

NetFiles: A Novel Approach to Parallel Programming of Master/Worker Applications

Philip Chan* and David Abramson

School of Computer Science and Software Engineering
Monash University
900 Dandenong Road, Caulfield East
Victoria 3145 Australia
{pchan, david}@csse.monash.edu.au

ABSTRACT

We propose a new approach to parallel programming of master-worker applications using an abstraction for interprocess communication called NetFiles. The programmer writes the parallel application as a collection of sequential modules that communicate with each other by reading and/or writing files. When run on a sequential machine, the data is written to and read from conventional files. But, when run on a parallel platform, these file operations are actually implemented using message passing. Our approach makes it possible to develop a parallel program in two phases. In the first phase, the user is concerned with how to decompose the data and code, but without concern for the details of the parallel platform. In the second phase, the program can be configured to run on a parallel environment with little or no modification to the sequential code. We demonstrate the effectiveness of our approach on two parallel master/worker programs: parallel matrix multiplication and parallel genetic algorithms.

Keywords

Parallel programming, programming methodology, PVM, master/worker applications

1 INTRODUCTION

While there is a large diversity of different approaches and systems for parallel programming proposed over the years, the

ones that are most commonly used for writing real-world parallel applications involve using a conventional sequential language like C or Fortran in conjunction with:

- a set of library functions for message passing and synchronization, e.g., the Message Passing Interface (MPI, 1994), Message Passing Interface 2 (MPI-2, 1997), and Parallel Virtual Machine (Geist, Beguelin, Dongarra, Jiang, Manchek and Sunderam, 1994); and
- directives for data-parallelism like High-Performance Fortran (HPF Forum, 1997) or directives/API for shared-memory parallelism like OpenMP (Dagum and Menon, 1998).

Parallel programming using message-passing primitives requires the programmer to perform decomposition, placement and communication explicitly. For a non-specialist application programmer, this requires having to learn parallel computing concepts, interprocess communication and synchronization and a new set of functions. This is often also accompanied with learning about the details of a new computer system and its operating system. Despite the difficulty in using of message passing, it is the dominant implementation technique for parallel programs in use today.

* On leave from De La Salle University, Manila, Philippines, where he is a lecturer and a PhD student.
Email: ccspc@ccs.dlsu.edu.ph

HPF and OpenMP are standards for specifying parallelism in a sequential programming language. HPF is a set of directives for the Fortran language for specifying data-parallelism where communication is implicit. The data-parallel model is best suited for problems that involve repeated operations having a regular structure, for example, loops over arrays. The main drawback with this model, however, is its inability to express irregular communication patterns (Wilson, 1994).

On the other hand, OpenMP (Dagum and Menon, 1998) is a more recent standard that provides an Application Programming Interface (API) for shared memory parallel programming. It uses a combination of directives, library functions, and environment variables for specifying both data parallel and course-grain parallelism in sequential languages. Currently, OpenMP supports Fortran, C and C++.

1.1 Motivation

It is a common practice to develop parallel programs by modifying and restructuring existing sequential code. Typically, the parallel programmer has to decompose the program into parallel tasks, write the code to create the tasks and manage them, and insert the function calls for performing message-passing. It is during this development phase that various logical errors can be inadvertently introduced into the program. Correcting all these errors, however, is much more difficult now since the code is no longer a sequential program.

We present an approach to parallel programming that does not require the use of new functions for message passing, synchronization and task management within the program code. Instead of employing explicit message passing, we propose the use of files and file read/write operations as an abstraction to interprocess communication. This is advantageous in at least three ways: (a)

the programmer can start developing parallel programs using their existing (sequential) programming skills; (b) since the program codes are sequential programs running on a sequential platform, it is easier to perform testing and debugging; and (c) because messages are effectively stored in files, it is possible to inspect the contents of the messages during debugging using conventional tools.

Like message-passing, our approach requires the programmer to perform tasks such as partitioning the problem into parallel tasks and identifying what data needs to be communicated between them, etc. However, the advantage of our approach over that of message passing is that the programmer is using familiar programming constructs and functions even when in actuality a parallel program is being written.

2 The NetFiles Approach

A NetFile parallel application is developed as a collection of interrelated but independently executable programs. Communication between these programs is achieved by means of “special files” called *NetFiles*. Read and write operations on these special files are actually implemented as message passing. In essence, we use well-understood file I/O semantics as an abstraction mechanism for interprocess communication.

2.1 Background

The general idea of using files as a means of exchanging data is not new. One classical example is the use of pipes in Unix shell programming. Another example is in the context of a network application on top of a network file system where several applications concurrently read/write a shared file using techniques like file/record locking to ensure the file consistency.

Our approach introduces another application of this concept in the context of

parallel programming. NetFiles builds on an existing parallel computational tool called Nimrod. Nimrod (Abramson, Sosic, Giddy and Hall, 1995) is a system for performing parametric modeling on distributed computers. Parametric modeling involve executing the same application several times, with each execution using a different set of input parameter values. Enfuzion* is the commercial version of the research system Nimrod.

In Nimrod/Enfuzion, the user describes an experiment by declarative plan file which essentially specifies the parameter space of the parametric execution. The plan file also specifies the pre-processing and post-processing operations, which typically includes copying input and output data files that may be needed by the application. Using this approach, the user can write a conventional file based application, without regard to the underlying parallel platform. It can then be ported to a parallel machine in a subsequent operation. However, the major disadvantage of Nimrod is that unless the computation time exceeds the time to copy the files in and out of the remote system, the performance is poor. Netfiles builds on the Nimrod abstraction but is designed to support a much higher ratio of communication to computation.

2.2 NetFiles: Concept of Operation

A NetFile is a logical unidirectional communication link between a writer process and one or more reader processes. Since they accept the standard file I/O operations (i.e., `open()`, `read()`, `write()`, and `close()`), NetFiles appear like conventional data files to the programmer. The main difference is that a `write()` actually performs a message send while a `read()` performs a receive operation.

There are two types of NetFiles: *point-to-point* and *multicast*. A point-to-point NetFile

is established between a pair of processes, with one process being the writer and the other being the reader. A `write()` operation on an open NetFile essentially causes a data transfer from the writer to the reader of the NetFile. In a multicast NetFile, all readers of the NetFile will be able to read the data written by that single `write()`.

As with normal data files, the `open()` primitive specifies whether a NetFile is opened for reading or writing. When its writer has not opened a NetFile for writing, any process attempting to open it for reading will be blocked until it is opened for writing.

One essential difference between a NetFile `read()` and a message receive is that the `read()` allows the programmer to specify the length in bytes of the incoming data. This means that the data written to a NetFile do not have message boundaries. For example, data sent via a `write()` may actually be read by several "small" `read()` operations at the receiver's end. Similarly, all data sent through multiple `write()` operations may actually be received by a single `read()`. Hence, the complementing `read()` and `write()` operations on the same NetFile do not necessarily correspond one-to-one with each other. We believe that this allows more flexibility on the programmer, and at the same time, it makes a NetFile behave like regular data files.

2.3 Components of NetFile Master/Worker Applications

A parallel master/worker application is built as a collection of executable programs or modules. A module is a sequential program written entirely with conventional sequential programming constructs. Inter-module communication is done by means of reading and writing one or more NetFiles.

There are three basic module types: source, transform, and sink, and they are described as follows:

* <http://www.turbolinux.com>

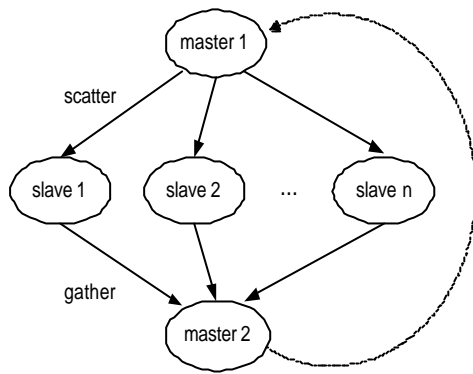


Figure 1 Master/Worker Application Structure

Source Module

This is a program that initiates the parallel application, usually by reading from input data files and writing to the first set of NetFiles.

Transform Module

This is a program that reads data from one or more NetFiles into local variables, performs some computations on the variable(s), and writes the result(s) to one or more NetFiles.

Sink Module

This is a program that is used to terminate the parallel application, by reading one or more NetFiles and performing post-processing activities like writing the results to data files.

A typical master/worker parallel program consists of one source module, one sink module, and one or more transform modules. It should be noted that it is also possible to have hybrid types of modules, for example, one that reads a single NetFile and creates a collection of output NetFiles.

Figure 1 illustrates the structure of a simple master/worker application. Solid arrows specify the data flow between the modules which can be implemented using NetFiles. Source module “master 1” creates a collection of NetFiles by scattering the work units among the worker modules. Each worker

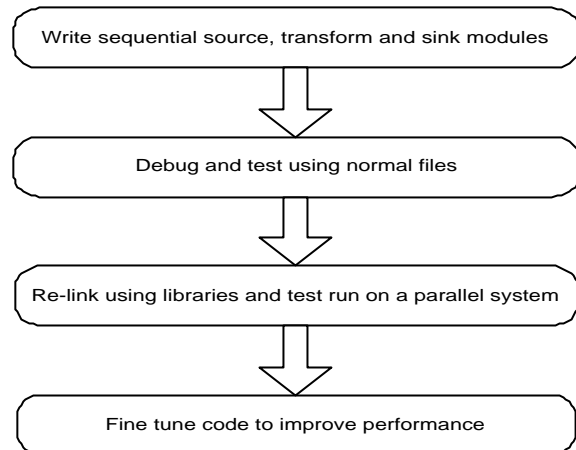


Figure 2 General Steps for Building NetFile Parallel Master/Worker Applications

“transforms” the data read via an input NetFile and creates results that are written to an output NetFile. The second master module reads (gathers) all the output NetFiles of the worker modules and performs some post-processing.

In the general case, we can have multiple scatter/gather phases, denoted by the dotted arrow in Figure 1. There are two ways to implement this: (a) the second master writes to a NetFile (or possibly a local data file) which will be read by a new “master 1” for the next phase; or (b) the second master performs another scatter operation to begin the next phase. In the second case, “master 2” actually is both a source and a sink module. With these building blocks, we can implement arbitrarily complex master/worker applications that have several scatter/gather phases.

2.4 Developing a NetFile-based Parallel Application

Figure 2 presents an overview of the methodology. Initially, the parallel programmer writes these modules and uses normal file I/O to implement the reading/writing of the NetFiles. Using normal file I/O primitives allows these modules to be tested and debugged like a sequential application, since they are amenable to sequential execution on a typical programming environment. This permits the programmer to correct, at this stage, any logical errors of the application *in isolation* of

```

parameter wno integer range from 1 to
10 step 1;
task rootstart
    execute master 1
endtask
task main
    copy root:input.$wno node:.
    node:execute slave $wno
    copy node:res.$wno root:res.$wno
end task
task rootfinish
    execute master 2
endtask

```

Figure 3 Sample plan file for a Master/Worker Application

issues that arise in parallel execution like synchronization and other timing dependent problems.

In implementing a master/worker application with a structure in Figure 1, the programmer may write a module[†] for the master and a module for the workers. To test these, the programmer may create a shell script to execute “master 1”, followed by a simple loop which executes each “worker” sequentially, then finally ending with the execution of “master 2”. During this development phase, the programmer is in fact already debugging a (virtually) parallel application.

When these modules are ready, Nimrod/Enfuzion may be used to stage its execution on a network of computers. Figure 3 illustrates how a plan file may be written for this. At the first stage, the master program is executed with parameter value 1 to indicate this is the first master (which will perform a scatter operation). It generates output for each worker, in this example, 10 files (i.e., `input.1` to `input.10`). Before running the worker, we copy the corresponding input file, and after execution, each worker generates

a result file (`res.1`, `res.2`, until `res.10`). This file is copied back to the root machine, which executes the second master to perform the final computation after gathering the results. While this demonstrates that it is possible to use Nimrod/Enfuzion for this purpose, our results (Section 4) show that we do not get very good performance results from it.

Our approach provides the next step in order to get better performance. These modules are re-linked using our new NetFiles library which implements the file read and write operations using message passing. This process creates executables ready for parallel execution on a distributed network environment. Since the modules do not invoke any functions that are necessary for parallel execution, an external runtime system must provide this functionality.

We have designed and implemented the NetFiles Application Execution Manager (NEM) that will stage the execution of the parallel application. This manager is responsible for creating the tasks from the modules and managing the communication between the tasks as they read and write NetFiles.

2.5 Configuring the Parallel Application

In order to properly stage the parallel application, the NEM requires configuration information of the parallel application. A configuration specifies two aspects of the parallel execution: the execution script and the NetFile dependencies between tasks. The execution script specifies the overall control structure of the application, e.g., what tasks will be created and when. This includes information such as: (a) which modules will be used to create the tasks, (b) the number of tasks per module, and (c) the command-line arguments for each task. The NetFile dependencies describe the properties of the NetFiles in terms of their type (point-to-point

[†] or two, one for “master 1” and another for “master 2”.

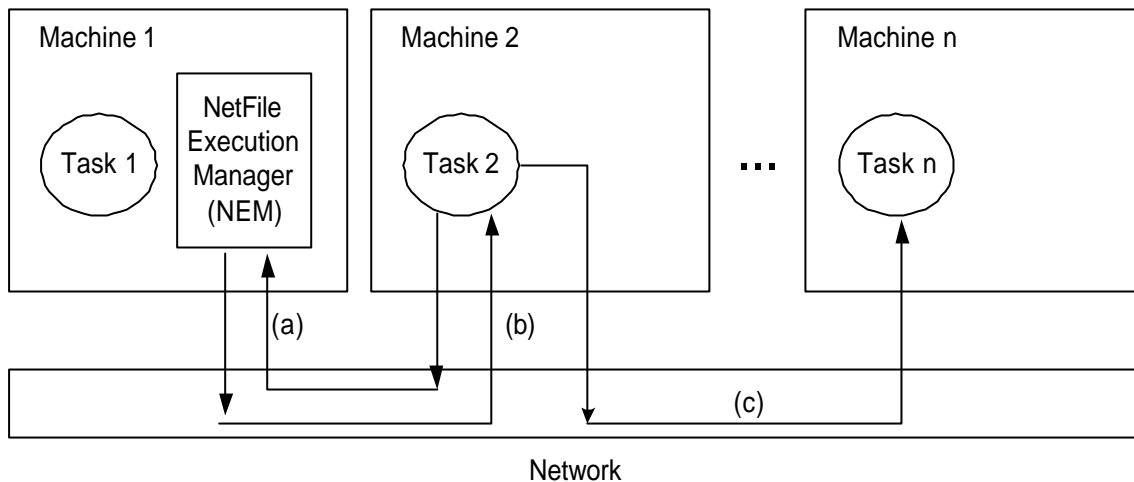


Figure 4 NetFile Execution Manager behavior during a NetFile Open() and Write()

or multicast) which affects how writes to the NetFile are performed and which task reads them in order to identify the reader when a `write()` operation is performed.

Tasks are running instances of modules which are created using a specific set of command-line arguments or parameters. In a NetFile-based parallel application, it is typical that several tasks are created from a single module with each task having a different set of parameter values. For example, a worker module may take as argument one parameter indicating its rank or relative position among a set of N workers. During execution, each of the N workers created will be assigned a different rank number as command-line argument. Similarly, in the example on Figure 1, we may have a single module to implement both “master 1” and “master 2”, which is distinguished by the command-line argument used to execute it.

3 NETFILES IMPLEMENTATION

We have implemented a prototype NetFile Execution Manager and the basic set of NetFile functions using PVM. The `read()` and `write()` operations for point-to-point NetFiles are implemented using

`pvm_recv()` and `pvm_send()`, respectively. The `write()` operation for broadcast/multicast NetFiles, on the other hand, is implemented using `pvm_mcast()`.

NetFile write operations are implemented in a straightforward manner, i.e., each `write()` actually performs a `pvm_pkbyte()` to pack the data into a send buffer and a `pvm_send()` to send the data to its intended reader. For a multicast type of NetFile, a `pvm_mcast()` is used instead of the `pvm_send()`.

As there are no message boundaries in NetFiles, we used a simple buffering mechanism within the `read()` operation. Essentially, this mechanism coalesces the data received due to multiple writes into one contiguous block stored in a local buffer. The `read()` operation copies only the specified number of bytes into the user buffer space.

Since it is the function of the NetFile Application Execution Manager to create the tasks using `pvm_spawn()`, it also maintains information about the association between each NetFile and its reader. Figure 4 demonstrates the role of the NEM. When a process performs an `open()`, it (a) sends a message containing the filename and the mode (whether

matmul.master.c	matmul.worker.c
<pre> main(int argc, char *argv[]) { obtain stage number from argv n = number of workers - 1 if (stage == SCATTER) { initialize matrices A and B : write entire matrix B into BFILE open AFILE00 to AFILEn distribute the rows of A to multiple AFILEs close AFILEs } else if (stage == GATHER) { open CFILE00 to CFILEn read rows and store into matrix C close files use matrix C } } </pre>	<pre> main(int argc, char *argv[]) { int fdA, fdB, n; n = get worker number from argv read entire BFILE into matrix B fdA = open AFILEn for reading fdB = open CFILEn for writing while (!eof(fdA)) { read a row from fdA perform row multiplication with B write resulting row to fdC } close(fdA); close(fdB); } </pre>

opening for read or write) to the NEM to obtain information about the file. Upon receipt of this message, the NEM checks its task table to identify the reader(s) of that NetFile. In the case of a point-to-point NetFile, the NEM (b) replies with the task ID of the reader. If the NetFile is a multicast, the NEM (b) replies with a vector containing the list of task IDs. Once the task receives this information, subsequent writes to that NetFile are just performed as (c) directed message send operations to the reader(s) of that NetFile.

In our current prototype, information about the configuration of the parallel application is stored within data structures of the NEM. For each application that we wish to stage, it is necessary to edit the source code of the NEM

to configure it properly. We are currently designing a language for specifying this configuration and a corresponding preprocessor.

4 EXPERIMENTAL RESULTS

We applied our methodology for implementing two master/worker applications: parallel matrix multiplication and parallel genetic algorithm. We conducted the runs on a Linux cluster using up to 9 dual-processor Pentium III running at 700 MHz, each with 128MB RAM. The machines are interconnected via a 10MB/sec. Ethernet. For each experiment, the master processes are always executed on a separate node, while all the workers are assigned on other processors.

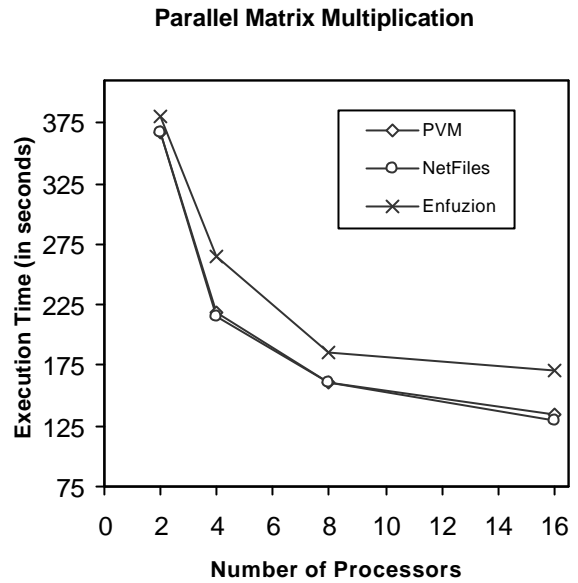
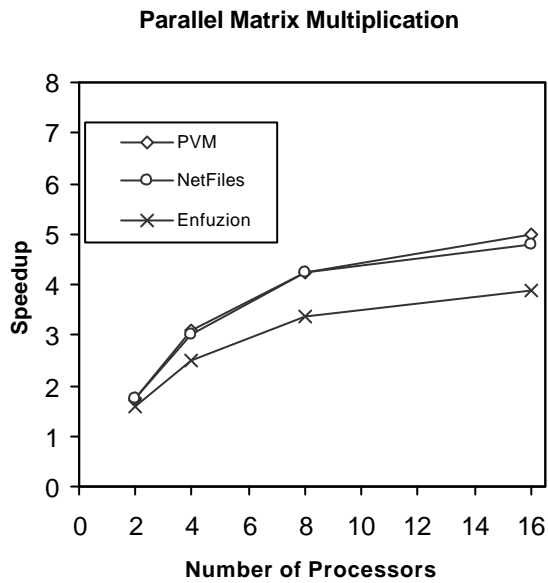


Figure 6 Speedup and Execution Time Result for the Parallel Matrix Multiplication Program

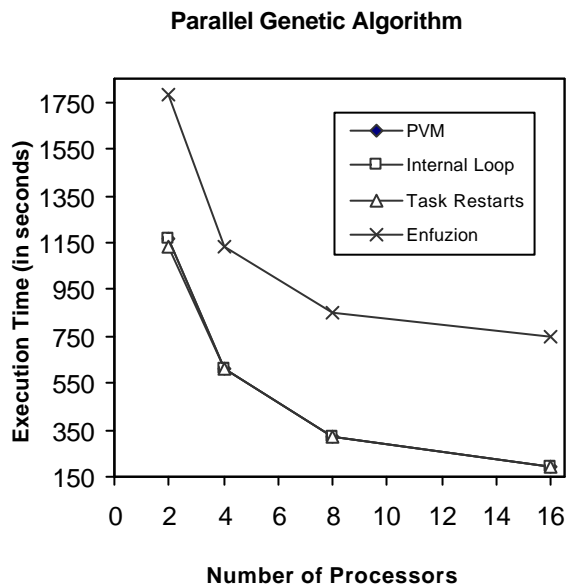
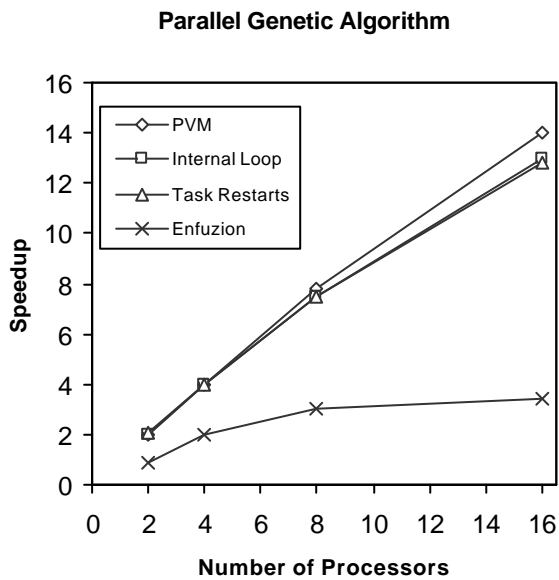


Figure 7 Speedup and Execution Time Results for the Parallel GA

4.1 Parallel Matrix Multiplication

As a first application to test our NetFiles methodology, we implemented a program for simple parallel matrix multiplication. The application was structured as in Figure 1 but with only one phase. The code outlines of the master and slave modules are presented in Figure 5.

The parallel matrix multiplication proceeds as follows: First, the entire matrix B is broadcast to all workers. Then the master distributes each row of matrix A in a round-robin fashion among all the workers. Each worker receives a set of rows of A and proceeds with the multiplication to obtain the resulting row and writes this row into an output

NetFile (CFILE). The program terminates when the master has completed reading the resulting matrix and stores it into a local data file.

Figure 6 presents the speedups and the execution times for the application using up to 16 processors for the workers. Each matrix is a 3000 x 3000 integer array. Understandably, this approach of parallelizing matrix multiplication yields a modest speedup of about 5 for 16 processors, due to the overhead in multicasting the entire B matrix. The results also show that although there is a high overhead when using Nimrod/Enfuzion, our approach allows the programmer to this tool to execute master/worker applications.

4.2 Parallel Genetic Algorithm

We also applied our methodology on a more complex parallel application: parallel genetic algorithm. We implemented this parallel GA for solving the travelling salesperson problem. We began by developing a sequential version of this program. This version accepted from the command-line various parameters such as the population size, number of generations, mutation rate, crossover rate, etc. We included functions for saving/loading the population and the GA parameters into a specified local file. This allowed us to save the population and resume the genetic algorithm at any point in time.

From this version, we incorporated functions for scatter and gather. The scatter essentially distributes the population into a user-specified number of subpopulations. Each subpopulation is stored in a file that is stored in the format used in the save function. The gather operation reads from a collection of files the subpopulations and resumes the GA with the combined population. With these functions in place, we are ready to test the essentially “parallel” GA by the following sequential execution:

1. run the GA program with scatter function creating a set of N files, where N is the number of subpopulations;
2. for each of the files, run a separate GA program and run for some number of generations and save the population to a new file;
3. run the GA program with the gather and scatter function, reading the outputs of the previous step and creating a new set of N files;
4. repeat step 2 and 3 for each of the N files;
5. on the last phase, skip the scatter function for step 3, in effect terminating the loop.

After we have tested that this sequence produced correct results, we re-linked the GA program with our NetFiles library and executed the parallel GA using our prototype NetFile Application Execution Manager (NEM) on our cluster.

Since this application involved multiple scatter/gather phases and this allowed us to consider two options in implementing the application. One option is where each phase triggers a restart of all the tasks: the master and the workers. This means that each task performed a read phase and then a write phase, terminating afterwards. A second option is to modify the modules so that each will run an internal loop. For example, the worker would read the input NetFile, performing the computation, writing the output NetFile, then return to the beginning of the loop. This allowed us to investigate the performance effects of restarting tasks. Figure 7 presents the results for the parallel genetic algorithm for solving TSP105 problem for 105 cities from TSPLIB[‡]. All runs involved 30 phases. Each phase involved creating 16 subpopulations, with each worker running the GA on each subpopulation for 1,500 generations.

[‡] <http://softlib.rice.edu/softlib/tsplib/>

MPI/PVM Operation	Equivalent in NetFiles
<pre>MPI_Comm_rank(); MPI_Finalize();</pre>	<pre>Not applicable since processes are not explicitly identified. Instead, NetFiles represents indirect communication between processes.</pre>
<pre>/* Broadcasting an array of int to all workers */ MPI_Pack(arr, numelem, MPI_INT, buf, SIZE, &pos, MPI_COMM_WORLD); MPI_Bcast(buf, size, MPI_PACKED, root_tid, MPI_COMM_WORLD);</pre>	<pre>/* configure b-netfile as broadcast type */ fd = open("b-netfile", O_WRONLY ...); write(fd, arr, sizeof(int) * numelem); /* close the file when done */</pre>
<pre>/* sending a message to task with dest as TID */ MPI_Pack(&var1, 1, MPI_INT, buf, SIZE, &pos, MPI_COMM_WORLD); MPI_Pack(&var2, 10, MPI_FLOAT, buf, SIZE, &pos, MPI_COMM_WORLD); MPI_Send(buf, pos, MPI_PACKED, dest, tag, MPI_COMM_WORLD);</pre>	<pre>/* filename is point-to-point */ fd = open(filename, O_WRONLY ...); write(fd, &var1, sizeof(int)); write(fd, &var2, sizeof(float) * 10);</pre>
<pre>/* receiving a message from task with source as TID */ MPI_Recv(buf, SIZE, MPI_PACKED, source, tag, MPI_COMM_WORLD, &st); MPI_Unpack(buf, SIZE, &pos, &var1, 1, MPI_INT, MPI_COMM_WORLD); MPI_Unpack(buf, SIZE, &pos, &var2, 10, MPI_FLOAT, MPI_COMM_WORLD);</pre>	<pre>/* filename is point-to-point */ fd = open(filename, O_RDONLY); read(fd, &var1, sizeof(int)); read(fd, &var2, sizeof(float) * 10);</pre>

Table 1. Some Code Comparisons between MPI Functions and NetFiles

From these results in Figure 7, we have two observations: First, the overhead in restarting the master and workers is minimal by comparison to the second method where we employ an internal loop. This is due to the fact that we restart the slaves as soon as the previous instance has terminated. This means that the master does not get delayed while waiting for the slaves to be restarted.

Second, notice that the PVM version performs noticeably better than the NetFile version. This is due to the fact that in the PVM version, we explicitly distributed the global population using a temporary array before packing each subpopulation into the send buffer. This meant that each subpopulation was sent via a single `pvm_send()`. In the NetFile version, on the other hand, we distributed the global population by writing each chromosome in a round-robin fashion among the NetFiles. Since each `write()` is actually implemented by a `pvm_send()`, the NetFile version is less efficient although the program code is more intuitive.

5 EVALUATION

Table 1 presents code equivalences between MPI and NetFiles in terms of commonly used patterns of basic communication operations. It should be noted that with NetFiles, the communication structure is established by the configuration of the application which is external to the program code. For each point-to-point NetFile, this configuration identifies the sender and the receiver. For a broadcast NetFile, this configuration specifies its sender and the identities of all receivers.

Secondly, instead of using a separate buffer for packing as in the MPI code fragments, the programmer simply performs the write operation when using NetFiles (as with ordinary data files). While our current version implements a `pvm_send()` for each `write()`, in future versions we will consider using some form of buffering within the `write()` operation.

5.1 Source Level Differences and Programmability

We compared the source codes of the PVM version and the NetFile version for both applications and we observe that:

- The number of source statements for reading and writing compared to sending and receiving are almost equal. While the NetFile version did not contain any code to deal with task IDs, it required code for opening and closing the NetFile. This included code for formatting the string variable to the proper filename.
- The PVM version had about 50 (total of 3 functions) additional source statements for dealing with the spawning of workers and the initialization of the machine configuration since we wanted to control where we would like to spawn the workers.

Our experience shows that it is much simpler to work on the sequential version of the applications by using files as an abstraction and then eventually migrate the code (by re-linking with the NetFile library) into a parallel version. One reason is that we were able to test the file version sequentially and control the execution of the master and the individual workers. This allows us to identify and correct any program errors much quickly.

6 RELATED WORK

Our work with NetFiles has been inspired and influenced in some ways by the following:

- Nimrod/Enfuzion (Abramson, et al., 1995), a computational tool which simplified the development of parametric modeling applications since it does not require reprogramming of the application code;
- Unix sockets (Stevens, 1998) which essentially provides the same read/write operation to that for data files. However, using sockets within programs require a much programming effort from setting up socket connections (for TCP) to use of

multiplexing to process data from multiple socket connections.

It is interesting to note of one approach that is most similar to our work is Virtual Shared Files (Konovalov, Samofalov, and Scharf, 1999). While this project shares the same idea of employing files as an abstraction for communication, the design and implementation details are quite different. Their approach extends the notion of a file and file space by using a special file-naming scheme, and also specifies constraints like requiring all operations to be applied to a file as a whole, and limiting file data and memory allocation to files that are owned by the task.

7 CONCLUSIONS AND FUTURE WORK

We have presented a novel parallel programming methodology for writing master/worker applications. We propose the use of NetFiles as an abstraction mechanism to message passing. Our approach is unique because the programmer can develop a parallel application by writing it entirely in a familiar sequential programming environment. This method facilitates parallel program development in two phases: in the first phase, the programmer focuses on decomposition and parallelisation issues in a familiar environment; in the second phase, the programmer deals with low-level issues, e.g., how communication is specified, etc.

We are currently working on the following:

- a configuration language to specify the script and the NetFile / task dependencies;
- a graphical notation which is as expressive as this configuration language to make it easier to stage and configure NetFile applications; and
- how this approach may be extended for parallel programming in general which involves peer-to-peer (or worker-to-worker) interprocess communication.

REFERENCES

- Abramson, D., Susic R., Giddy J., and Hall B. (1995): "Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations," In: *Proc. of the 4th IEEE Symposium on High Performance Distributed Computing*.
- Dagum, L. and Menon R. (1998): "OpenMP: An Industry-Standard API for Shared Memory Programming," *IEEE Computational Science and Engineering*, 5(1):46-55.
- Geist, A., Beguelin A., Dongarra J., Jiang W., Manchek R., and Sunderam V. (1994): *PVM Parallel Virtual Machine: A User's Guide and Tutorial for Network Parallel Computing*, MIT Press.
- High-Performance Fortran Forum (1997): *High-Performance Fortran Language Specification*, Version 2.0, January 31.
- Kononov, A., Samofalov V., and Scharf S. (1999): "Virtual Shared Files: Towards User-Friendly Inter-Process Communication," In: *Proc of the 5th International Conference on Parallel Computing Technologies (PaCT-99)*, St. Petersburg, Russia, September 6-10.
- Loveman, D. (1993): "High-Performance Fortran," *IEEE Parallel and Distributed Technology*, 1(1).
- Message Passing Interface Forum (1994): "MPI: A Message-Passing Interface Standard."
- Message Passing Interface Forum (1997): "MPI-2: Extensions to the Message-Passing Interface Standard."
- Stevens, W. R. (1998): *Unix Network Programming: Networking APIs: Sockets and XTI*, Vol. 1., 2nd Edition, Prentice Hall.
- Wilson, G. (1994): *Practical Parallel Programming*, MIT Press, Cambridge, MA.