

Netfiles: An Enhanced Stream-based Communication Mechanism

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. *Netfiles* is an alternative API for message passing on distributed memory machines that is based on pipes. It provides enhanced capabilities such as broadcasts and gather operations. Because *Netfiles* overload conventional file I/O operations, parallel programs can be developed and tested on a file system before execution on a parallel machine. *Netfiles* is part of a parallel programming system called FAbriC. This paper also presents the design and implementation of the FAbriC architecture and demonstrate the effectiveness of this approach by means of two parallel applications: a parallel shallow water model application and parallel Jacobi method.

1 Introduction

Stream-based communication dates back to the early days of Unix. It encompasses mechanisms such as pipes, named pipes and sockets with TCP/IP. This model of communication is well-understood and has been successful.

We present an enhanced stream communication mechanism called *Netfiles*. The *Netfile* abstraction overloads the file and file I/O primitives with message passing functionality specifically for parallel programming, with minimal deviation from semantics of conventional file I/O. This overloading allows a parallel program to be written and tested using conventional files prior to its execution on a distributed-memory parallel machine.

2 *Netfiles*: Enhanced Pipes for IPC

The basic communication instance between two tasks is the reading and writing of a common *Netfile*, identified by a filename. This filename is a user-specified string that is task/node independent, allowing dynamic reconfiguration. The sender opens the *Netfile* in WRITE mode, while the receiver opens in READ mode. When both ends are open, the *Netfile* effectively becomes a channel for communication, like a Unix pipe. A `write()` behaves like a `send()`, and a `read()` behaves

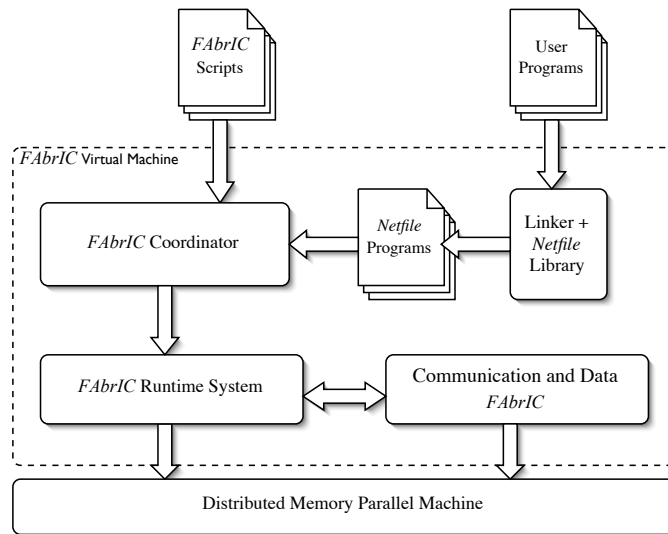


Fig. 1. System Architecture of FAbriC

like a `receive()`. Non-blocking *Netfile* semantics is also possible, e.g., writing to a *Netfile* completing even before the start of reading. This allows communication and computation to overlap, potentially improving performance.

No implicit type information is encoded within the stream. Thus, the data written through a *Netfile* do not have message boundaries, enhancing programming flexibility. For example, data sent via a `write()` may actually be read by several consecutive “small” `read()` operations.

Netfiles support collective operations, namely: broadcast and gather. A single-writer, multiple-reader *Netfile* behaves as a broadcast, closely resembling multiple reads on a single file. A single-reader *Netfile* with multiple writers appending data can be used to represent a gather.

Netfiles form part of a proposed parallel programming system called FAbriC. Here, a parallel program is first written to use files and file I/O as for communication. No explicit `send()` or `receive()` are used in the program. Once tested, the program may be relinked with our *Netfile*-library for parallel execution on a distributed memory machine.

3 FAbriC System Architecture and its Implementation

Fig. 1 presents the key components of FAbriC. User programs are relinked with the *Netfile* library to create FAbriC-ready applications. Execution is specified via an external run script written in the FAbriC coordination language. The FAbriC Coordinator interprets this and starts tasks through the FAbriC Runtime System.

```

task sm1 (nslaves)
    program "exe/shallow-master" args "$(nslaves) 0"
endtask
task sm2 (nslaves, maxiters)
    program "exe/shallow-worker" args "-1 $(maxiters) $(nslaves)"
endtask
task sw (slave_id, maxiters)
    program "exe/shallow-worker" args "$(slave_id) $(maxiters)"
endtask
par do
    execute sm1 (16)
    par i from 1 to 16 do
        execute sw (i, 100)
    enddo
    execute sm2 (16, 100)
enddo

```

Fig. 2. Sample Run Script

The Communication and Data FAbriC implements the FAbriC Space, a collective abstraction for both stream-like and persistent *Netfiles*. We describe below our prototype implementation of this FAbriC virtual machine.

3.1 Communication FAbriC: *Netfile* Lookup Service

The *Netfile* lookup service matches the reader and the writer of each *Netfile*. Each unmatched `open()` is recorded as pending. When the matching `open()` arrives, this pending request is used to match the pair. Once matched, communication between them takes place via a TCP/IP connection. For efficiency, connections are reused for different *Netfiles* between the same task pair.

Collective communications are implemented on top of the point-to-point *Netfiles*. For efficiency, broadcasts are performed by establishing tree-structured network of *Netfile* connections between tasks. A gather may use the same tree where the leaf tasks initiate the sending, with all data coalesced at the root task.

3.2 FAbriC Coordinator and Runtime System

We designed a coordination language with constructs for sequential (`seq`) and parallel (`par`) execution. Fig. 2 is a script that specifies parallel execution of two masters (scatter and gather) and 16 workers. The `task-endtask` pair specifies details of the task, i.e., the program executable name and how arguments are translated into command-line parameters. The `execute` statement causes a task to be started with a specified set of arguments.

Execution of a FAbriC application involves running the FAbriC Coordinator with two input files: (a) the run script; and (b) the machine configuration, a

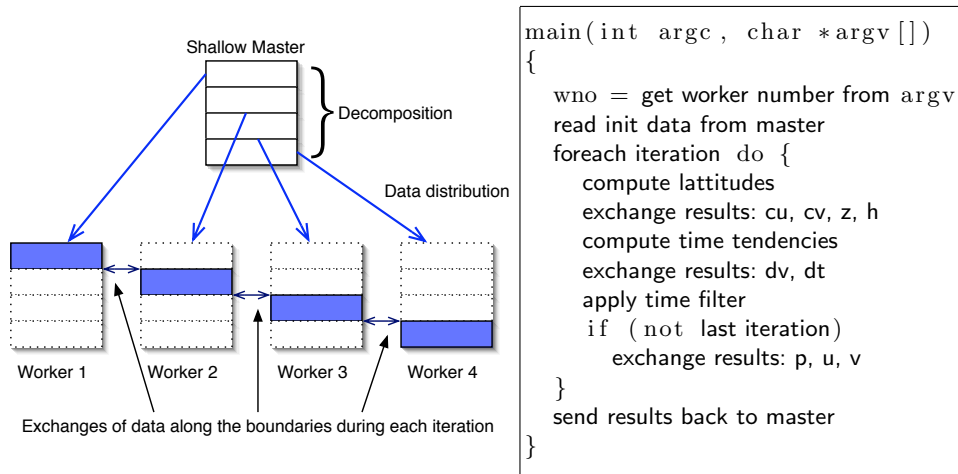


Fig. 3. Data Distribution and Data Exchanges in Parallel Shallow Model (4 workers)

list of host names to use. This coordinator starts tasks via the FABrIC Runtime System, which is a collection of execution daemons, one on each host.

To initiate a task, the coordinator sends a `task_start` request to the daemon on a selected host. This request contains the logical TID used for *Netfile*-lookup for matching tasks. The daemon starts the task and passes the logical TID to the task via an environment variable.

Upon task startup, the daemon replies with the process ID (`pid`). The pair (process ID, host logical ID) is used by the coordinator to uniquely identify a task in the application. When the task completes, its corresponding execution daemon notifies the coordinator by a `task_complete` message. Once all tasks have completed, the coordinator terminates.

4 Case Study 1: Parallel Shallow Water Equations

To evaluate our prototype, we selected a numerical model involving shallow water equations [7, 10]. It employs both master-worker and worker-worker communication where data along the boundaries (Fig. 3) are exchanged between adjacent workers during each iteration. At the end of the iterations, workers send their results to the master. We used the MPI code of Abramson, Dix & Whiting [1].

4.1 *Netfiles* Version of Parallel Shallow

With the FABrIC approach, we first built a parallel shallow application using standard file I/O primitives for interprocess communication. To be descriptive of what the files represented, the variable names of the data structures were

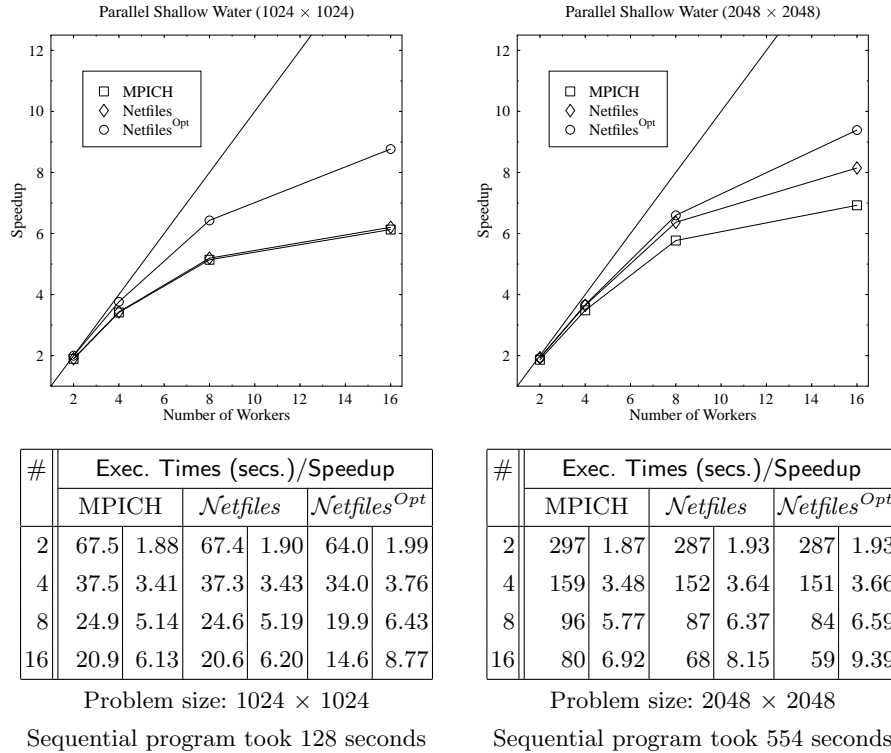


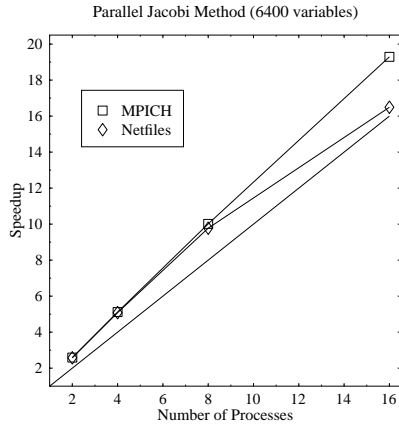
Fig. 4. Parallel shallow water model execution times and speedups

used as filenames. For data exchanges involving neighboring elements, we used the row number of the data elements in the filename.

This version was first tested on a local file system using background task execution. Wrapper functions for `open()` and `close()` which provided synchronisation were used. After testing, we relinked the code with our *Netfile* library. The *Netfile* program is essentially identical to the MPI code in the manner of data and task decomposition and distribution of data between tasks.

4.2 Experimental Results

We considered two grid size configurations: 1024×1024 and 2048×2048. For each run, we executed the model for 100 iterations. And the speedups were based on the execution time of the MPI program with 1 worker. Fig. 4 gives the results for two versions of the *Netfile* parallel shallow programs. In the first one, *Netfiles* were opened and closed for each message. For small problem sizes, the overhead of requests-replies with the *Netfile*-lookup service dominated the communication cost. In the second version (identified as *Netfiles^{Opt}*), we “fine-tuned” the program code so that open *Netfiles* for worker-to-worker communications are not



#	Exec. Times (secs.)/Speedup			
	MPICH		<i>Netfiles</i>	
2	296.82	2.58	298.28	2.57
4	150.15	5.11	151.06	5.08
8	76.70	10.01	78.58	9.77
16	39.81	19.29	46.57	16.49

Problem size: 6400 × 6400 matrix
 Sequential program took 768 seconds

Fig. 5. Parallel Jacobi execution times and speedups

closed but are reused. This minor optimisation resulted in the best performance since it reduced the number of lookups.

These results show that with our prototype, *Netfiles* applications can perform as well as their MPI equivalents. And in large problem sizes, our *Netfiles* program outperformed its MPI equivalent. We suspect that performance gain is due to our method of reusing the TCP connections between tasks.

Experiments were run on a Linux cluster of Intel Xeon processors (each with 3 GHz and 1GB RAM) connected by a Gigabit Ethernet switch. All programs were compiled using gcc version 3.3.4. For MPI, we used MPICH (1.2.6) from Argonne National Laboratories and Mississippi State University.

5 Case Study 2: Parallel Jacobi Method

We also built a parallel Jacobi iterative method for solving linear equations. This code is characterized by collective communication at the end of each iteration. At the start of the application, the matrix that represents the coefficients of the linear equations is distributed to all other tasks. During each iteration, all tasks perform the Jacobi method and obtain its segment of the solution vector. This solution vector is gathered and broadcast to the other tasks.

Fig. 5 shows the results on a problem size of 6400 variables. The linear equations are stored in a 6400^2 matrix of `doubles` and convergence was at 960 iterations. The superlinear speedup is most likely due to the cache and virtual memory issues. It is clear that matching the performance of the MPI version requires further optimisation in our collective communications.

We used an IBM POWER5 cluster (4 cpus/node, minimum of 8 GB RAM, Suse Linux) connected via Myrinet. All programs were compiled using IBM `x1c` 64-bit compiler and MPICH (version 1.2.6).

6 Related Work

The idea of using files as a proxy for message passing was inspired from the Nimrod project [2]. Nimrod is a software tool which simplifies the development of parametric modelling applications. During an experiment, data files needed by jobs are supplied via file transfers, effectively a form of interprocess communication. One way to reduce transfer times is to use sockets [9] with TCP connections but the added programming complexity can be discouraging. The *Netfile* programming interface is simpler since it abstracts connection details (IP addresses/ports) and requires only an `open()` to setup communication.

The idea of extending standard file I/O for channel communication was probably first proposed by Seevers et al. [8] for Dataparallel C. We were unable to find any subsequent works describing its development and implementation. *FABrIC/Netfiles*, on the other hand, is an implementation and supports collective communication like broadcast and gather operations.

Similarly, Virtual Shared Files [5] was also proposed to employ a file space model for IPC. The key difference with *Netfiles* is that we overload standard file I/O primitives instead of introducing new ones. In addition, a *Netfile* can behave in two modes: point-to-point stream and a distributed shared object. This shared object semantics for *Netfiles* is currently being investigated.

Message passing libraries (PVM [4] and MPI [6] implementations) are widely employed tools for parallel programming on a distributed memory system like a network of workstations. *FABrIC/Netfiles* is essentially message passing within file I/O primitives. This presents a benefit that applications can be built and tested on a conventional file system, prior to parallel execution.

Our earlier *Netfiles* paper [3] focussed on master-worker parallel programming with no worker-worker communication. And our prototype library and runtime system was implemented on top of PVM. In this paper, we present the *FABrIC* system architecture and its implementation on top of sockets. We also demonstrate the effectiveness of *FABrIC/Netfiles* for interworker communication.

7 Conclusions

We have presented the *FABrIC* parallel programming approach using an abstraction called the *Netfile*. While a basic *Netfile* is essentially a pipe that allows IPC across machines on a network, its enhanced form includes multicast capability. It enables a parallel application to be written in two phases: (a) write and test on a familiar sequential environment using files and file I/O as a means for inter-task communication; and (b) obtain the parallel version by relinking with the *Netfile* library.

FABrIC is evaluated using two case studies: a parallel shallow water model; and parallel Jacobi method. Results show that with some minor fine-tuning, the

FAbriC parallel shallow water application consistently performed better than the equivalent MPICH program. However, our *Netfile* implementation of collective communications will require further optimisation as evident from results in the parallel Jacobi iteration. Overall, this demonstrates viability of this approach and encourages further investigation.

Acknowledgements

We thank the Victorian Partnership for Advanced Computing for the use of their IBM POWER5 cluster.

References

1. David Abramson, Martin Dix, and Paul Whiting. A Study of the Shallow Water Equations on Various Parallel Architectures. In *Proc. of the 14th Australian Computer Science Conference*, pages 6:1–6:12, Sydney, Australia, 1991.
2. David A. Abramson, Rok Susic, Jonathan Giddy, and B. Hall. Nimrod: A Tool for Performing Parameterised Simulations using Distributed Workstations. In *Proc. of the 4th IEEE Symposium on High Performance Distributed Computing*, Virginia, August 1995. IEEE Press.
3. Philip Chan and David Abramson. NetFiles: A Novel Approach to Parallel Programming of Master/Worker Applications. In *Proc. of the 5th International Conference and Exhibition on High-Performance Computing in the Asia-Pacific Region (HPCAsia 2001)*, Queensland, Australia, September 2001.
4. Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM Parallel Virtual Machine: A User's Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.
5. Alexandr Konovalov, Victor Samofalov, and Sergei Scharf. Virtual Shared Files: Towards User-Friendly Inter-Process Communications. In *Proc. of the 5th International Conference on Parallel Computing Technologies (PaCT-99)*, St. Petersburg, Russia, September 1999.
6. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, May 1994.
7. Robert Sadourny. The Dynamics of Finite-Difference Models of the Shallow Water Equations. *Journal of Atmospheric Sciences*, 32(4):680–689, 1975.
8. Bradley K. SeEVERS, Michael J. Quinn, and Philip J. Hatcher. A Parallel Programming Environment Supporting Multiple Data-Parallel Modules. In *SIGPLAN Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Multiprocessors*, pages 44–47, Boulder, Colorado, 1992.
9. W. Richard Stevens. *Unix Network Programming: Networking APIs: Sockets and XTI*, volume 1. Prentice-Hall PTR, 2nd edition, 1998.
10. Warren M. Washington and Claire L. Parkinson. *An Introduction to Three-Dimensional Climate Modeling*. Oxford University Press, 1986.