-k migration.

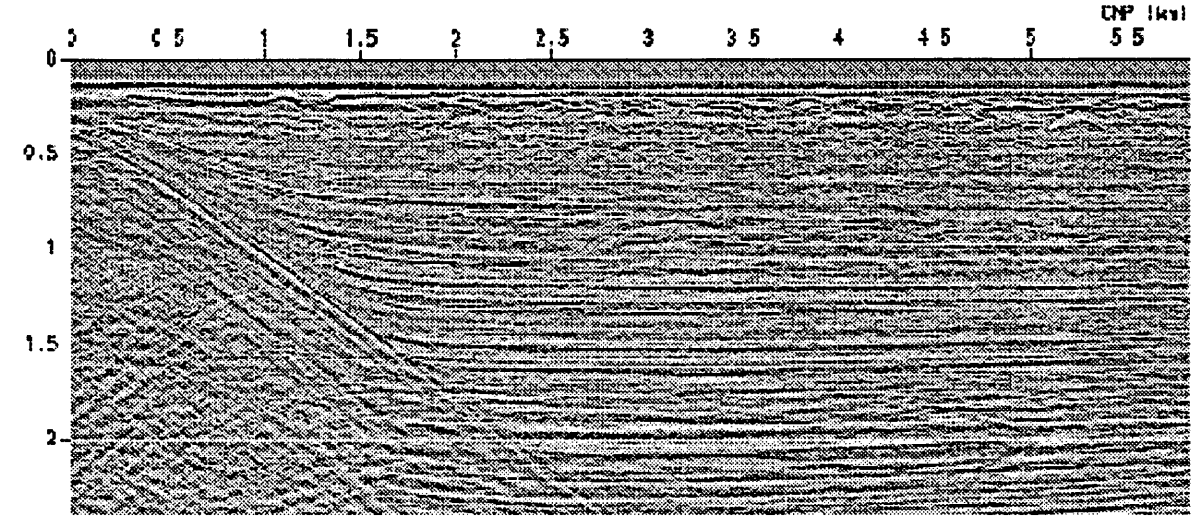


Figure 2.1 (a) Unmigrated Seismic Section

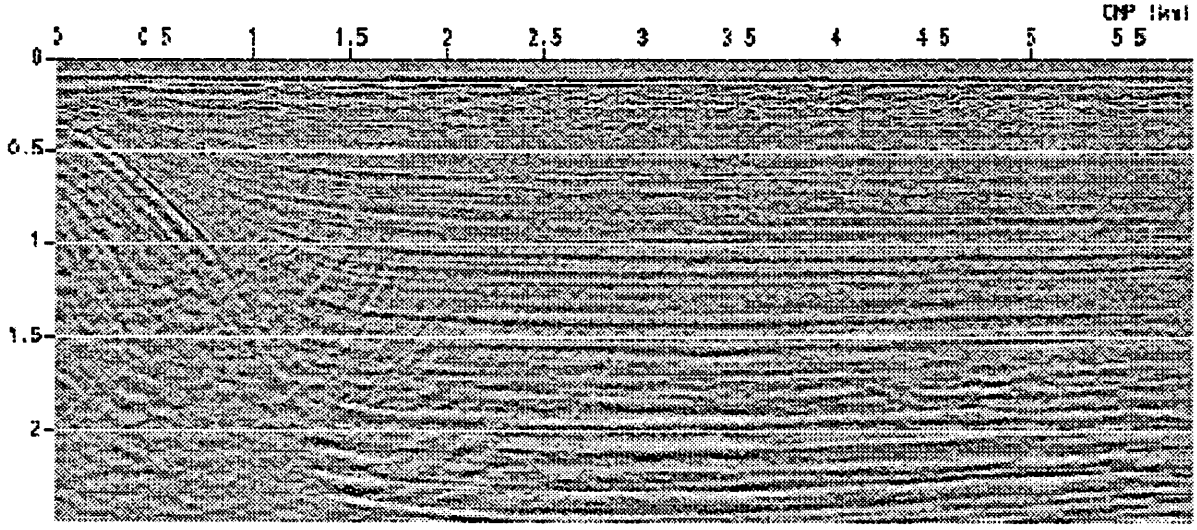


Figure 2.1 (b) Migrated Seismic Section

The finite difference method migrates the observed wave field at the surface downward by recursive solutions of the wave equation.

This application uses f-k migration techniques to migrate the section. The solution of the wave equation is extrapolated downward by using the Fourier transform of the wave equation.

The advantage of the f-k migration method is that the output samples can be computed independently of other results, whereas solution by finite differences requires the solution of simultaneous equations. The f-k migration method, therefore, is much easier to parallelize since each computation is independent of the others.

The serial application was written by Dave Hale and uses a modified Taylor series to perform the depth extrapolation of the seismic section.[5]

The application begins by reading the input data and parameters and performing a Fast Fourier Transform on the seismic section (p), changing time and distance to frequency and wavenumber within the transformed section (cp). The program then begins a nested loop which first loops over each frequency and then loops over each depth point where a call to the procedure etextrap is made. The call to etextrap consumes the majority of the computation time and itself loops over all samples in the wave field (nx).

The call to etextrap uses the input extrapolation table (et), the velocity profile information (wdxov), and the transformed wave field for one particular frequency (cpx) to produce the extrapolated wave field for that frequency.

The migrated wave field (cpx) is then accumulated with output from other depths and frequencies to produce the migrated section (q).

The resulting section is output through xmovie which graphically displays the section. Intermediate results can

also be sent to xmovie. An input parameter (mw) controls the frequency at which the xmovie display is updated.

Chapter 3

PARADIGMS

Each of the parallel implementations of this application, besides Linda, use remote procedure calls (rpc) and message-passing as the mode of communication. The processing elements communicate by passing data in the form of messages.

The traditional paradigm used with message-passing is the client/server model. The client calls on multiple servers to process data elements and return the results to the client.

One paradigm used by this application however, uses a slightly different approach. The client starts the servers as in a normal client/server model, but then each server is free to decide what resources or actions it may request from the client. Due to these differences, the client is often referred to as the manager while the servers are called workers. This is the Godzilla paradigm.[6]

Using the traditional approach, this application could have been designed so that each server would be given a wave field for a particular frequency, and the client would wait until the results were returned. At this point the client could sum the results and send another data element to the server.

The Godzilla paradigm allows each worker to request data independent of the manager. Each worker requests data from the manager, processes the data, and keeps a sum of the wave fields it has calculated. When the display frequency parameter (mw) dictates that a report of results is needed, the worker signals the manager that it is going to report, sends the data, and clears its internal sum. Thus the summation of the wave fields is left to the workers which frees the manager to satisfy more requests for data.

This process of freeing the manager to satisfy more worker requests is referred to as load balancing. Since the manager spends less time accumulating results, more time can be spent keeping the workers busy.

The benefit of proper load balancing becomes apparent when one or more of the machines being used in parallel are also being used for other processes. The client/server paradigm cannot adjust the data distribution to account for this situation.

Linda is a paradigm that is markedly different from either the client/server model or Godzilla.

Linda is a complete distributed system that uses its own communication methods explicitly designed for parallel processing. The version of Linda used for this application is C-Linda from Scientific Computing Associates Inc. [7]

The concepts of Linda were created by David Gelernter as a method for distributed processing.[8] The basis of the language is the concept of a "shared memory space" called a "tuple space". Linda implements and manages this space thus providing a way to "share" memory on distributed systems.

The basic Linda language contains six operations that are added to a sequential language. Linda data objects are described as being passive or active. Passive objects, ordinary data objects, are added to the tuple space (TS) with the "out" procedure. The data is retrieved with "in", which waits for the data if necessary, "rd" which also waits but does not remove the object from the TS, "inp" which does not wait, and "rdp" which neither waits nor removes the data object.

Active data objects, executable procedures, are placed into TS with "eval". These objects are retrieved by the first available processing elements, determined by Linda, and defined in a "hosts" file which contains a list of node names.

This application was implemented in Linda by first "outing" the initialization data into TS. Then each frequency data object was "outed" and the workers were started by n calls to "eval", where n is the number of remote machines defined in the "hosts" file.

The workers first "rd" the initialization data, leaving the object in the TS for the other workers, and then proceed to "inp" the frequency data object and process the array. When no more frequency data objects are available, determined by the status of the "no wait" "inp", each worker deposits a "finished" tuple into TS with an "out", thus signaling the manager that the worker is done.

The display frequency parameter (mw) is also used with Linda. When the value of mw is equal to the number of frequencies accumulated, a tuple is "outed" for the manager to find.

Meanwhile, after the manager starts each worker with a call to "eval", another process is created to handle the processing of the results from the workers (a report manager). Once found, these result tuples are displayed with xmovie.

If a worker is not able to find a frequency data object in the TS, it "outs" a message to the manager that it is done. Once all the workers have reported to the manager in this

fashion, the manager sends a dummy report tuple to the report manager. This report tuple contains a flag that signals the report manager to finish. Once all child processes created by the manager have finished, it is safe to end the program.

Chapter 4

ENVIRONMENTS

The machines used for these experiments are a network of IBM RISC System/6000 Model 530's. Since this hardware is fairly new, the number of distributed programming environments is limited. At present no concurrent programming languages are available. Thus, we are limited to environments designed for loosely coupled processing elements such as NCS and PVM.

These environments are based upon remote procedure calls and message-passing. Linda, which offers some of the advantages of shared memory is also available.

4.1 NCS

The Network Computing System (NCS) provides an environment on the RS/6000's for distributed processing. The system is made up of the network computing kernel (NCK) and the network interface definition language (NIDL).

The NCK provides an rpc library and a location broker. The rpc library contains system calls which are used to make procedure calls to routines on remote machines. The location broker serves as a remote node manager used by the managing process (or client) to make remote procedure calls.

These routines were designed with the client/server paradigm in mind. The client makes remote procedure calls to a number of servers, often passing data to the servers, which process the data and return results.

The NIDL is used to define relationships between the client and server. This code is then compiled by the NIDL compiler to produce stub code. The stub code contains the rpc code necessary for the communication between the client and server. This communication code is often difficult to write. The NIDL compiler saves the programmer time and effort by producing the stubs. The programmer then modifies the stub code to include the application code.

4.2 PVM (Parallel Virtual Machine)

PVM is public-domain software developed at Oak Ridge National Laboratory to support parallel computing on loosely coupled networks of processing elements. The programmer uses the library routines provided by PVM directly.

The environment provides a daemon called pvmd which is executed on the host machine. The daemon reads a host file which contains the node names of all other processing elements to be included in processing. The host daemon then starts the daemons on the other nodes and coordinates the communication between processes.

The user may then start the application as usual. The placement of executables in directories known to pvmd is used to identify images of be executed remotely. When the main executable is run on the host machine, the "initiate" routine is used to start a given image on a remote machine.

The "snd" and "rcv" routines are used to pass messages between processes. A message buffer area is maintained by pvmd for process communication. The "initsend" routine is used to clear the buffer and the "get(var type)" and "put(var type)" routines are used to retrieve from and insert into the message buffer. The "terminate" routine is used to kill the remote images and the "leave" routine signals pvmd that the application is finished.

This environment proved to be very flexible and easy to use. Using the Godzilla paradigm, the workers were started on the remote machines using "initiate" and were then responsible for sending short messages to the manager identifying what type of service was required. If initialization or the next frequency vector was requested, the manager would fill the buffer with the required data and issue a "snd". The worker would then retrieve the data with "rcv" and "get(var type)". If a report was to be made from the worker to the manager, the worker would signal the manager that it was ready to report, fill the buffer, and issue a

"snd". The manager would then "get" the data and add it to previous results.

4.3 libimcs

IMCS (Inter-Machine Communication Services) is the interface between the operating system of the RS/6000 workstation and the Serial Link Adapter (SLA) which allows external devices to be connected to the workstation by means of fiber-optic connections. IMCS is a device driver which controls the SLA hardware. IMCS is a kernel extension as is if_imcs which is the interface layer between TCP/IP and IMCS. The International Standards Organization created the open systems interconnection model (OSI) for computer communication. While most systems in use today were built before the OSI model was created, they are usually described in terms of the model.[9]

In OSI nomenclature, if_imcs is part of the "data link" layer as is IMCS. The following table shows the relationship of the two interfaces used in this report.

Table 4.1 The OSI Model

OSI Model	Token Ring	Fiber Optics
Application	migration app	migration app
Presentation	not used	not used
Session	not used	not used
Transport	TCP	TCP
Network	IP	IP
Data Link	IEEE 802.5	if_imcs IMCS
Physical	Token Ring	Fiber Optics

The optical link was designed for external storage devices (e.g. tape drives) but can be used, as it is here, for networking machines. The connections are point-to-point, so the number of workstations that can be connected is limited. With the hardware currently used at CSM, the maximum number of machines on one IMCS network is three. Routing hardware does exist which can increase the number of workstations in the network.

Due to this hardware limitation, the timings used for environment and paradigm comparisons were set to a configuration of three IBM RS/6000 Model 530 workstations.

A library of communication routines, coded by IBM, was used for the purpose of experimenting with fiber-optic communications as a basis for distributed processing. I reorganized this code into the library "libimcs". This library is documented in Appendix A.

The library routines are written directly into the application by the programmer, much like PVM. The programmer is responsible for his own structures that are to be used in communication.

The manager or client program starts the execution of a program on the remote workstations with a call to "start_shells". This library routine takes the name of the file that contains the node names of the remote machines and the path and name of the file to be executed as arguments (see

Appendix A). It also opens a communication channel to each of the remote machines.

The executables running on the remote machines may then make a call to "start_workers" to initialize a reciprocal communication channel back to the manager or client.

The "send_imcs" and "recv_imcs" routines are then used to communicate between processes. An urgent flag can be passed to these routines to indicate whether a higher priority should be used during that communication cycle. This flag is used by the manager to send and receive messages when a worker is either waiting to receive data from the manager or send a report to the manager.

The "tell_imcs" routine is used for communication coordination. The call raises an exception that is handled by a call to "handler", a routine that the user is free to modify. The handler in this application is used to enqueue the id of the process that called "tell_imcs" in a stack. The manager then uses this stack which indicates the machines that

are ready for another processing cycle in the Godzilla paradigm.

This routine, "tell_imcs", is not used in the client/server implementation of the application.

4.4 Linda

In parallel processing, hardware configurations are normally referred to as either shared or distributed memory systems. Shared memory systems are implemented on hardware capable of providing inter-process communication by means of addressable memory that is available to all processes.[10] On these machines, concurrent languages are often used. Concurrent languages often resemble serial languages (such as C or FORTRAN) with various additions to manage the shared memory space.

On the other hand, message-passing is the basis of distributed systems such as the network of workstations used for this research. These machines have no way of sharing

information at the hardware level; therefore, this ability must be implemented in software.

C-Linda is a language of its own, as opposed to a collection of library routines as is PVM and libimcs. The main difference is that the syntax of the Linda-specific calls is checked by the Linda pre-compiler. The data sharing mentioned above, is implemented in software by the C-Linda environment. This data sharing is hidden from the programmer. The user only has access to the six primitives mentioned in Chapter 3, and does not need to concern himself with the complexities of how the simulation of shared memory is implemented.

The communication methods and procedures used by PVM, libimcs, and NCS were not written explicitly for parallel processing but were implemented using existing software, such as TCP/IP, which is primarily used for other communication purposes. The message passing environments use a layer of application software to provide the programmer with access to these communication routines. C-Linda also hides these system

calls from the programmer by providing communication support through the use of its six primitives. While the message passing environments "repackage" the system calls to make their use easier, C-Linda has the effect of truly hiding their use from the user. The programmer is not aware of sending and receiving messages but is only aware of "outing" and "ining" tuples, thus providing the simulation of shared memory.

Chapter 5

TIMING RESULTS

All timing results presented here were performed with the application's reporting frequency set to infinity. In other words, the workers reported the results only once, when there were no more frequency data objects to calculate. This reporting frequency could have been held constant, for comparison purposes, at any value.

The original serial application ran in 177.60 seconds. The megaflop rate was calculated using equation 5.1:

$$\text{MF Rate} = (nx * nz * nw * nops) / \text{real time} \qquad (5.1)$$

where:

nx = the number of seismic traces in the input section

nz = the depth of the output array

nw = the number of frequencies produced by the FFT

nops = the number of floating point operations in the inner loop (the granularity of the distribution)

real time = the elapsed time of execution of the program (or "Real" time) as measured by the unix "time" utility

For the data set used, these values are as follows:

nx = 467

nz = 200

nw = 228

nops = 198

Therefore, the serial version ran at 23.7 megaflops.

The chip used by the RS/6000 Model 530 runs at 25 Mhz, and the hardware is able to process two instructions each cycle. Therefore, the theoretical optimum megaflop rate for the hardware is 50 megaflops. This rate is observed only when the application perfectly matches the architecture of the hardware (e.g. operations always occur in pairs which can be recognized as such and implemented as simultaneous operations by the hardware). In addition, the code must be written so that it can be perfectly optimized by the compiler.

Table 5.1 compares the results of running the application on three IBM RS/6000 Model 530 workstations under each of the environments presented (three were used due to the hardware limitations of libimcs).

Table 5.1 Summary of Results

Environment	Paradigm	Time	MF rate
NCS	Godzilla	82.22	51.28
PVM	Godzilla	84.78	49.73
libimcs	Godzilla	98.47	42.82
NCS	client/server	NA	(see Chapter 6)
PVM	client/server	610.95	6.90
libimcs	client/server	94.12	44.80
Linda	Linda	127.59	34.93

Table 5.2 compares the results of running the application on a number of IBM RS/6000 Model 530 workstations using Godzilla under NCS. I was only able to use up to five 530's

although twelve 530's are available on this network. I did not have execute permission to run the location broker routines and the system administrators felt that they did not yet have enough experience with these routines.

Table 5.2 NCS/Godzilla Results

Num of PE's	Time	MF rate
1	185.48	22.73
2	106.28	39.67
3	82.22	51.28
4	69.95	60.28
5	64.23	65.65

Table 5.3 compares the results of running the application on a number of IBM RS/6000 Model 530 workstations using Godzilla under PVM.

Table 5.3 PVM/Godzilla Results

Num of PE's	Time	MF rate
1	214.55	19.65
2	117.09	36.01
3	84.78	49.73
4	70.07	60.17
5	62.80	67.14
6	58.59	71.97
7	56.15	75.09
8	54.68	77.11
9	55.07	76.57
10	56.06	75.21

Table 5.4 compares the results of running the application on a number of IBM RS/6000 Model 530 workstations using Godzilla under libimcs. (The number of machines is limited to 3 due to hardware constraints).

Table 5.4 libimcs/Godzilla Results

Num of PE's	Time	MF rate
1	384.30	10.97
2	152.28	27.69
3	98.47	42.82

Table 5.5 compares the results of running the application on a number of IBM RS/6000 Model 530 workstations using client/server under libimcs.

Table 5.5 libimcs Client/Server Results

Num of PE's	Time	MF rate
1	199.88	21.09
2	119.80	35.20
3	94.12	44.80

Table 5.6 compares the results of running the application on a number of IBM RS/6000 Model 530 workstations using Linda.

Table 5.6 Linda Results (Version 1)

Num of PE's	Time	MF rate
1	NA	(see Chapter 6)
2	NA	(see Chapter 6)
3	222.61	18.94
4	121.90	34.59
5	88.80	47.48
6	72.21	58.39
7	63.41	66.49
8	58.09	72.58
9	54.91	76.79
10	53.16	79.32

The first version of the Linda program was not able to support as many workers as processing elements used; therefore, a second version was developed (see Appendix F). This will be explained further in Chapter 6.

Table 5.7 compares the results of running the application on a number of IBM RS/6000 Model 530 workstations using the second version of Linda.

Table 5.7 Linda Results (Version 2)

Num of PE's	Time	MF rate
1	NA	(see Chapter 6)
2	221.90	19.00
3	120.72	34.93
4	85.30	49.43
5	70.53	59.78
6	61.51	68.55
7	56.34	74.84
8	50.94	82.77
9	49.69	84.86
10	48.52	86.90

Chapter 6

CONCLUSIONS

6.1 Environments

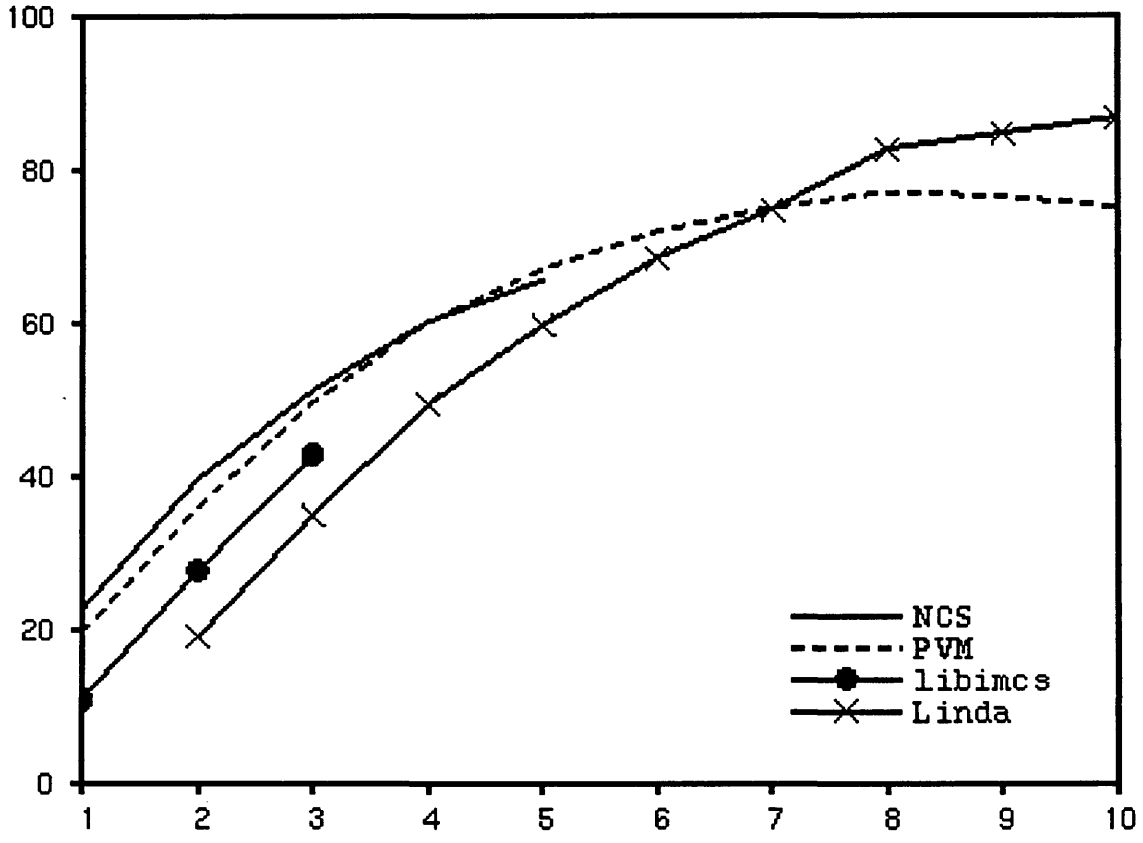
The common baseline used to compare the different versions of the application was the use of 3 RISC/6000 Model 530 workstations. As can be seen in Figure 6.1, the fastest runtime was turned in by the NCS version followed by PVM, libimcs, and finally Linda.

The three machines used are connected by a token ring network as well as the fiber-optic network. One of the most important differences between the three message-passing environments (NCS, PVM, and libimcs) is the choice of networks.

MegaFlop Rate Comparison

Load Balancing (Godzilla/Linda) Used

MegaFlops



Number of Machines

Figure 6.1 MegaFlop Rate Comparison

The OSI model, presented in Chapter 4, is repeated here for reference:

Table 6.1 The OSI Model

OSI Model	Token Ring	Fiber Optics
Application	migration app	migration app
Presentation	not used	not used
Session	not used	not used
Transport	TCP	TCP
Network	IP	IP
Data Link	IEEE 802.5	if_imcs IMCS
Physical	Token Ring	Fiber Optics

All four environments are reported as using TCP as the transport level protocol and IP at the network level by the unix utility "netstat". NCS, PVM, and Linda use the IEEE 802.5 standard at the data link level and the actual token ring networking hardware at the physical layer, while libimcs

uses a combination of the if_imcs and IMCS kernel extensions as the data link level and the fiber optic cabling and SLA (Serial Link Adapter) at the physical layer.

On any network, the data link layer defines the MTU (Maximum Transmission Unit) which is usually set to the maximum message size that the physical layer can handle. The normal MTU for a token ring network is 4,464 bytes.[11] On the network at CSM, the MTU for the token ring is set to 1,500 bytes due to the former use of a communications bridge which required a lower MTU. The MTU set by IMCS, according to the unix utility "netstat", is 28,672 bytes.

The normal method of setting the data link variables for a given Internet network is to define the values in the header file, /usr/include/netinet/if_X.h, where X identifies the communications standard. Therefore, the values for token ring can be found in the file: /usr/include/netinet/if_802_5.h. IMCS, on the other hand, has no such header file. Instead, the data link values must have been set within the executable kernel extensions if_imcs and/or IMCS.

6.1.1 libimcs vs NCS

In order to achieve the maximum possible application speed-up, the use of the CPU on each machine must be maximized. A test was created to compare the relative CPU utilization of the application using Godzilla under NCS and libimcs. The manager only was executed on one 530 and one worker was executed on another 530.

Using the unix utility "vmstat" the remote worker under NCS was observed using approximately 96% of the CPU during runtime, while the worker under libimcs used only about 80% of the CPU. The other 20% was idle.

This suggests that the workers are not being supplied with enough data to keep the CPU busy under libimcs. One of the main reasons for pursuing the use of fiber-optic networks in parallel programming is to minimize the communication time in order to keep remote machines supplied with the necessary data.

In addition, the manager, when using NCS, was found to be using only 30% of the CPU (the other 70% being idle), while the libimcs manager was using virtually all of the CPU. It appears that while the NCS manager is distributing data and waiting for the next request, the libimcs manager is always busy. I believe that this is also the reason that when 2 workstations are used under libimcs, the speedup is greater than twice one machine. Since the manager (which runs on only one machine) is using so much of the CPU, the load balancing of Godzilla causes the other machine to be used much more efficiently.

One hypothesis, presented by George Almasi, was that the TCP/IP protocol suite may be limiting the packet size used by IMCS so that the bandwidth of the fiber-optic connections was not being taken advantage of.[12] The limit imposed by TCP/IP is 65,535 bytes per packet (from /usr/include/netinet/ip.h), while the size of the frequency data objects is only 3,744 bytes and the MTU of the IMCS hardware is reported as being

28,672 bytes. In fact, IMCS was observed using 1/8th the number of packets during a typical run under Godzilla as compared with NCS using token ring. The limiting packet size factor when using token ring is the lower MTU of 1,500 bytes. This means that the token ring network must use 3 packets to send each frequency data object as opposed to only 1 for IMCS. Theoretically, the ratio of NCS to libimcs packets should be closer to 1:5, based on the size of each message transferred by the application. But, the token ring network at CSM is also used for the NFS (Network File System) that supports most files used by the workstations (including this application) to which these extra packets could be attributed. By using the unix utility "netstat", approximately 2 packets per second were observed traversing the token ring network, during low-traffic periods, when my application was not running. Taking this figure into account, the ratio is lowered to 1:6. The most important result is that the NCS version, while sending many more packets, ran faster than the libimcs version.

IMCS does require a relatively large header (128 bytes) as compared to token ring (32 bytes); however, as explained above, this does not result in a larger number of packets produced for the fiber-optic network as compared to token ring.

Another interesting discovery was that 2 pages of memory are added to the process space of each worker executable running under libimcs each time a message is received. This additional memory is allocated by the call to msg_to_imcs which is called by both send_imcs and recv_imcs. The memory is allocated only when recv_imcs calls msg_to_imcs. This problem may be due to the lack of message buffering by IMCS or a failure to free allocated memory within the kernel extension that contains msg_to_imcs. The use of buffers by applications using IMCS is strongly recommended by the IMCS Programmer's Guide, but this requires writing a kernel extension (i.e. a device driver).

Another 2 pages were added to the worker process by the failure to free memory within the recv_imcs routine. This

problem was discovered while coding the client/server version of the libimcs application and will be changed within libimcs as well.

These additions to the process space did not result in paging when the migration program was the only process running on the machine; however, if any other process was using a significant amount of memory, the application would start paging shortly after starting.

A test was run to determine how much time is used for the memory allocation of one page (4096 bytes). It was determined that, on the average, 0.219ms were required; therefore, each recv_imcs accounts for 0.439ms of memory allocation time since each call to recv_imcs causes two separate one page allocations. This problem contributes to the fact that the fiber optic network is slower than expected, but fails to account for even the time difference between NCS and libimcs.

As illustrated by the CPU comparisons above, the use of the fiber-optic network under Godzilla requires more work by

the CPU running the manager process than does the token ring network. An investigation of the libimcs code reveals that no standard network level protocol (e.g. IP) is used by libimcs. Instead, a library routine which polls the socket opened for use by the fiber-optic connection is used by both the send and receive routines. Since the manager waits for the workers to request data, the manager is constantly polling the socket in order to trap the incoming message. Therefore, I became suspicious of the use of TCP and IP as reported by "netstat". I used the library routine "getrusage" which reports statistics concerning resource utilization including the number of IP packets sent and received. This output confirmed the fact that IMCS does not send IP packets, but instead uses signals which are documented as "ru_nsignals" by "getrusage". These signals are described as being similar to software interrupts by the AIX documentation.

The use of these signals by the if_imcs/IMCS kernel extensions in addition to the polling used by the libimcs software seem to be conflicting modes of communication. If signals are the most efficient mode of communication for

parallel programming, it seems that they should be used throughout. If the signaling process does produce system interrupts, as it should, there should be no need for polling. Unfortunately, I have no access to the if_imcs/IMCS source code to further investigate this contradiction. From the results obtained, it seems that the major problem affecting the transmission speed of the fiber-optic network is the choice of communication mode(s) within the kernel extensions and/or libimcs.

IMCS is a technology which is very new and still evolving rapidly. It is, at this time, slower than the other two message passing environments, but has much more potential for speed improvements than the other platforms. Chapter 7 will discuss some possible areas of investigation that could provide more insight into this area.

6.1.2 PVM vs NCS

The small difference in performance between NCS and PVM (1.55 MFlops) is probably due to the extra process, pvmd,

which runs concurrently on all machines used in parallel processing under PVM. This process is a daemon which coordinates the interprocess communication. It is responsible for checking for dead child processes, receiving and sending messages, and adding standard output streams to messages addressed to the master process (manager).

The number of packets sent over the token ring network by PVM compared very closely with that of NCS; however, the use of the pvmd daemon requires additional interprocess communication between pvmd and the worker or manager executable running on the same processor. This communication is carried out by the "sendto" and "recvfrom" socket routines and is not reported in the network packet statistics. Although communication between processes on the same machine, using unix sockets, is very fast, it is a step that is not needed when using NCS. This extra communication could account for the difference in PVM and NCS runtimes.

The PVM/Godzilla timings also showed that the application was slowed when more than 8 processing elements (pe's) were

used. The use of the unix utility "vmstat" showed that the processor on which the manager ran would reach nearly 100% CPU usage when 8 pe's were used. As more pe's were added, the CPU usage of workers dropped as they were not being supplied with data as frequently by the manager.

The subtle differences in timing may also be due to a variety of other factors. PVM is a highly portable platform that is designed to run on a variety of platforms. On the other hand, this version of NCS was designed specifically for the IBM RS/6000. Therefore, NCS may take advantage of certain hardware features that the designers of PVM may not be aware of. For example, the RS/6000 C compiler "xlc" contains an undocumented optimization flag "-qxflag=hsflt" which turns off the checking of floating point numbers that are converted to integer. It is a dangerous option but one that may save many CPU cycles. NCS may use this option while it appears, after checking the PVM makefile, that PVM does not. I noticed a speedup of approximately 5% when this flag is used with the libimcs code.

The choice of application-level protocols could be another factor. As pointed out above, PVM uses socket routines for interprocess communication, while NCS uses the rpc (remote procedure call) library built on top of XDR. XDR (eXternal Data Representation) is a platform that was built to optimize application code by taking advantage of, among other things, several possible methods of interprocess communication.

6.1.3 NCS

Due to the speed of the NCS version of the application, it was used as the baseline for comparing the other platforms.

The results reported by Almasi et. al., show that the critical point where the manager starts to impede the progress of the workers occurs at 10 machines.[12] This supports my hypothesis that the additional interprocess communication like that initiated by the pvmd daemon is not needed by NCS due to the inclusion of the communication directly into the user's processes.

A discussion of the results produced by the Linda environment are presented in the next section.

6.2 Paradigms

6.2.1 Godzilla vs. client/server (PVM)

The comparisons of Godzilla and client/server paradigms under PVM show, that for this particular application, the client/server model is almost useless when a small number of machines are considered. The results point out the problems with the lack of load balancing provided by the PVM client/server paradigm.

Load balancing can be provided in a number of ways. It can be implemented by the paradigm, as in Godzilla, or it can be provided by the distributed processing environment (e.g. Linda).

When running the PVM client/server version of the application, the manager was observed communicating with one particular worker in multiple cycles before switching to another worker. Adjustments should be made to PVM that would prevent this synchronization of manager and worker that produces the starvation of the other workers.

6.2.2 Godzilla vs. client/server (libimcs)

As shown with client/server runtimes under libimcs, the implementation of the Godzilla paradigm under libimcs significantly slows the runtime of a small number of workstations (3) that are strictly devoted to this application. The load balancing provided by the Godzilla paradigm should result in faster runtimes when processing elements are used which are also being taxed by other processes. This, of course, is the normal circumstance in most computing centers.

What the client/server version of the libimcs application failed to show was an improved time over the Godzilla version

of the NCS application. This points to the fact that, for some reason, the raw speed of the fiber optic network is still not being taken advantage of by the libimcs code and/or the if_imcs and IMCS system software.

6.2.3 Linda vs. Godzilla

The first version of the application using Linda used separate processes for the initialization of the data as well as the report handling. This is version 1 in Chapter 5. Executing this version on a small number of processors often produced deadlock due to the additional initialization and reporting processes. The controlling process in C-Linda (tsnet), attempts to balance the processes according to the work load observed on each machine. With a large number of processes (e.g. 4 - manager, worker, initializer, and reporter) running on a relatively small number of machines (e.g. 1 to 3), tsnet would refuse to start a given process. Since that process was missing, the entire program would halt due to deadlock.

Version 2 used the manager to handle the reporting (which was done only once at the end of the program) and the initialization. This version enabled me to utilize the small number of processing elements more efficiently. Instead of using one machine solely for the manager, which used less than half of the CPU time during a given run, I was able to use it to run a worker as well. This method produced results that were better for comparisons with the other environments and paradigms. (However, I was not able to get this version to run on one machine due to the deadlock problem explained above).

The straight comparison of the Linda application to the others is very difficult due to the major differences between the paradigms. For example, the frequency data objects were dispatched into the tuple space (TS) before starting any of the workers to avoid deadlock resulting from having more processes than processors. This makes comparing the runtime during the processing of these objects difficult. It is interesting to note that while the total number of packets going out across the network compared very closely to that of

NCS, the rate at which these packets moved was much slower (2 packets/sec) than NCS (5 packets/sec).

Another interesting discovery was that "getrusage" showed that Linda uses signals to communicate just as libimcs. Even though the software interrupts would seem to provide a quicker mode of communication, it may be that their use, due to implementation specifics, is actually slower than socket communication.

The difference between the message-passing and Linda runtimes shows that the added convenience of simulated shared memory on a distributed system does not come without cost. Similar to PVM, Linda uses another executable, tsnet, to control the interprocess communication. Unlike PVM, this routine is called with the manager executable as one of the arguments. (It is interesting to note that, due to this arrangement, Linda applications cannot accept input from standard input). The tsnet process then runs the manager and is the only process running on the manager's processor (it does not fork another process). Also, tsnet runs only on the

manager's node and not on any of the worker's, unlike pvmd.

Even though the paradigms are very different as are the communication modes (signals vs. sockets), during the processing cycle of each frequency data object the same number of packets should be observed going out across the network since both are using the token ring network (the signal packets should observe the MTU set by the system). However, as stated above, the rate observed with Linda is much lower indicating that more time is used satisfying each worker request for data. Although I do not have access to the Linda source code (it is a commercial product), I believe that the slower packet rate observed, as compared to NCS, is due to the overhead of maintaining the tuple space which is described below.

Comparing Godzilla and Linda, both paradigms use distributed naming at the application level. In other words, data can be sent to any process from any other. In Godzilla, a frequency data object is requested from the manager when the worker is ready. A specific data object could be given to a

different worker each time the program is run depending on the relative processing loads of each at runtime. Likewise, a given data object could be "ined" by any of the Linda worker processes.

The difference appears at the implementation level of the paradigm. In Godzilla, the worker signals the manager (whose id is known by each worker) that it is ready for data. When that message is received, the manager knows which worker sent it. Therefore, the data is sent in the form of a message from the manager to a particular worker.

A quite different method is used by Linda. The worker does not need to communicate with the manager. Instead it must send the request to tsnet which scans the tuple space for a data object that matches the arguments in the "in" or "rd". This scanning and other tuple management functions, such as removing the tuple when it is "ined", is the overhead mentioned above.

The fact that the frequency data objects are released into TS before any of the workers start does make a significant difference in timing. This process was timed at a little over 2 seconds, and when subtracted from the Linda runtime, shows only a 4 second difference between the Linda and NCS versions using 5 processing elements. The overhead of running the tsnet code and performing the TS management functions may account for the balance of this additional time.

Chapter 7

FUTURE ENHANCEMENTS

7.1 libimcs

Recently, IBM has released a new SLA card to control the intermachine fiber-optic connections. Unfortunately, this new hardware was incompatible with our software (including the kernel extensions), and a new release of the software was not available. When a new release of the software is available, it may be that some of the problems that cause the slow transmission of messages have been addressed.

If not, the best course of action would be to obtain the source code to the kernel extensions and deduce where communication bottleneck occurs. Since the packet size limit of TCP/IP (65,535 bytes) is higher than the reported MTU of IMCS (28,672 bytes), it seems that the use of TCP/IP by IMCS could improve the performance. Instead of using routines which execute in the user's process space to handle the communication, those services could be provided by system

calls which can run at higher priorities. In addition, the use of TCP/IP would provide a more efficient alternative to the socket polling alternative currently used. Instead of constantly polling the socket, in the user's process space, the TCP/IP system routines would handle communication at higher system priorities. It may be that TCP/IP was bypassed for performance reasons, but it certainly seems that it could be an improvement on the current method.

Once IMCS is tied into a reliable low level protocol, the fiber-optic network should be made available to the programmer through the remote procedure call library on the RS/6000. This would provide users with a familiar interface through which the fiber-optic network could be used.

A choice of paradigms could then be added as a higher level of application software. Client/server could be supported, as well as Godzilla, in order to provide some load balancing. Linda, while not the fastest solution, seems to be the easiest paradigm to use from the programmer's point of view. The small number of primitives that are added to a base

language facilitate the porting of applications to parallel environments.

At present, libimcs is not as easy to work with as PVM. PVM provides data buffer routines that eliminate the need for explicit user-defined data structures. The addition of this kind of data management could improve libimcs.

7.2 PVM

As mentioned in Chapter 6, PVM was designed to be highly portable. In order to obtain the maximum speedup from this environment, the source code could be modified to take advantage of the RS/6000 hardware. IBM has produced guidelines for performance tuning that could be used for this purpose.[13]

The PVM software supports the use of UDP as well as TCP. UDP is a datagram transport protocol as opposed to a streams protocol. The advantage of using UDP is the additional speed

due to a smaller amount of error checking done by the network software. The "netstat" utility reports that PVM uses TCP, but this has been shown not to be reliable. With a few changes to the PVM library, PVM could be altered to use UDP exclusively.

Another approach to improvement would be to modify PVM to take advantage of the platforms that NCS is built upon, RPC and XDR. XDR (eXternal Data Representation) is a platform that provides a standard for describing data that is independent of the system hardware. This provides the basis for the access of different hardware to be used in distributed processing. The use of RPC library by PVM would provide a more versatile programming environment by giving the application access to a wider range of communication modes and could also eliminate the need for the pvmd daemon.

7.3 NCS

NCS can also use UDP for message passing. As mentioned in Chapter 4, NCS stub code is produced by a pre-compiler to enable the programmer to avoid writing the actual interprocess communication code. However, it may be that faster communication modes exist other than those chosen by the pre-compiler.

As mentioned above, if IMCS was made available through the RPC library, NCS would have an additional, and hopefully faster, method of communication available.

While the pre-compiler hides the actual coding of the communication software from the programmer, the code produced is very difficult to read. Another improvement to NCS would be the additional support of the Linda primitives. This would go one step further towards NCS's stated goal of hiding the communication implementation details from the user. I found NCS the most difficult environment to work use.

7.4 Linda

The Linda environment could be improved by adding the support of the XDR standard, assuming that XDR becomes available on more platforms. This would provide the programmer with access to a variety of hardware platforms for distributed programming.

One drawback of the Linda environment is the inability to support standard input to the application. If the RPC library was supported by Linda, the need for tsnet could be eliminated on the RS/6000 platform since the RPC library provides a location broker. While this could improve the performance of the applications executed in this manner, the portability of C-Linda would be hurt.

7.5 Conclusion

This report shows that, on the RS/6000, no one parallel processing environment is superior to the others. Linda is the best environment in terms of ease of use and the small

learning curve owing to the number of primitives and implementation of shared memory in software which provides communication encapsulation.

The rpc library and location broker built upon NCS and XDR provide a good platform for interprocess communication between distributed processing elements, but lack the ease of use for parallel programming provided by Linda. Even the other message passing environments (libimcs and PVM) seem easier use to due to fact that the programmer need not learn how to use an additional pre-compiler and the location broker.

The lack of load-balancing within PVM is its major drawback while portability to other platforms is its strongest asset. IMCS still suffers from the inability of the system software to take advantage of the raw speed offered by the fiber-optic connections. When this problem is solved, its addition to the NCS/XDR platform would provide the programmer with an environment that would be excellent for communication-intensive applications.

The best possible solution from the programmer's point of view would be an environment that provided Linda as the interface to very fast communication routines, that provide the programmer with the option of using fiber-optic networks.

REFERENCES CITED

[1] Markoff, John, "David Gelernter's Romance With Linda", The New York Times, Sunday, January 19, 1992

[2] Milton B. Dobrin, "Introduction to Geophysical Prospecting" 1976 McGraw-Hill Inc

[3] R. L. Sengbush, "Seismic Exploration Methods", 1983 IHRDC Publishers, pp. 171-185.

[4] W. M. Telford, L. P. Geldart, R. E. Sheriff, D. A. Keys, "Applied Geophysics", 1988, Press Syndicate of the University of Cambridge, pp. 357-362.

[5] D. Hale, "Stable Explicit Depth Extrapolation of Seismic Wave Fields", 60th Annual International Meetings, Society of Exploration Geophysics, Expanded Abstracts, pp. 1301-1304, 1990.

[6] Morris Meyer, "The Godzilla Approach to Distributed Computing", BuzzNUG Buzzings, p. 33, March 1990.

[7] Scientific Computing Associates, Inc., "C-Linda Reference Manual", 1990, Scientific Computing Associates, Inc.

[8] D. Gelernter, "Generative Communication in Linda", ACM Transactions on Programming Languages and Systems, Vol.7, No. 1, January 1985, pp. 80-112.

[9] W. Richard Stevens, "Unix Network Programming", 1990, Prentice-Hall, Inc, pp. 173-174

[10] G. Almasi, A. Gottlieb, "Highly Parallel Computing", 1989 Benjamin/Cummings, p. 24.

[11] W. Richard Stevens, "Unix Network Programming", 1990, Prentice-Hall, Inc, p. 186

[12] G. Almasi, T. McLuckie, J. Bell, A. Gordon, D. Hale, "Parallel Distributed Seismic Migration"

[13] R. Bell, "IBM RISC System/6000 Performance Tuning for Numerically Intensive FORTRAN and C Programs", 1990, Technical and Supercomputing Centre of Competence, IBM United Kingdom Ltd

Appendix A
User Guide to libimcs

The routines outlined in this guide are called by directly from the user's application. The object files are located in the library "libimcs".

The parallel processing paradigm that these routines were designed to employ is called the Godzilla paradigm. The client starts the servers as in a normal client/server model, but then each server is free to decide what resources or actions it may request from the client. Due to these differences, the client is often referred to as the manager while the servers are called workers.

The Godzilla paradigm allows each worker to request data independent of the manager. Each worker requests data from the manager, processes the data, and may optionally manage subtotals of the results on the remote machine. If the summation of the data is left to the workers, the manager is free to satisfy more worker requests for data.

An ascii file, called the remotes file, is used to define each worker node to be used within the paradigm.

Main User Routines

start_shells

Called by: User's manager application program.
Prototype: int start_shells(int verbose, unsigned long saveport, char *Command, char *RemotesFile, char *Path, int Host_as_Worker)
Description: Begins the execution of the worker program on each node found in the remotes file and then opens an imcs communication path to that machine.

If the value of verbose is not 0, the command line sent to the remote node will be echoed.

The value of saveport will be the starting port address used for worker/manager communication.

The value of Command will be used as the name of the program to be executed on the remote node using the fork command.

The value of RemotesFile is the name of the remotes file to be passed to remotesInit. This variable can be null in which case the default file name "./.remotes" will be used.

The value of Path is the used as the path of the worker executable.

If Host_as_Worker is not 0, the manager node will also execute the worker program.

Returns: 0 on success; forced exit otherwise
Error Messages: if call to open_imcs fails: "imcs open error"

start_workers

Called by: User's worker application program.
Prototype: int start_workers(int argc, char* argv[], char** rtn_hostname, int *verbose, int* id_ptr)
Description: Opens a communication path to the manager with open_imcs.

Sets the return arguments (rtn_hostname = name of the worker node), verbose, and (id_ptr = process id of the worker node(s)).

Returns: N/A (void)
Error Messages: if call to open_imcs fails: "open error"

tell_imcs

Called by: User's worker application program.
Prototype: int tell_imcs(int id, int sending)
Description: Uses the socket routine "send" to send one character to the manager indicating that a processing cycle is being requested. This routine uses an interrupt mechanism to call the routine which must be called "handler" by the manager application. (See "Additional Recommended User Routines").

The first argument is the process id of the worker requesting attention.

The second argument should be sent to 1. A value of 0 signifies a receive operation and is not currently supported.

Returns: 0 on success; otherwise an incorrect process id was given

Error Messages: Warning message given if a value of 0 is given to sending.

send_imcs

Called by: User's manager or worker application program.

Prototype: int send_imcs(int id, caddr_t buffer, int size, int urgent)

Description: Sends the buffer pointed to by "buffer" and having the size "size" to the process "id".

A non-zero value for the urgent flag causes the routine to make calls to the sighold and sigrelease socket routines to support critical region code. This should only be used by the manager to prevent interrupts from other workers.

Returns: 0 on success; -1 if communication fails

Error Messages: none

recv_imcs

Called by: User's manager or worker application program.

Prototype: int recv_imcs(int id, caddr_t buffer, int size, int urgent)

Description: Receives the message buffer into the structure pointed to by "buffer" and having the size "size" from the process "id".

A non-zero value for the urgent flag causes the routine to make calls to the sighold and sigrelease socket routines to support critical region code. This should only be used by the manager to prevent interrupts from other workers.

Returns: size of buffer sent on success; -1 if communication fails

Error Messages: indicates which socket routine failed

Other Library routines

remotesInit

Called by: start_shells
Prototype: void remotesInit(char *remotesFile)
Description: Checks for the existence of a remotes file (which contains the node name and process id of each imcs node to be used in processing) and sets a global file pointer for use in remotesNext and remotesFinit.

If the argument (remotesFile) is null, the environment variable RemotesEnvName is checked. If it is also null, the default name of "./.remotes" is used.

Returns: N/A (void)
Error Messages: if file not found or unable to open: "error: can't open remotes file %s\n", remotesFile

remotesNext

Called by: start_shells
Prototype: int remotesNext(char **name, int *procid)
Description: Returns the next name, process id pair from the remotes file that was opened by remotesInit.
Returns: 0 if both name and process id not found (or eof)
1 on success
Error Messages: if global file pointer (see remotesInit) is null: "error: remotes file not initialized\n");

remotesFinit

Called by: start_shells
Prototype: void remotesFinit()
Description: Closes the file pointed to by the global pointer set in remotesInit if that pointer is not null (i.e. remotesInit was not successfully called).

Returns: N/A (void)
Error Messages: N/A

Additional Recommended User Routines

These routines can be used by the user application to handle the interrupts caused by tell_imcs.

```
/*
 * Queue for pushing processor ids
 */
static int head, length=0, idtable[MAX_WORKERS];
static int emptyqueue()
{
    return !length;
}
static void enqueue(int id)
{
    idtable[(head+length) % MAX_WORKERS] = id;
    length++;
}
static int dequeue()
{
    int rval = idtable[head];
    head = ++head % MAX_WORKERS;
    length--;
    return rval;
}

/*
Handling routine for installing in tell_imcs.
*/
void handler(int id, int sending)
{
    enqueue(id);
}
```

Example Application

The following files provide an example of a simple application using the libimcs routines.

makefile

```
# Makefile
INCLUDES = -I../include
LIBS     = -L../lib

OPTC = -O -qxflag=hsflt

.c.o:
       $(CC) -c $(CFLAGS) $(INCLUDES) $<
```

```
all: manager workers

manager: manager.o
	$(CC) -o manager manager.o \
		$(LIBS) -lzilla\
		-lbsd -lm /lib/crt0.o -lc -bimport:../lib/imcs_tool.exp\
		 -bimport:../lib/imcs.exp

workers: workers.o
	$(CC) -o workers workers.o \
		$(LIBS) -lzilla\
		-lbsd -lm /lib/crt0.o -lc -bimport:../lib/imcs_tool.exp\
		  -bimport:../lib/imcs.exp
	cp workers $(HOME)/bin


manager.o: manager.h routines.h
workers.o: manager.h routines.h
```

Header Files

manager.h

```
#ifndef _manager_
#define _manager_

#include <signal.h>
#include "routines.h"

void go();
void handler(int,int);

#endif
```

routines.h

```
#ifndef _routines_
#define _routines_

#include "cwp.h"
#include "zilla.h"

#define MAX_ROUTINES (16)
#define MAX_NX       (0x00008000)
#define MAX_NX_NZ    (0x001000000)
#define MAX_BUFFER   (sizeof(AnyMsg))

enum {
    INIT,
    NEXT_FREQUENCY,
    REPORT_WORK
};
```

```
typedef struct _SimpleSend {
    unsigned long routine;
    char hostname[32];
} SimpleSend;

typedef struct _ReportSend {
    SimpleSend hdr;
    int num;
} ReportSend;

typedef struct _InitReply {
    int num;
} InitReply;

typedef struct _NextReply {
    int done;
    int num;
} NextReply;

typedef union _AnyMsg {
    SimpleSend header;
    ReportSend report;
    InitReply init;
    NextReply next;
} AnyMsg;

#endif
```

workers.h

```
#ifndef _workers_
#define _workers_

#include "routines.h"

void miget_init(int, char*, int*, int*, int*, eTable**, float***);
int miget_next_frequency(int, char*, float*, complex**);
void miget_report_work(int, char*, int, int, int, float**);

#endif
```

C Files

manager.c

```
#include <signal.h>
#include "manager.h"

/*
Table to hold addresses of rpc's.
*/
static struct Routines
{
    void (*address)();
} routines[MAX_ROUTINES];

int num;
int num_times=5;
int count;

static AnyMsg buffer;

int verbose;

int main(int argc, char **argv)
{
unsigned long saveport;
int status=0;
int Host_As_Worker=0;
static char
    *Command     = "workers",
    *RemotesFile = NULL,
    *Path        = "${HOME}/bin";

/* do application initialization here */

        count = 0;
        num = 1;
        verbose = 1;
        saveport = 1500;
        status = start_shells(verbose, saveport, Command, RemotesFile, Path,
                              Host_As_Worker);
        if (status) {
                printf("start_shells error: %d\n",status);
                return(0);
        }
        go();
}

/*
Routine to send the initial data to the workers.
*/
static int reported = 0;
/* MUST be named manager_init */
void manager_init(int id, SimpleSend *rcv)
{
int status=0;
```

```
    InitReply *reply = (InitReply*)&buffer;

    if (verbose)
        fprintf(stderr, "%s calling appl_init()\n", rcv->hostname);

    reply->num = num;

    /* urgent send */
    status = send_imcs(id, reply, sizeof(InitReply), 1);
    if (status == -1)
    {
        perror("manager send error 1 in init");
        exit(-1);
    }
}


/*
Routine to send new frequency tables.
*/
void manager_next_frequency(int id, SimpleSend* rcv)
{
int status=0;
    NextReply *reply = (NextReply*)&buffer;

    if (verbose)
        fprintf(stderr, "%s calling appl_next_frequency()\n", rcv->hostname);

    if (count > num_times)
    {
        reply->done = 1;
        /* urgent send */
        status = send_imcs(id, reply, sizeof(NextReply), 1);
        if (status == -1)
        {
            perror("manager send error 1 in next");
            exit(-1);
        }
    }
    else
    {
        int i, kw;

       if (verbose)
            fprintf(stderr, "%d/%d\n", count, num_times);

        reply->done = 0;
        reply->num = num;

        /* urgent send */
        status = send_imcs(id, reply, sizeof(NextReply), 1);
        if (status == -1)
        {
            perror("manager send error 2 in next");
            exit(-1);
        }
```

```
        count++;
    }
}


/*
Routine to report work.
*/
static int results;
void manager_report_work(int id, ReportSend *report, int* alldone)
{
    int ix, iz;
    float *temp;

    if (verbose)
        fprintf(stderr, "%s calling app1_report_work()\n",
            report->hdr.hostname);

fprintf(stderr,"reported num=%d\n",report->num);
    reported += report->num;
fprintf(stderr,"reported =%d\n",reported);

    if ((count == num_times+2) || (reported == 1200)) *alldone = 1;
    else *alldone = 0;
}



/*
Routine to implement communication loop.
*/
void go()
{
int status;
int alldone=0;
AnyMsg buffer;

    routines[INIT].address = manager_init;
    routines[NEXT_FREQUENCY].address = manager_next_frequency;
    routines[REPORT_WORK].address = manager_report_work;
    while (!alldone)
    {
        int id;

        while (emptyqueue()) {
               if (alldone) return;
        }

        /* busy wait */ ;
        id = dequeue();

        /* urgent receive */
        status = recv_imcs(id, &buffer, MAX_BUFFER, 1);
        if (status == -1)
        {
            perror("recieve error 1 in go");
            exit(-1);
```

```
        }
        switch (buffer.header.routine)
        {
        case INIT:
            (*routines[INIT].address)(id, &buffer);
            break;
        case NEXT_FREQUENCY:
            (*routines[NEXT_FREQUENCY].address)(id, &buffer);
            break;
        case REPORT_WORK:
            (*routines[REPORT_WORK].address)(id, &buffer, &alldone);
            break;
        }
    }
}

/*
Queue data structure for saving id's of requesting workers.  These
should be atomic operations.
*/
static int head, length=0, idtable[MAX_WORKERS];
static int emptyqueue()
{
    return !length;
}
static void enqueue(int id)
{
    idtable[(head+length) % MAX_WORKERS] = id;
    length++;
}
static int dequeue()
{
    int rval = idtable[head];
    head = ++head % MAX_WORKERS;
    length--;
    return rval;
}

/*
Handling routine for installing in tell_imcs.
*/
void handler(int id, int sending)
{
    enqueue(id);
}
```

workers.c

```
#include "routines.h"

void app1_report_work(int id, char *hostname, int num);
static AnyMsg buffer;

int main(int argc, char* argv[])
{
```

```
int verbose,id;
char *hn_ptr;
    start_workers(argc, argv, &hn_ptr, &verbose, &id);

    /* do initialization here */
    if (verbose)
        fprintf(stderr, "Host (%s) started.  Entering work loop.\n", hn_ptr);

    /* call user incremental routine until user signals to end */
    while (!usr_next(id, hn_ptr, verbose));
}

int usr_next(int id, char* hostname, int verbose)
{
int num=0;
double dummy=1.0,dummy2=2.0;
double ans=0.0,seed=0.5;
int i,j;

        if (!app1_next_frequency(id, hostname, &num))
        {
            app1_report_work(id, hostname, num);
            return(1);
        }

        /* work */

        for (i=0; i<10000; i++)
           for (j=0; j<2000; j++)
                ans = ans + (seed * dummy2);

        num = (int)(ans/100000);
        app1_report_work(id, hostname, num);
        return(0);
}

void app1_init( int id, char *hostname, int *num)
{
    SimpleSend *send = (SimpleSend*)&buffer;
    InitReply *reply = (InitReply*)&buffer;

    if (tell_imcs(id, 1) == -1)
    {
        perror("tell error 1 in init");
        exit(-1);
    }

    send->routine = INIT;
    strcpy(send->hostname, hostname);
    if (send_imcs(id, send, sizeof(SimpleSend),0) == -1)
    {
        perror("worker send error 1 in init");
        exit(-1);
    }

    if (recv_imcs(id, reply, sizeof(InitReply), 0) == -1)
    {
```

```
        perror("worker receive error 1 in init");
        exit(-1);
    }
    *num = reply->num;
}

int appl_next_frequency(int id, char *hostname, int *num)
{
    SimpleSend *send = (SimpleSend*)&buffer;
    NextReply *reply = (NextReply*)&buffer;

    if (tell_imcs(id, 1) == -1)
    {
        perror("tell error 1 in next");
        exit(-1);
    }

    send->routine = NEXT_FREQUENCY;
    strcpy(send->hostname, hostname);
    if (send_imcs(id, send, sizeof(SimpleSend),0) == -1)
    {
        perror("worker send error 1 in next");
        exit(-1);
    }
    if (recv_imcs(id, reply, sizeof(NextReply), 0) == -1)
    {
        perror("worker receive error 1 in next");
        exit(-1);
    }
    if (reply->done)
        return 0;
    *num = reply->num;
    return 1;
}


void appl_report_work(int id, char *hostname, int num)
{
    ReportSend *send = (ReportSend*)&buffer;

    if (tell_imcs(id, 1) == -1)
    {
        perror("tell error 1 in report");
        exit(-1);
    }

    send->hdr.routine = REPORT_WORK;
    strcpy(send->hdr.hostname, hostname);
    send->num = num;
    if (send_imcs(id, send, sizeof(ReportSend),0) == -1)
    {
        perror("worker send error 1 in report");
        exit(-1);
    }
}
```

Appendix B
Serial Version of the Application

```
/* This version for demonstrating accumulation of frequencies */

#include "par.h"
#include "extrap.h"
#include <time.h>

char *sdoc =
"MIGET2 - MIGration via an Extrapolator Table of 2-D zero-offset data\n"
"\n"
"miget2 <infile >outfile efile= vfile= nt= nx= nz= [optional parameters]\n"
"\n"
"Required Parameters:\n"
"efile=                 name of file containing extrapolator table\n"
"vfile=                 name of file containing v(x,z)\n"
"nt=                    number of time samples\n"
"nx=                    number of inline samples (traces)\n"
"nz=                    number of depth samples\n"
"\n"
"Optional Parameters:\n"
"dt=1.0                 time sampling interval\n"
"dx=1.0                 inline sampling interval (trace spacing)\n"
"dz=1.0                 depth sampling interval\n"
"freqs=0,0,fNyq,fNyq    corner frequencies of trapezoidal bandpass filter\n"
"ntpad=nt               number of zero samples by which to pad time axis\n"
"verbose=0              =1 for diagnostics on standard error\n"
"\n";

/* functions declared and used internally */
void makejw(int iwmin, int iwmax, int *jw);

main (int argc, char **argv)
{
        int nt,nx,nz,ntpad,nw,ntfft,it,ix,iz,
                iw,iw0,iw1,iw2,iw3,iwmin,iwmax,nfreqs,verbose,mw,*jw,kw;
        float dt,dx,dz,freqs[4],dw,fw,w,scale,fftscl,qscale,pscale,
                *p,*wdxov,*temp,**dx2ov,**q;
        complex **cp,*cpx;
        char *efile="",*vfile="";
        FILE *infp=stdin,*outfp=stdout,*efp,*vfp;
        eTable *et;

        /* hook up getpar to handle the parameters */
        initargs(argc,argv);
        askdoc(0);

        /* get required parameters */
        if (!sgetpar("efile",&efile)) err("must specify efile!\n");
        if (!sgetpar("vfile",&vfile)) err("must specify vfile!\n");
        if (!igetpar("nt",&nt)) err("must specify nt!\n");
        if (!igetpar("nx",&nx)) err("must specify nx!\n");
        if (!igetpar("nz",&nz)) err("must specify nz!\n");
```

```
/* get optional parameters */
if (!fgetpar("dt",&dt)) dt = 1.0;
if (!fgetpar("dx",&dx)) dx = 1.0;
if (!fgetpar("dz",&dz)) dz = 1.0;
if (!igetpar("ntpad",&ntpad)) ntpad=nt;
if ((nfreqs=fgetpar("freqs",freqs))==0) {
        freqs[0] = 0.0;
        freqs[1] = 0.0;
        freqs[2] = 0.5/dt;
        freqs[3] = 0.5/dt;
} else if (nfreqs!=4) {
        err("less than 4 freqs specified!\n");
}
if (!igetpar("verbose",&verbose)) verbose = 0;
if (!igetpar("mw",&mw)) mw=9999999;

/* determine frequency w sampling */
ntfft = npfar(nt+ntpad);
nw = ntfft/2+1;
dw = 2.0*PI/(ntfft*dt);

/* read extrapolator table */
if ((efp=fopen(efile,"r"))==NULL)
        err("error opening efile=%s\n",efile);
et = etread(efp);

/* read and normalize velocities */
dx2ov = alloc2float(nx,nz);
if ((vfp=fopen(vfile,"r"))==NULL)
        err("error opening vfile=%s\n",vfile);
temp = alloc1float(nz);
for (ix=0; ix<nx; ++ix) {
        fread(temp,sizeof(float),nz,vfp);
        for (iz=0; iz<nz; ++iz)
                dx2ov[iz][ix] = dx*2.0/temp[iz];
}
free1float(temp);

/* read and Fourier transform input data */
p = alloc1float(ntfft);
cp = alloc2complex(nw,nx);
for (ix=0; ix<nx; ++ix) {
        if (fread(p,sizeof(float),nt,infp)!=nt)
                err("error reading trace number %d\n",ix+1);
        for (it=nt; it<ntfft; ++it)
                p[it] = 0.0;
        pfarc(1,ntfft,p,cp[ix]);
}
free1float(p);

/* determine sample indices for frequency filter */
iw0 = MAX(0,MIN(nw-1,NINT(2.0*PI*freqs[0]/dw)));
iw1 = MAX(0,MIN(nw-1,NINT(2.0*PI*freqs[1]/dw)));
iw2 = MAX(0,MIN(nw-1,NINT(2.0*PI*freqs[2]/dw)));
iw3 = MAX(0,MIN(nw-1,NINT(2.0*PI*freqs[3]/dw)));

/* apply frequency filter and FFT scaling */
```

```
for (iw=0; iw<nw; ++iw) {
        fftscl = (iw==0 || iw==nw-1 ? 1.0/ntfft : 2.0/ntfft);
        if (iw<iw0)
                scale = 0.0;
        else if (iw0<=iw && iw<iw1)
                scale = (float)(iw-iw0)/(float)(iw1-iw0)*fftscl;
        else if (iw1<=iw && iw<=iw2)
                scale = fftscl;
        else if (iw2<iw && iw<=iw3)
                scale = (float)(iw3-iw)/(float)(iw3-iw2)*fftscl;
        else
                scale = 0.0;
        for (ix=0; ix<nx; ++ix) {
                cp[ix][iw].r *= scale;
                cp[ix][iw].i *= scale;
        }
}
iwmin = iw0;
iwmax = iw3;

/* allocate workspace */
jw = alloc1int(nw);
q = alloc2float(nx,nz);
wdxov = alloc1float(nx);
cpx = alloc1complex(nx);

/* zero migrated data */
for (iz=0; iz<nz; ++iz)
        for (ix=0; ix<nx; ++ix)
                q[iz][ix] = 0.0;

/* compute frequency indices */
makejw(iwmin,iwmax,jw);

/* loop over frequencies w */
for (iw=iwmin; iw<=iwmax; ++iw) {

        /* compute frequency */
        kw = jw[iw];
        w = kw*dw;

        /* progress report */
        if (verbose) fprintf(stderr,"%d/%d\n",iw-iwmin,iwmax-iwmin);

        /* load wavefield */
        for (ix=0; ix<nx; ++ix)
                cpx[ix] = cp[ix][kw];

        /* compute scale factors */
        qscale = (float)(iw-iwmin)/(float)(iw-iwmin+1);
        pscale = (float)(iwmax-iwmin)/(float)(iw-iwmin+1);

        /* loop over depths z */
        for (iz=0; iz<nz; ++iz) {

                /* accumulate migrated data */
                for (ix=0; ix<nx; ++ix)
```

```
                        q[iz][ix] = qscale*q[iz][ix]+pscale*cpx[ix].r;

                /* make w*dx/v(x) */
                for (ix=0; ix<nx; ++ix)
                        wdxov[ix] = w*dx2ov[iz][ix];

                /* extrapolate wavefield */
                etextrap(et,nx,wdxov,cpx);
        }

        /* write migrated data */
        if ((iw-iwmin)%mw==mw-1) {
                temp = alloc1float(nz);
                for (ix=0; ix<nx; ++ix) {
                        for (iz=0; iz<nz; ++iz)
                                temp[iz] = q[iz][ix];
                        fwrite(temp,sizeof(float),nz,outfp);
                }
                free1float(nz);
        }
    }

    /* write migrated data */
    temp = alloc1float(nz);
    for (ix=0; ix<nx; ++ix) {
        for (iz=0; iz<nz; ++iz)
                temp[iz] = q[iz][ix];
        fwrite(temp,sizeof(float),nz,outfp);
    }
    free1float(nz);

    /* free workspace */
    free1float(wdxov);
    free2float(dx2ov);
    free2float(q);
    free1complex(cpx);
    free2complex(cp);
}

/* make frequency indices (triangular random) */
void makejw (int iwmin, int iwmax, int *jw)
{
    int kwstep=iwmax-iwmin,nw=0,kw,iw;

    /* loop until done */
    while (nw<iwmax-iwmin+1) {

        /* make a random kw */
        kw = iwmin+franuni()*(iwmax-iwmin+1);
        kw += iwmin+franuni()*(iwmax-iwmin+1);
        kw /= 2;

        /* search for kw in jw list */
        for (iw=0; iw<nw; ++iw)
                if (kw==jw[iw]) break;

        /* if kw not in list, add it */
```

```
                if (iw==nw)
                        jw[nw++] = kw;
        }
}
```

Appendix C
libimcs/Godzilla Version

NOTE: The application code that this version has in common with the serial version has been omitted.

```
#include "par.h"
#include "extrap.h"
#include "manager.h"

char *sdoc =
"MIGET2 - MIGration via an Extrapolator Table of 2-D zero-offset data\n"
"\n"
"miget2 <infile >outfile efile= vfile= nt= nx= nz= [optional parameters]\n"
"\n"
"Required Parameters:\n"
"efile=                 name of file containing extrapolator table\n"
"vfile=                 name of file containing v(x,z)\n"
"nt=                    number of time samples\n"
"nx=                    number of inline samples (traces)\n"
"nz=                    number of depth samples\n"
"\n"
"Optional Parameters:\n"
"dt=1.0                 time sampling interval\n"
"dx=1.0                 inline sampling interval (trace spacing)\n"
"dz=1.0                 depth sampling interval\n"
"freqs=0,0,fNyq,fNyq    corner frequencies of trapezoidal bandpass filter\n"
"ntpad=nt               number of zero samples by which to pad time axis\n"
"verbose=0              =1 for diagnostics on standard error\n"
"port=1500              port number for communicating with workers\n"
"\n";

/* functions declared and used internally */
void makejw(int iwmin, int iwmax, int *jw);

/* variables declared and used internally */
int nx, nz, iw, iwmin, iwmax, *jw, mw;
float dw, **q, **dx2ov;
complex **cp;
eTable *et;
FILE *outfp = stdout;
static char *ProgName    = NULL;
int verbose;

int main(int argc, char **argv)
{
unsigned long saveport;
int status=0;
int Host_As_Worker=0;
static char
    *Command     = "miget2accd",
    *RemotesFile = NULL,
    *Path        = "${HOME}/bin";

    int nt,ntpad,nw,ntfft,it,ix,iz,iw0,iw1,iw2,iw3,nfreqs;
```

```
    unsigned long nlength;
    float dt,dx,dz,freqs[4],w,scale,fftscl,qscale,pscale,*p,**v;
    char *efile="",*vfile="";
    FILE *infp=stdin,*efp,*vfp;

/* do application initialization here */

        .
        .
        .

    /* compute frequency indices */
    makejw(iwmin,iwmax,jw);

        status = start_shells(verbose, saveport, Command, RemotesFile, Path,
                              Host_As_Worker);
        if (status) {
                printf("start_shells error: %d\n",status);
                return(0);
        }
        go();
}


#include "par.h"
#include "extrap.h"
#include "manager.h"
#include <signal.h>

/*
Table to hold addresses of rpc's.
*/
static struct Routines
{
    void (*address)();
} routines[MAX_ROUTINES];

extern int nx, nz, iw, iwmin, iwmax, verbose, *jw, mw;
extern float dw, **q, **dx2ov;
extern complex **cp;
extern eTable *et;
extern FILE *outfp;


/*
Routine to send the initial data to the workers.
*/
static int reported = 0;
void manager_init(int id, SimpleSend *rcv)
{
AnyMsg buffer;
int status;
    InitReply *reply = (InitReply*)&buffer;

    if (verbose)
        fprintf(stderr, "%s calling miget_init()\n", rcv->hostname);
```

```
    reply->mw = mw;
    reply->nx = nx;
    reply->nz = nz;
    reply->nhmax = et->nhmax;
    reply->nwdxov = et->nwdxov;
    reply->dwdxov = et->dwdxov;
    reply->fwdxov = et->fwdxov;
    reply->dzodx = et->dzodx;

    status = send_imcs(id, reply, sizeof(InitReply), 1);
    if (status == -1)
    {
        perror("manager send error 1 in init");
        exit(-1);
    }

    status = send_imcs(id, et->nh, et->nwdxov * sizeof(int), 1);
    if (status == -1)
    {
        perror("manager send error 2 in init");
        exit(-1);
    }

    status = send_imcs(id, et->e[0], et->nhmax * et->nwdxov * sizeof(complex), 1);
    if (status == -1)
    {
        perror("manager send error 3 in init");
        exit(-1);
    }

    status = send_imcs(id, dx2ov[0], nx * nz * sizeof(float), 1);
    if (status == -1)
    {
        perror("manager send error 4 in init");
        exit(-1);
    }
}


/*
Routine to send new frequency tables.
*/
void manager_next_frequency(int id, SimpleSend* rcv)
{
AnyMsg buffer;
int status;
    NextReply *reply = (NextReply*)&buffer;

    if (verbose)
        fprintf(stderr, "%s calling miget_next_frequency()\n", rcv->hostname);

    if (iw > iwmax)
    {
        reply->done = 1;
        status = send_imcs(id, reply,
            sizeof(NextReply) - MAX_NX*sizeof(complex), 1);
        if (status == -1)
```

```
        {
            perror("manager send error 1 in next");
            exit(-1);
        }
    }
    else
    {
        int i, kw;

       if (verbose)
            fprintf(stderr, "%d/%d\n", iw-iwmin, iwmax-iwmin);

        reply->done = 0;
        kw = jw[iw];
        reply->pw = kw*dw;
        for (i = 0; i < nx; ++i)
            reply->cpx[i] = cp[i][kw];

        status = send_imcs(id, reply,
            sizeof(NextReply) - (MAX_NX-nx)*sizeof(complex), 1);
        if (status == -1)
        {
            perror("manager send error 2 in next");
            exit(-1);
        }

        iw++;
    }
}


/*
Routine to report work.
*/
static float **results = NULL;
void manager_report_work(int id, ReportSend *report, int* alldone_ptr)
{
int status;
int ix, iz;
float *temp;

    if (verbose)
        fprintf(stderr, "%s calling miget_report_work()\n",
            report->hdr.hostname);

    if (results == NULL)
        results = alloc2float(nx, nz);

    reported += report->num;

    status = recv_imcs(id, results[0], nx*nz*sizeof(float),1);
    if (status == -1)
    {
        perror("manager receive error 1 in report");
        exit(-1);
    }
    for (ix = 0; ix < nx; ++ix)
```

```
        for (iz = 0; iz < nz; ++iz)
            q[iz][ix] += results[iz][ix];

    /*  fwrite(q[0], sizeof(float), nx*nz, outfp);  */

    /* write migrated data (put in by GSA 6Ap91) */
        {
                float *qtemp=alloc1float(nz);
        for (ix=0; ix<nx; ++ix) {
                        for (iz=0; iz<nz; ++iz)
                                qtemp[iz] = q[iz][ix];
                        fwrite(qtemp,sizeof(float),nz,outfp);
                }
                free1float(qtemp);
        }

    if (reported == iwmax-iwmin+1)
    {
        if (results != NULL)
            free(results);
        *alldone_ptr = 1;
    } else *alldone_ptr = 0;
}



/*
Routine to implement communication loop.
*/
void go()
{
int status;
int alldone=0;
AnyMsg buffer;

    routines[INIT].address = manager_init;
    routines[NEXT_FREQUENCY].address = manager_next_frequency;
    routines[REPORT_WORK].address = manager_report_work;
    while (!alldone)
    {
        int id;

        while (emptyqueue()) {
                if (alldone) return;
        }

            /* busy wait */ ;
        id = dequeue();
        status = recv_imcs(id, &buffer, MAX_BUFFER,1);
        if (status == -1)
        {
            perror("recieve error 1 in go");
            exit(-1);
        }
        switch (buffer.header.routine)
        {
        case INIT:
```

```
            (*routines[INIT].address)(id, &buffer);
            break;
        case NEXT_FREQUENCY:
            (*routines[NEXT_FREQUENCY].address)(id, &buffer);
            break;
        case REPORT_WORK:
            (*routines[REPORT_WORK].address)(id, &buffer, &alldone);
            break;
        }
    }
}

/*
Queue data structure for saving id's of requesting workers.  These
should be atomic operations.
*/
static int head, length=0, idtable[MAX_WORKERS];
static int emptyqueue()
{
    return !length;
}
static void enqueue(int id)
{
    idtable[(head+length) % MAX_WORKERS] = id;
    length++;
}
static int dequeue()
{
    int rval = idtable[head];
    head = ++head % MAX_WORKERS;
    length--;
    return rval;
}

/*
Handling routine for installing in tell_imcs.
*/
void handler(int id, int sending)
{
    enqueue(id);
}



#include "par.h"
#include "extrap.h"
#include "workers.h"

static AnyMsg buffer;

static float **zero(float**, int, int);

static    int mw, nx, nz, count = 0;
static    float **q, *wdxov, **dx2ov;
static    complex *cpx;
static    eTable *et;
extern int id;
```

```
int main(int argc, char* argv[])
{
int verbose,id;
char *hn_ptr;

    start_workers(argc, argv, &hn_ptr, &verbose, &id);

    /* do initialization here */
    /* get initial tables to do problem */
    miget_init(id, hn_ptr, &mw, &nx, &nz, &et, &dx2ov);

    /* allocate workspace */
    q = zero(alloc2float(nx,nz), nx, nz);
    wdxov = alloc1float(nx);

    /* while manager has work to do ask for it and do it */
    if (verbose)
        fprintf(stderr, "Host (%s) started.  Entering work loop.\n", hn_ptr);

    /* call user incremental routine until user signals to end */
    while (!usr_next(id, hn_ptr, verbose));

}

int usr_next(int id, char* hostname, int verbose)
{
int ix, iz;
float w;

        /* get next frequency and check for errors */
        if (!miget_next_frequency(id, hostname, &w, &cpx))
        {
            if (count)
                 miget_report_work(id, hostname, count, nx, nz, q);
            /* free et table */
            free2float(dx2ov);
            free1float(wdxov);
            free2float(q);
            return(1);
        }

        /* loop over depths z */
        for (iz = 0; iz < nz; ++iz)
        {
            /* accumulate migrated data */
            for (ix = 0; ix < nx; ++ix)
                q[iz][ix] += cpx[ix].r;

            /* make w*dx/v(x) */
            for (ix = 0; ix < nx; ++ix)
                wdxov[ix] = w * dx2ov[iz][ix];

            /* extrapolate wavefield */
            etextrap(et, nx, wdxov, cpx);
        }
```

```
        /* report work every mw frequencies */
        if ((count = ++count%mw) == 0)
        {
            miget_report_work(id, hostname, mw, nx, nz, q);
            zero(q, nx, nz);
        }
        return(0);
}

static float **zero(float **q, int nx, int nz)
{
    int i, j;
    for (j = 0; j < nz; ++j)
       for (i = 0; i < nx; ++i)
            q[j][i] = 0.0;
    return q;
}

void miget_init(
    int id, char *hostname,
    int *mw, int *nx, int *nz, eTable **et, float ***dx2ov)
{
    SimpleSend *send = (SimpleSend*)&buffer;
    InitReply *reply = (InitReply*)&buffer;

    if (tell_imcs(id, 1) == -1)
    {
        perror("tell error 1 in init");
        exit(-1);
    }

    send->routine = INIT;
    strcpy(send->hostname, hostname);
    if (send_imcs(id, send, sizeof(SimpleSend),0) == -1)
    {
        perror("worker send error 1 in init");
        exit(-1);
    }
    if (recv_imcs(id, reply, sizeof(InitReply),0) == -1)
    {
        perror("worker receive error 1 in init");
        exit(-1);
    }
    *mw = reply->mw;
    *nx = reply->nx;
    *nz = reply->nz;
    *et = (eTable*)malloc(sizeof(eTable));
    (*et)->nhmax = reply->nhmax;
    (*et)->nwdxov = reply->nwdxov;
    (*et)->dwdxov = reply->dwdxov;
    (*et)->fwdxov = reply->fwdxov;
    (*et)->dzodx = reply->dzodx;

    (*et)->nh = alloc1int((*et)->nwdxov);
    if (recv_imcs(id, (*et)->nh, (*et)->nwdxov * sizeof(int),0) == -1)
    {
        perror("worker receive error 2 in init");
```

```
        exit(-1);
    }

    (*et)->e = alloc2complex((*et)->nhmax, (*et)->nwdxov);
    if (recv_imcs(id, (*et)->e[0],
        (*et)->nhmax * (*et)->nwdxov * sizeof(complex),0) == -1)
    {
        perror("worker receive error 3 in init");
        exit(-1);
    }

    *dx2ov = alloc2float(*nx, *nz);
    if (recv_imcs(id, (*dx2ov)[0], (*nx) * (*nz) * sizeof(float),0) == -1)
    {
        perror("worker receive error 4 in init");
        exit(-1);
    }
}

int miget_next_frequency(
    int id, char *hostname, float *pw, complex **cpx)
{
    SimpleSend *send = (SimpleSend*)&buffer;
    NextReply *reply = (NextReply*)&buffer;

    if (tell_imcs(id, 1) == -1)
    {
        perror("tell error 1 in next");
        exit(-1);
    }

    send->routine = NEXT_FREQUENCY;
    strcpy(send->hostname, hostname);
    if (send_imcs(id, send, sizeof(SimpleSend),0) == -1)
    {
        perror("worker send error 1 in next");
        exit(-1);
    }
    if (recv_imcs(id, reply, sizeof(NextReply),0) == -1)
    {
        perror("worker receive error 1 in next");
        exit(-1);
    }
    if (reply->done)
        return 0;
    *pw = reply->pw;
    *cpx = reply->cpx;
    return 1;
}


void miget_report_work(
    int id, char *hostname, int num, int nx, int nz, float **q)
{
    ReportSend *send = (ReportSend*)&buffer;

    if (tell_imcs(id, 1) == -1)
```

```
    {
        perror("tell error 1 in report");
        exit(-1);
    }

    send->hdr.routine = REPORT_WORK;
    strcpy(send->hdr.hostname, hostname);
    send->num = num;
    if (send_imcs(id, send, sizeof(ReportSend),0) == -1)
    {
        perror("worker send error 1 in report");
        exit(-1);
    }

    if (send_imcs(id, q[0], nx*nz*sizeof(float),0) == -1)
    {
        perror("worker send error 2 in report");
        exit(-1);
    }
}
```

Appendix D
PVM/Godzilla Version

NOTE: The application code that this version has in common with the serial version has been omitted.

```
#include "par.h"
#include "extrap.h"
#include "manager.h"

char *webe = 0;                              /* this process name */
int nprocessors = 1;          /* number of processors to use */
int *pinums = 0;                     /* processor instance numbers */

char *sdoc =
"MIGET2 - MIGration via an Extrapolator Table of 2-D zero-offset data\n"
"\n"
"miget2 <infile >outfile efile= vfile= nt= nx= nz= [optional parameters]\n"
"\n"
"Required Parameters:\n"
"efile=                 name of file containing extrapolator table\n"
"vfile=                 name of file containing v(x,z)\n"
"nt=                    number of time samples\n"
"nx=                    number of inline samples (traces)\n"
"nz=                    number of depth samples\n"
"\n"
"Optional Parameters:\n"
"dt=1.0                 time sampling interval\n"
"dx=1.0                 inline sampling interval (trace spacing)\n"
"dz=1.0                 depth sampling interval\n"
"freqs=0,0,fNyq,fNyq    corner frequencies of trapezoidal bandpass filter\n"
"ntpad=nt               number of zero samples by which to pad time axis\n"
"verbose=0              =1 for diagnostics on standard error\n"
"port=1500              port number for communicating with workers\n"
"\n";

/* functions declared and used internally */
void makejw(int iwmin, int iwmax, int *jw);

/* variables declared and used internally */
int nx, nz, iw, iwmin, iwmax, *jw, mw;
float dw, **q, **dx2ov;
complex **cp;
eTable *et;
FILE *outfp = stdout;
static char *ProgName    = NULL;
int verbose;

int main(int argc, char **argv)
{
int i;
unsigned long saveport;
int status=0;
int Host_As_Worker=1;
```

```
static char
    *Command     = "miget2accd",
    *RemotesFile = NULL,
    *Path        = "${HOME}/bin";

    int nt,ntpad,nw,ntfft,it,ix,iz,iw0,iw1,iw2,iw3,nfreqs;
    unsigned long nlength;
    float dt,dx,dz,freqs[4],w,scale,fftscl,qscale,pscale,*p,**v;
    char *efile="",*vfile="";
    FILE *infp=stdin,*efp,*vfp;
        int myinum;                          /* my instance number */

/* do application initialization here */

                .
                .
                .
    /* compute frequency indices */
    makejw(iwmin,iwmax,jw);

        webe = (webe = rindex(argv[0], '/')) ? webe + 1 : argv[0];
        nprocessors = 11;
        pinums = (int*)malloc(nprocessors * sizeof(int));

        /* enroll in pvm */
        myinum = enroll(webe);

        /* create n instances of workers */
        for (i = 0; i < nprocessors; i++) {
                if ((pinums[i] = initiate("workers", (char*)0)) < 0) {
                        fputs("can't initiate calculator\n", stderr);
                        leave();
                        exit(1);
                }
        }

        go();

        /* kill processors we created */
        for (i = 0; i < nprocessors; i++) {
                terminate("workers", pinums[i]);
        }

        /* un-enroll from pvm */
        leave();
        exit(0);
}


#include "par.h"
#include "extrap.h"
#include "manager.h"
#include <signal.h>

extern char *webe = 0;                       /* this process name */
extern int nprocessors = 1;          /* number of processors to use */
extern int *pinums = 0;                      /* processor instance numbers */
```

```
int slaveinum;          /* processor id responding */

/*
Table to hold addresses of rpc's.
*/
static struct Routines
{
    void (*address)();
} routines[MAX_ROUTINES];

extern int nx, nz, iw, iwmin, iwmax, verbose, *jw, mw;
extern float dw, **q, **dx2ov;
extern complex **cp;
extern eTable *et;
extern FILE *outfp;


/*
Routine to send the initial data to the workers.
*/
static int reported = 0;
void manager_init(int id, SimpleSend *rcv)
{
AnyMsg buffer;
int status;
    InitReply *reply = (InitReply*)&buffer;
int i,j;

    if (verbose)
        fprintf(stderr, "%d calling miget_init()\n", slaveinum);

        initsend();
        putnint(&mw, 1);
        putnint(&nx, 1);
        putnint(&nz, 1);
        putnint(&(et->nhmax), 1);
        putnint(&(et->nwdxov), 1);
        putnfloat(&(et->dwdxov), 1);
        putnfloat(&(et->fwdxov), 1);
        putnfloat(&(et->dzodx), 1);
        if (snd("workers", slaveinum, 3)) {
                fprintf(stderr, "error sending to <%s,%d>\n",
                                "workers", slaveinum);
                leave();
                exit(1);
        }

        initsend();
        putnint(et->nh, et->nwdxov);
        if (snd("workers", slaveinum, 4)) {
                fprintf(stderr, "error sending to <%s,%d>\n",
                        "workers", slaveinum);
                leave();
                exit(1);
        }

        initsend();
```

```
        pvmPutNComplex ( et->nhmax * et->nwdxov, et->e[0]);
        if (snd("workers", slaveinum, 5)) {
                fprintf(stderr, "error sending to <%s,%d>\n",
                        "workers", slaveinum);
                leave();
                exit(1);
        }

        initsend();
        pvmPutNFloat(nx*nz,*dx2ov);
        if (snd("workers", slaveinum, 6)) {
                fprintf(stderr, "error sending to <%s,%d>\n",
                        "workers", slaveinum);
                leave();
                exit(1);
        }
}


/*
Routine to send new frequency tables.
*/
void manager_next_frequency(int id, SimpleSend* rcv)
{
int status;
int done;
float pw;
complex cpx[MAX_NX];

    if (verbose)
        fprintf(stderr, "%d calling miget_next_frequency()\n", slaveinum);

    if (iw > iwmax)
    {
        done = 1;
        initsend();
        putnint(&done, 1);
        if (snd("workers", slaveinum, 7)) {
                fprintf(stderr, "error sending to <%s,%d>\n",
                                "workers", slaveinum);
                leave();
                exit(1);
        }
    }
    else
    {
        int i, kw;

       if (verbose)
            fprintf(stderr, "%d/%d\n", iw-iwmin, iwmax-iwmin);

        done = 0;
        kw = jw[iw];
        pw = kw*dw;
        for (i = 0; i < nx; ++i) cpx[i] = cp[i][kw];

        initsend();
```

```
        putnint(&done, 1);
        putnfloat(&pw, 1);
        pvmPutNComplex(nx, cpx);
        if (snd("workers", slaveinum, 7)) {
                fprintf(stderr, "error sending to <%s,%d>\n",
                                "workers", slaveinum);
                leave();
                exit(1);
        }
        iw++;
    }
}


/*
Routine to report work.
*/
static float **results = NULL;
void manager_report_work(int id, ReportSend *report, int* alldone_ptr)
{
int status;
int ix, iz;
int num;

    if (verbose)
        fprintf(stderr, "%d calling miget_report_work()\n", slaveinum);

    if (results == NULL)
        results = alloc2float(nx, nz);


        if (rcv(8) == 8) {
                getnint(&num, 1);
                getnfloat(*results, nx*nz);
        } else {
                fprintf(stderr, "error receiving message type 8\n");
                leave();
                exit(1);
        }
    reported += num;

    for (ix = 0; ix < nx; ++ix)
        for (iz = 0; iz < nz; ++iz)
            q[iz][ix] += results[iz][ix];

        {
                float *qtemp=alloc1float(nz);
                for (ix=0; ix<nx; ++ix) {
                        for (iz=0; iz<nz; ++iz)
                                qtemp[iz] = q[iz][ix];
                        fwrite(qtemp,sizeof(float),nz,outfp);
                }
                free1float(qtemp);
        }

    if (reported == iwmax-iwmin+1)
    {
```

```
        if (results != NULL)
            free(results);
        *alldone_ptr = 1;
    } else *alldone_ptr = 0;
}



/*
Routine to implement communication loop.
*/
void go()
{
int status;
int alldone=0;
AnyMsg buffer;
        int myint=0;
        int i,j;


        fprintf(stderr,"wake up workers\n");
        /* Need this so workers can get manager's id */
        for (i = 0; i < nprocessors; i++) {
                initsend();
                if (snd("workers", pinums[i], 1)) {
                        fprintf(stderr, "error sending to <%s,%d>\n",
                                "workers", pinums[i]);
                        leave();
                        exit(1);
                }
        }

    routines[INIT].address = manager_init;
    routines[NEXT_FREQUENCY].address = manager_next_frequency;
    routines[REPORT_WORK].address = manager_report_work;
    while (!alldone)
    {
        int id;

        if (rcv(2) != 2) {
                fprintf(stderr, "error receiving message type 2\n");
                leave();
                exit(1);
        }

        rcvinfo((int*)0, (int*)0, (char*)0, &slaveinum);

        for (j = 0; j < nprocessors; j++)
                if (pinums[j] == slaveinum) break;

        if (j < nprocessors) {
                getnint(&myint, 1);
        } else {
                fprintf(stderr, "got response from unknown processor %d?\n",
                                slaveinum);
        }
```

```
        switch (myint)
        {
        case INIT:
               (*routines[INIT].address)(id, &buffer);
               break;
        case NEXT_FREQUENCY:
               (*routines[NEXT_FREQUENCY].address)(id, &buffer);
               break;
        case REPORT_WORK:
               (*routines[REPORT_WORK].address)(id, &buffer, &alldone);
               break;
        }
    }
}
```

```
#include "par.h"
#include "extrap.h"
#include "workers.h"

static AnyMsg buffer;

static float **zero(float**, int, int);

complex cpx_array[MAX_NX];
static    int mw, nx, nz, count = 0;
static    float **q, *wdxov, **dx2ov;
static    complex *cpx=cpx_array;
static    eTable *et;
extern int id;

char *webe;
int myinum;                              /* my instance number */
char mastername[32];   /* who sent us this tile */
int masterinum;
int cmd;

int main(int argc, char* argv[])
{
int verbose,id;
char *hn_ptr;

        webe = (webe = rindex(argv[0], '/')) ? webe + 1 : argv[0];
        myinum = enroll(webe);

    /* get initial tables to do problem */
    miget_init(id, hn_ptr, &mw, &nx, &nz, &et, &dx2ov);

    /* allocate workspace */
    q = zero(alloc2float(nx,nz), nx, nz);
    wdxov = alloc1float(nx);

    /* while manager has work to do ask for it and do it */
    if (verbose)
```

```
        fprintf(stderr, "Host (%s) started.  Entering work loop.\n", NULL);

    /* call user incremental routine until user signals to end */
    while (!usr_next(id, hn_ptr, verbose));

        fprintf(stderr, "exiting\n");
        leave();
        exit(1);
}

int usr_next(int id, char* hostname, int verbose)
{
int ix, iz;
float w;

        /* get next frequency and check for errors */
        if (!miget_next_frequency(id, hostname, &w, &cpx))
        {
            if (count)
                 miget_report_work(id, hostname, count, nx, nz, q);
            /* free et table */
            free2float(dx2ov);
            free1float(wdxov);
            free2float(q);
            return(1);
        }

        /* loop over depths z */
        for (iz = 0; iz < nz; ++iz)
        {
            /* accumulate migrated data */
            for (ix = 0; ix < nx; ++ix)
                q[iz][ix] += cpx[ix].r;

            /* make w*dx/v(x) */
            for (ix = 0; ix < nx; ++ix)
                wdxov[ix] = w * dx2ov[iz][ix];

            /* extrapolate wavefield */
            etextrap(et, nx, wdxov, cpx);
        }

        /* report work every mw frequencies */
        if ((count = ++count%mw) == 0)
        {
            miget_report_work(id, hostname, mw, nx, nz, q);
            zero(q, nx, nz);
        }
        return(0);
}

static float **zero(float **q, int nx, int nz)
{
    int i, j;
    for (j = 0; j < nz; ++j)
       for (i = 0; i < nx; ++i)
            q[j][i] = 0.0;
```

```
    return q;
}

void miget_init(
    int id, char *hostname,
    int *mw, int *nx, int *nz, eTable **et, float ***dx2ov)
{

        /* if got wake up signal from manager */
        if (rcv(1) == 1) {
                rcvinfo((int*)0, (int*)0, mastername, &masterinum);
                initsend();
                cmd = INIT;
                putnint(&cmd, 1);
                if (snd(mastername, masterinum, 2))
                        fprintf(stderr, "error sending command to manager\n");
        }

        if (rcv(3) == 3) {
                getnint(mw,1);
                getnint(nx,1);
                getnint(nz,1);
                *et = (eTable*)malloc(sizeof(eTable));
                getnint(&((*et)->nhmax),1);
                getnint(&((*et)->nwdxov),1);
                getnfloat(&((*et)->dwdxov),1);
                getnfloat(&((*et)->fwdxov),1);
                getnfloat(&((*et)->dzodx),1);
                (*et)->nh = alloc1int((*et)->nwdxov);
        }

        if (rcv(4) == 4)
                getnint((*et)->nh,(*et)->nwdxov);
        (*et)->e = alloc2complex((*et)->nhmax, (*et)->nwdxov);

        if (rcv(5) == 5)
                pvmGetNComplex((*et)->nhmax * (*et)->nwdxov,(*et)->e[0]);
        *dx2ov = alloc2float(*nx, *nz);

        if (rcv(6) == 6) pvmGetNFloat((*nx) * (*nz), **dx2ov);


int miget_next_frequency(
    int id, char *hostname, float *pw, complex **cpx)
{
int done;

        initsend();
        cmd = NEXT_FREQUENCY;
        putnint(&cmd, 2);
        if (snd(mastername, masterinum, 2))
                fprintf(stderr, "error sending command to manager\n");

        if (rcv(7) == 7) {
                getnint(&done, 1);
                if (done) return 0;
                getnfloat(pw, 1);
```

```
                pvmGetNComplex(nx,*cpx);
                return 1;
        }
        return 0;
}


void miget_report_work(
    int id, char *hostname, int num, int nx, int nz, float **q)
{
    ReportSend *send = (ReportSend*)&buffer;

        initsend();
        cmd = REPORT_WORK;
        putnint(&cmd, 1);
        if (snd(mastername, masterinum, 2))
                fprintf(stderr, "error sending command to manager\n");

        initsend();
        putnint(&num, 1);
        putnfloat(*q, nx*nz);
        if (snd(mastername, masterinum, 8))
                fprintf(stderr, "error sending report to manager\n");
}
```

Appendix E
NCS/Godzilla Version

NOTE: The application code that this version has in common with the serial version has been omitted as well as the stub code produced by the NIDL pre-compiler.

```
/* This version for demonstrating accumulation of frequencies */

#include "par.h"
#include "extrap.h"
#include "workers.h"
#include "MgrUtils.h"
#include "Net.h"
#include "Error.h"
#include "miget2acc.h"
#include <stdio.h>
#include <strings.h>
#include <sys/signal.h>

char *sdoc =
"MIGET2 - MIGration via an Extrapolator Table of 2-D zero-offset data\n"
"\n"
"miget2 <infile >outfile efile= vfile= nt= nx= nz= [optional parameters]\n"
"\n"
"Required Parameters:\n"
"efile=                 name of file containing extrapolator table\n"
"vfile=                 name of file containing v(x,z)\n"
"nt=                    number of time samples\n"
"nx=                    number of inline samples (traces)\n"
"nz=                    number of depth samples\n"
"\n"
"Optional Parameters:\n"
"dt=1.0                 time sampling interval\n"
"dx=1.0                 inline sampling interval (trace spacing)\n"
"dz=1.0                 depth sampling interval\n"
"freqs=0,0,fNyq,fNyq    corner frequencies of trapezoidal bandpass filter\n"
"ntpad=nt               number of zero samples by which to pad time axis\n"
"verbose=0              =1 for diagnostics on standard error\n"
"\n";

static char
    *Family      = "ip",
    *Command     = "miget2accd",
    *RemotesFile = NULL,
    *Path        = "${HOME}/bin",
    *ProgName    = "manager";

/* functions declared and used internally */
void makejw(int iwmin, int iwmax, int *jw);

/* variables declared and used internally */
int nx, nz, iw, iwmin, iwmax, verbose, *jw, mw;
```

```
float dw, **q, **dx2ov;
complex **cp;
eTable *et;
FILE *outfp = stdout;

main (int argc, char **argv)
{
        int nt,ntpad,nw,ntfft,it,ix,iz,iw0,iw1,iw2,iw3,nfreqs,kw;
        unsigned long nlength, port;
        float dt,dx,dz,freqs[4],w,scale,fftscl,qscale,pscale,*p,*temp;
        char *efile="",*vfile="",cmd[256],name[256];
        FILE *infp=stdin,*efp,*vfp;

        /* hook up getpar to handle the parameters */
               .
               .
               .
        /* compute frequency indices */
        makejw(iwmin,iwmax,jw);

        /* prepare for network execution */
        nlength = sizeof(name);
        initPort(
            Family, &miget2$if_spec, miget2$server_epv, name, nlength, &port);
        netInit(RemotesFile);
        sprintf(
            cmd,
            "PATH=${PATH}:%s;%s -v -n %s -p %d -f %s",
            Path, Command, name, port, Family);
        if (verbose)
            fprintf(stderr, "--- COMMAND ---\n%s\n", cmd);

        netGo(cmd);
            fprintf(stderr, "passed netGo\n");
        listenAtPort((unsigned long)1);

        free2float(q);
        free1int(jw);
        free2complex(cp);
        free2float(dx2ov);
}

#include <strings.h>
#include "MgrUtils.h"
#include "Error.h"

uuid_$t uuid_$nil;

int initPort(char *familyName,
    rpc_$if_spec_t *pIfSpec, rpc_$epv_t *pEntryPointVector,
    char *hostname, unsigned long nLength, unsigned long *pPort)
{
    socket_$addr_t sockaddr;
    unsigned long family, slength;
    status_$t status;
    char *pChar;
```

```
    /*
     * Get protocol family for communication.
     */
    if (familyName == NULL)
        familyName = "ip";
    family = socket_$family_from_name(familyName, strlen(familyName), &status);
    if (status.all != 0)
        error(
            FATAL,
            "Can't convert family name '", familyName, "' to family.",
            NULL
            );

    /*
     * Establish a socket to listen on.
     */
    rpc_$use_family_wk(family, pIfSpec, &sockaddr, &slength, &status);
    if (status.all != 0)
        error(
            FATAL,
            "Can't establish a socket using family '", familyName, "'.",
            NULL);

    /*
     * Register socket.
     */
    rpc_$register(pIfSpec, pEntryPointVector, &status);
    if (status.all != 0)
        error(
            FATAL,
            "Can't register socket.",
            NULL);

    /*
     * Convert socket address to its' name-port-family counterpart.
     */
    nLength--;  /* Save space for NULL termination. */
    rpc_$sockaddr_to_name(&sockaddr, slength,
        hostname, &nLength, pPort, &status);
    if (status.all != 0)
        error(
            FATAL,
            "Can't convert socket to its' name-port-family synonym.",
            NULL);
    hostname[nLength] = '\0';  /* Remember to NULL terminate */

    /*
     * Remember to trim family name from hostname.
     */
    if ((pChar = strchr(hostname, ':')) != NULL)
        strcpy(hostname, ++pChar);
}

void listenAtPort(unsigned long numberOfListeners)
{
    status_$t status;
```

```
    rpc_$listen(numberOfListeners, &status);
    if (status.all != 0)
        error(FATAL, "Can't listen on socket.\n");
}


#include "extrap.h"
#include "Error.h"
#include "par.h"
#include "miget2acc.h"
#include <stdio.h>
#include <strings.h>
#include <sys/param.h>

static char
    *ProgName,
    *HostName,
    *Family;
static int
    Verbose = 0;
static unsigned long
    Port;

static void
    commandLineParse(int, char*[]);

static void usage()
{
    fprintf(
        stderr,
        "usage: %s [-v] -n hostname -p port -f family\n",
        ProgName);
    exit(1);
}

int main(int argc, char* argv[])
{
    char hostname[32];
    int ix, iz, mw, nx, nz, elast, dlast, count = 0;
    float **q, *wdxov, **dx2ov;
    complex *cpx;
    eTable *et;
    miget2$etable_t PREet;
    miget2$real_table_t PREdx2ov;
    handle_t handle;
    status_$t status;

    ProgName = argv[0];



    /* get command line arguments */
    commandLineParse(argc, argv);

    /* get hostname */
    gethostname(hostname, sizeof(hostname));
```

```
/* create handle to communicate with manager */
handle = handleInit(HostName, Port, Family);

if (Verbose)
    fprintf(stderr, "Host (%s) started.  Entering work loop.\n", hostname);

/* get initial tables to do problem */
miget2$init(handle, hostname,
    &mw, &nx, &nz, &PREet, &dlast, PREdx2ov, &status);

/* allocate workspace */
q = alloc2float(nx,nz);
wdxov = alloc1float(nx);
cpx = alloc1complex(nx);

/* convert incomming data to correct data structures -- unmarshaling
   after unmarshaling (hey, that's efficient!) */
et = (eTable*)malloc(sizeof(eTable));
et->nhmax = PREet.nhmax;
et->nwdxov = PREet.nwdxov;
et->dwdxov = PREet.dwdxov;
et->fwdxov = PREet.fwdxov;
et->dzodx = PREet.dzodx;
et->nh = PREet.nh;
et->e = alloc2complex(et->nhmax, et->nwdxov);
bcopy(PREet.e, et->e[0], et->nhmax*et->nwdxov*sizeof(complex));
dx2ov = alloc2float(nx,nz);
bcopy(PREdx2ov, dx2ov[0], nz*nx*sizeof(float));

/* zero migrated data */
for (iz=0; iz<nz; ++iz)
    for (ix=0; ix<nx; ++ix)
        q[iz][ix] = 0.0;

/* while manager has work to do ask for it and do it */
while (TRUE)
{
    int clast;
    float w;

    /* get next frequency and check for errors */
    miget2$nextFrequency(handle, hostname, &w, &clast, cpx, &status);
    if (status.all == miget2acc$all_done)
    {
        miget2$reportWork(handle, hostname, count, nx*nz, q[0], &status);
        free1complex(cpx);
        free1float(wdxov);
        free2float(q);
        exit(0);
    }
    else if (status.all != status_$ok)
        error(FATAL, "Can't get work from Manager", NULL);

    /* loop over depths z */
    for (iz=0; iz<nz; ++iz)
    {
        /* accumulate migrated data */
```

```
            for (ix=0; ix<nx; ++ix)
                q[iz][ix] = q[iz][ix]+cpx[ix].r;

            /* make w*dx/v(x) */
            for (ix=0; ix<nx; ++ix)
                wdxov[ix] = w*dx2ov[iz][ix];

            /* extrapolate wavefield */
            etextrap(et,nx,wdxov,cpx);
        }


        /* report work every mw frequencies */
        if  ((count = ++count%mw) == 0 )

        {
            int ix, iz;

            miget2$reportWork(handle, hostname, mw, nx*nz, q[0], &status);

            /* zero migrated data */
            for (iz=0; iz<nz; ++iz)
               for (ix=0; ix<nx; ++ix)
                    q[iz][ix] = 0.0;
        }
    }
}

static void commandLineParse(int argc, char *argv[])
{
    unsigned long next = 1;
    int HostNameSet = 0, PortSet = 0, FamilySet = 0;

    while (next < argc)
    {
        char *flag = argv[next++];

        if (strcmp(flag, "-v")==0)
            Verbose = 1;
        else if (strcmp(flag, "-n")==0 && next<argc)
        {
            HostNameSet = 1;
            HostName = argv[next++];
        }
        else if (strcmp(flag, "-p")==0 && next<argc)
        {
            PortSet = 1;
            Port = strtol(argv[next++], NULL, 10);
        }
        else if (strcmp(flag, "-f")==0 && next<argc)
        {
            FamilySet = 1;
            Family = argv[next++];
        }
        else
        {
            error(WARNING, "Bad flag usage.", NULL);
```

```
            usage();
        }
    }

    if (!HostNameSet || !PortSet || !FamilySet)
        usage();
}
```

Appendix F
Linda Version

NOTE: The application code that this version has in common with the serial version has been omitted.

```
#include "linda.h"
#include "par.h"
#include "extrap.h"
#include "routines.h"
#include <time.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/resource.h>

char *sdoc =
"MIGET2 - MIGration via an Extrapolator Table of 2-D zero-offset data\n"
"\n"
"miget2 <infile >outfile efile= vfile= nt= nx= nz= [optional parameters]\n"
"\n"
"Required Parameters:\n"
"efile=                 name of file containing extrapolator table\n"
"vfile=                 name of file containing v(x,z)\n"
"nt=                    number of time samples\n"
"nx=                    number of inline samples (traces)\n"
"nz=                    number of depth samples\n"
"\n"
"Optional Parameters:\n"
"dt=1.0                 time sampling interval\n"
"dx=1.0                 inline sampling interval (trace spacing)\n"
"dz=1.0                 depth sampling interval\n"
"freqs=0,0,fNyq,fNyq    corner frequencies of trapezoidal bandpass filter\n"
"ntpad=nt               number of zero samples by which to pad time axis\n"
"verbose=0              =1 for diagnostics on standard error\n"
"port=1500              port number for communicating with workers\n"
"\n";

/* functions declared and used internally */
void makejw(int iwmin, int iwmax, int *jw);

/* variables declared and used internally */
int nx, nz, iw, iwmin, iwmax, *jw, mw;
float dw, **q, **dx2ov;
complex **cp;
eTable *et;
FILE *outfp = stdout;
int verbose;

int real_main()
{
int i;
unsigned long saveport;
int status=0;
int Host_As_Worker=1;
```

```
static char
    *Command     = "miget2accd",
    *RemotesFile = NULL,
    *Path        = "${HOME}/bin";

    int nt,ntpad,nw,ntfft,it,ix,iz,iw0,iw1,iw2,iw3,nfreqs;
    unsigned long nlength;
    float dt,dx,dz,freqs[4],w,scale,fftscl,qscale,pscale,*p,**v;
                c                          h                          a                          r
*efile="/scratch/jcrabtre/serial/et20.1001",*vfile="/scratch/jcrabtre/figures/goldvels.F";
    char *infile="/scratch/jcrabtre/figures/goldstks.F";
    FILE *infp,*efp,*vfp;
int num_procs=1;

/* do application initialization here */

                .
                .
                .
    /* compute frequency indices */
    makejw(iwmin,iwmax,jw);

    /* linda section */

        /* call manager_init to out initialization data */
        manager_init();

        /* call manager_init to out frequency data objects */
        manager_next_frequency();

        /* spawn workers and report manager */
        for (i=0; i<num_procs; i++) eval(workers());
        eval(manager_report_work());

        /* once all workers and report manager is thru w/ initialization
           data, in the data to clean up the tuple space */

        for (i=0; i<num_procs+1; i++) in("done init");
        in("manager_init init1", ? int, ? int, ? int, ? int, ? int, ? int,
               ? float, ? float, ? float);
        in("manager_init init2", ? int*:et->nwdxov);
        in("manager_init init3", ? et->e[0]:);
        in("manager_init init4", ? dx2ov[0]:);

        /* once all workers finish, signal report manager that there is
           no more data coming */

        for (i=0; i<num_procs; i++) in("worker done");
        out("manager_report_work results", (int) 1, q[0]:nx*nz);

        /* wait till report manager ends */
        in("reports done");

        /* ok to kill workers */
        out("hold until done");

}
```

```
/*
Routine to send the initial data to the workers.
*/
manager_init()
{
        if (verbose)
                fprintf(stderr, "manager_init() started\n" );

        out("manager_init init1", verbose, mw, nx, nz, et->nhmax, et->nwdxov,
                et->dwdxov, et->fwdxov, et->dzodx);
        out("manager_init init2", et->nh:et->nwdxov);
        out("manager_init init3", et->e[0]:et->nhmax * et->nwdxov);
        out("manager_init init4", dx2ov[0]:nx * nz);
}


/*
Routine to send new frequency tables.
*/
manager_next_frequency()
{
float pw;
complex cpx[MAX_NX];

    if (verbose)
        fprintf(stderr, "manager_next_frequency() started\n");

    while (1) {
        int i, kw;

        if (iw > iwmax) break;

        kw = jw[iw];
        pw = kw*dw;
        for (i = 0; i < nx; ++i) cpx[i] = cp[i][kw];

        out("manager_next_frequency data", iw-iwmin, pw, cpx:nx);
        iw++;
    }
}


#include "linda.h"
#include "par.h"
#include "extrap.h"
#include "routines.h"

static float **zero(float**, int, int);
static float **one(float**, int, int);
float **q, *wdxov, **dx2ov;

workers()
{
int verbose=0;
int ix, iz;
```

```
float pw;
int done=0;
complex cpx_array[MAX_NX];
eTable et;
int mw, nx, nz, count = 0;
complex *cpx=cpx_array;
int num=0;
int nxnz=0;

        /* get initialization data */
        rd("manager_init init1", ? verbose, ? mw, ? nx, ? nz, ? et.nhmax, ?
                et.nwdxov, ? et.dwdxov, ? et.fwdxov, ? et.dzodx);
        et.nh = alloc1int(et.nwdxov);
        et.e = alloc2complex(et.nhmax, et.nwdxov);
        dx2ov = alloc2float(nx, nz);
        nxnz = nx*nz;
        rd("manager_init init2", ? et.nh:et.nwdxov);
        rd("manager_init init3", ? et.e[0]:);
        rd("manager_init init4", ? dx2ov[0]:);

        /* signal manager that we are done with init data */
        out("done init");

        /* allocate workspace */
        q = zero(alloc2float(nx,nz), nx, nz);
        wdxov = alloc1float(nx);


        while (1) {

                /* if no data object found in TS, break */
                if(!inp("manager_next_frequency data", ? num, ? pw, ? cpx:nx))
                        break;

                /* loop over depths z */
                for (iz = 0; iz < nz; ++iz) {
                        /* accumulate migrated data */
                        for (ix = 0; ix < nx; ++ix)
                                q[iz][ix] += cpx[ix].r;

                        /* make w*dx/v(x) */
                        for (ix = 0; ix < nx; ++ix)
                                wdxov[ix] = pw * dx2ov[iz][ix];

                        /* extrapolate wavefield */
                        etextrap(&et, nx, wdxov, cpx);
                }

                /* report work every mw frequencies */
                if ((count = ++count%mw) == 0)
                {
                        out("manager_report_work results", (int) 0, q[0]:nx*nz);
                        zero(q, nx, nz);
                        in("got it");
                }
        } /* end while */
```

```
        /* NOTE: wouldnt work with nx*nz instead of nxnz as above */

        out("manager_report_work results", (int) 0, q[0]:nxnz);
        in("got it");  /* wait for signal from report manager */
        out("worker done");
        rd("hold until done");

        /* free et table */
        free2float(dx2ov);
        free1float(wdxov);
        free2float(q);
}

static float **zero(float **q, int nx, int nz)
{
    int i, j;
    for (j = 0; j < nz; ++j)
        for (i = 0; i < nx; ++i)
            q[j][i] = 0.0;
    return q;
}


#include "linda.h"
#include "par.h"
#include "extrap.h"
#include "routines.h"


/*
Routine to report work.
*/
manager_report_work()
{
int verbose,nx,nz;
int ix, iz;
complex cpx[MAX_NX];
int done=0;
float **results = NULL;
float **q;
FILE *outfp = stdout;
int nxnz;

    rd("manager_init init1", ? verbose, ? int, ? nx, ? nz, ? int, ? int,
               ? float, ? float, ? float);
    out("done init");

    nxnz = nx*nz;

    if (verbose)
        fprintf(stderr, "miget_report_work() started\n");

    q = alloc2float(nx,nz);

    /* zero migrated data */
    for (iz=0; iz<nz; ++iz)
        for (ix=0; ix<nx; ++ix)
```

```
            q[iz][ix] = 0.0;

    if (results == NULL)
        results = alloc2float(nx, nz);


    while (1) {
        in("manager_report_work results", ? done, ? results[0]:nxnz);
        out("got it");
        if (done) break;

        if (verbose) fprintf(stderr,"data reported\n");

        for (ix = 0; ix < nx; ++ix)
                for (iz = 0; iz < nz; ++iz)
                        q[iz][ix] += results[iz][ix];
        {
                float *qtemp=alloc1float(nz);
                for (ix=0; ix<nx; ++ix) {
                        for (iz=0; iz<nz; ++iz)
                                qtemp[iz] = q[iz][ix];
                        fwrite(qtemp,sizeof(float),nz,outfp);
                }
                free1float(qtemp);
        }

    } /* end while */

    if (results != NULL) free(results);
    out("reports done");
}
```