# MPI-HPF COMMUNICATION TECHNIQUES

---

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

---

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

---

By

Hrishikesh Pradeep Divate

May, 2000

# MPI-HPF COMMUNICATION TECHNIQUES

Hrishikesh Pradeep Divate

APPROVED:

Dr. S. Lennart Johnsson
Department of Computer Science, UH

Dr. Jaspal Subhlok
Department of Computer Science, UH

Dr. B. Montgomery Pettitt
Department of Chemistry, UH

Dean, College of Natural Sciences and Mathematics

# Acknowledgements

I am very grateful to my advisor, Dr. Lennart Johnsson for giving me the opportunity to work with him and for having taught me so much during the past two years. I would also like to thank the members of my thesis committee, Dr. Pettitt and Dr. Subhlok for helpful discussions and for reviewing this document.

Many thanks to Charlie Hu for guiding me in the initial stages of my work. I am also very grateful to Zdenko Tomasic for being an excellent person to work with and for providing valuable tips and suggestions during the course of this work. I would like to thank Olle for helping me understand various aspects of Globus and the IBM SP2. I thank past and present members of our research group for creating a congenial atmosphere to work in. Thanks to Heidi Lorenz-Wirzba at Caltech and the National Center for Supercomputing Applications (NCSA) support staff, I was able to learn a lot about the parallel computers installed at the two sites.

I consider myself extremely fortunate to have made many good friends at UH and would specially like to thank Manish, Rishad, Kaushik, Mangesh, Damal, Anil, Sumanth, Rishu and Gunjan for helping me out and for all the good times we had. Many thanks also go to Subin George for interesting discussions.

Finally, I would like to thank my parents and my brother for their love, constant support and encouragement in all my endeavors.

Support from the Technical Management Concepts Inc. grant TMC96-5835-0029-02, computing time at the San Diego Supercomputing Center and the Center for Advanced Computing Research (CACR) through NPACI, NCSA and Argonne National Lab is hereby gratefully acknowledged.

# MPI-HPF COMMUNICATION TECHNIQUES

---

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

---

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

---

By

Hrishikesh Pradeep Divate

May, 2000

# Abstract

This thesis presents high-performance, portable and efficient techniques for communication between parallel programs written using the Message Passing Interface (MPI) libraries and High Performance Fortran (HPF).

Our techniques, communication with UNIX sockets, communication with MPI and communication using shared memory, have been tested and run on a distributed memory machine (IBM SP2), shared memory machines (HP Exemplar, SGI Origin 2000) and the Globus metacomputing toolkit. The characteristics of the techniques on each platform are discussed.

We believe our techniques will be useful for integrating existing MPI and HPF codes. In addition, MPI-HPF communication will be useful, where a mixture of data and task parallelism is desired. This is because MPI is popularly used for implementing task parallelism and HPF is an effective language for implementing data parallel applications. The behavior and integration of application codes with our communication techniques is also presented in this thesis.

# Contents

**Chapter 1 Introduction** 1
- 1.1 Need for Parallel Computing . . . 1
- 1.2 Parallel Programming Paradigms . . . 2
  - 1.2.1 The Message-Passing Model . . . 3
  - 1.2.2 The Shared-Memory Programming Model . . . 4
  - 1.2.3 The Data Parallel Model . . . 5
- 1.3 Our problem of calling HPF from MPI . . . 6
- 1.4 Related Work . . . 7
- 1.5 Organization of the thesis . . . 8

**Chapter 2 The Message Passing Interface (MPI)** 9
- 2.1 Task Parallel Programming . . . 9
- 2.2 Programming with MPI . . . 10
  - 2.2.1 Process Handling . . . 10
  - 2.2.2 Point-to-Point Communication . . . 11
  - 2.2.3 Collective Communication . . . 13
  - 2.2.4 Communicators and Topologies . . . 14

**Chapter 3 High Performance Fortran (HPF)** 16
- 3.1 Data Parallel Programming . . . 16
- 3.2 Programming with HPF . . . 16
  - 3.2.1 Data Mapping Features . . . 17
  - 3.2.2 Data Parallelization Features . . . 18
  - 3.2.3 Extrinsic Procedures . . . 19
  - 3.2.4 Intrinsic and Library Procedures . . . 19

**Chapter 4 Analysis of Communication Techniques** 21
- 4.1 Approach to the problem . . . 21
- 4.2 Socket Communication . . . 22
  - 4.2.1 Unix Domain Protocols . . . 24

4.2.2 Internet protocols . . . . . 26
4.3 Communication using MPI . . . . . 26
4.4 Shared Memory . . . . . 28

**Chapter 5 Application Codes** **29**
5.1 Fast Molecular Dynamics (FMD) . . . . . 29
5.2 The Fast Multipole Method (FMM) and the Anderson Method . . . . 31
5.3 Calling the Anderson code from FMD . . . . . 31

**Chapter 6 Performance Analysis** **34**
6.1 Parallel and Distributed Computing Platforms used for performance evaluation . . . . . 34
6.1.1 The IBM SP2 at UH . . . . . 34
6.1.2 The IBM SP2 at SDSC . . . . . 35
6.1.3 The HP Exemplar at CACR . . . . . 36
6.1.4 The SGI Origin 2000 at the National Center for Supercomputing Applications (NCSA) . . . . . 37
6.1.5 The Globus Metacomputing Toolkit . . . . . 38
6.2 Experimental Setup . . . . . 38
6.2.1 Model Programs . . . . . 38
6.2.2 Application Codes . . . . . 39
6.3 Communication with Sockets . . . . . 40
6.3.1 The IBM SP2 . . . . . 41
6.3.2 The HP Exemplar . . . . . 51
6.3.3 The SGI Origin 2000 . . . . . 60
6.3.4 Summary of Socket Communication . . . . . 69
6.4 Communication with MPI . . . . . 71
6.4.1 The IBM SP2 . . . . . 72
6.4.2 The HP Exemplar . . . . . 86
6.4.3 The SGI Origin 2000 . . . . . 95
6.4.4 Globus . . . . . 97
6.4.5 Summary of MPI Communication . . . . . 99
6.5 Shared Memory Communication . . . . . 100
6.6 Summary of model program mechanisms . . . . . 103
6.7 Performance of Application Codes . . . . . 106

**Chapter 7 Summary/Conclusions and Future Work** **114**
7.1 Summary . . . . . 114
7.2 Conclusions and Recommendations . . . . . 115
7.3 Ongoing and Future Work . . . . . 116

**References** **117**

# List of Figures


1.1 Parallel programming models. . . . . . . . . . . . . . . . . . . . . . . . 3
1.2 MPI-HPF inter-process communication. . . . . . . . . . . . . . . . . . 7

2.1 Example MPI program. . . . . . . . . . . . . . . . . . . . . . . . . . . 12

3.1 HPF data layout. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

4.1 Client-server socket communication. . . . . . . . . . . . . . . . . . . . 23
4.2 MPI-HPF communication with sockets. . . . . . . . . . . . . . . . . . 23
4.3 Client-server communication with UNIX domain sockets. . . . . . . . 25
4.4 MPI-HPF communication with MPI in a four process system. . . . . 27

5.1 Example FMD patch assignments to processors. . . . . . . . . . . . . 30
5.2 FMD-Anderson communication. . . . . . . . . . . . . . . . . . . . . . 32
5.3 Emulating irregular array distributions. . . . . . . . . . . . . . . . . . 33

6.1 Architecture of an IBM SP processor. . . . . . . . . . . . . . . . . . . 35
6.2 HP Exemplar node architecture. . . . . . . . . . . . . . . . . . . . . . 36
6.3 Modules in an SGI Origin 2000 system. . . . . . . . . . . . . . . . . . 37
6.4 Various types of socket communication on the IBM SP2. . . . . . . . 41
6.5 Memory copy vs. socket communication (Variable Data Size) on the IBM SP2. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
6.6 MPI-HPF RTT standard deviation (Variable Data Size per processor) on the IBM SP2. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
6.7 Memory copy standard deviation on the IBM SP2. . . . . . . . . . . 45
6.8 Increase in socket buffers on the IBM SP2. . . . . . . . . . . . . . . . 46
6.9 MPI-HPF RTT standard deviation (8 MB per processor) on the IBM SP2. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
6.10 MPI-HPF socket RTT/2 for Constant Data Sizes per processor on the IBM SP2. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
6.11 MPI-HPF socket RTT/2 for Constant Data Sizes per processor on the IBM SP2. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

6.12 MPI-HPF socket RTT variations with Constant Data Size (128-2048 bytes) on one processor of the IBM SP2. . . . . . . . . . . . . . . . . . 50
6.13 MPI-HPF socket RTT variations with Constant Data Size (16 KB-1 MB) on one processor of the IBM SP2. . . . . . . . . . . . . . . . . . 50
6.14 Various types of socket communication on the HP-Exemplar. . . . . . 52
6.15 Memory copy vs. socket communication on the HP Exemplar. . . . . 54
6.16 Memory copy standard deviation on the HP Exemplar. . . . . . . . . 55
6.17 MPI-HPF socket RTT standard deviation (Variable Data Size) on the HP Exemplar. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
6.18 MPI-HPF socket RTT standard deviation (8 MB per processor) on the HP Exemplar. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
6.19 MPI-HPF socket RTT/2 for Constant Data Sizes per processor on the HP Exemplar. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
6.20 MPI-HPF socket RTT/2 for Constant Data Sizes per processor on the HP Exemplar. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
6.21 MPI-HPF socket RTT variations with Constant Data Size (128 bytes -128 KB) on one processor of the HP Exemplar. . . . . . . . . . . . . 58
6.22 Various types of socket communication on the SGI Origin 2000. . . . 61
6.23 Memory copy vs. socket communication on the SGI Origin 2000. . . . 62
6.24 MPI-HPF socket RTT standard deviation (Variable Data Size) on the SGI Origin 2000. . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
6.25 Memory copy standard deviation on the SGI Origin 2000. . . . . . . . 65
6.26 MPI-HPF socket RTT standard deviation (8 MB per processor) on the SGI Origin 2000. . . . . . . . . . . . . . . . . . . . . . . . . . . 66
6.27 MPI-HPF socket RTT/2 for Constant Data Sizes per processor on the SGI Origin 2000. . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
6.28 MPI-HPF socket RTT/2 for Constant Data Sizes per processor on the SGI Origin 2000. . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
6.29 MPI-HPF socket RTT variations with Constant Data Size (128 bytes-128 KB) on two processors of the SGI Origin 2000. . . . . . . . . . . 68
6.30 Observed vs. expected RTT for MPI-HPF communication using MPI (communicating processes on different processors). . . . . . . . . . . . 74
6.31 Standard deviation for MPI-HPF RTT using MPI with communicating processes on different processors on the IBM SP2. . . . . . . . . . 75
6.32 MPI-HPF RTT/2 using MPI with communicating processes on different processors for Constant Data Sizes per processor on the IBM SP2. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
6.33 MPI-HPF RTT/2 using MPI with communicating processes on different processors for Constant Data Sizes per processor on the UH IBM SP2. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77

6.34 MPI communication vs. MPI-HPF communication with Variable Data Size on the UH IBM SP2. . . . . . . . . . . . . . . . . . . . . . . . 79
6.35 MPI communication vs. MPI-HPF communication with Constant Data Size on the IBM SP2. . . . . . . . . . . . . . . . . . . . . . . . 80
6.36 MPI-HPF communication with MPI (communicating processes on the same processor). . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
6.37 MPI-HPF RTT (using MPI) standard deviation on the IBM SP2 (communicating processes on the same processor). . . . . . . . . . . . . . 82
6.38 MPI-HPF RTT/2 (using MPI with communicating processes on the same processor) for Constant Data Sizes per processor on the IBM SP2. 83
6.39 MPI-HPF RTT/2 (using MPI with communicating processes on the same processor) for Constant Data Sizes per processor on the UH IBM SP2. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
6.40 Observed vs. expected MPI-HPF communication with MPI on the HP Exemplar. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
6.41 MPI-HPF RTT (using MPI) standard deviation on the HP Exemplar. (Variable Data Size) . . . . . . . . . . . . . . . . . . . . . . . . . 89
6.42 MPI communication vs. MPI-HPF communication with Variable Data Size per processor on the HP Exemplar. . . . . . . . . . . . . . . . 91
6.43 MPI communication vs. MPI-HPF communication with Constant Data Size 8 MB per processor on the HP Exemplar. . . . . . . . . . . 92
6.44 MPI-HPF RTT (using MPI) for Constant Data Sizes per processor on the HP Exemplar. . . . . . . . . . . . . . . . . . . . . . . . . . . 93
6.45 MPI-HPF RTT (using MPI) for Constant Data Sizes per processor on the HP Exemplar. . . . . . . . . . . . . . . . . . . . . . . . . . . 94
6.46 Various types of MPI communication on the SGI Origin 2000. . . . . 96
6.47 Shared memory communication on the IBM SP2.(Constant Data Size) 101
6.48 Shared memory communication on the HP Exemplar.(Constant Data Size) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
6.49 Shared memory communication on the SGI Origin 2000.(Constant Data Size) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
6.50 Application programs (send) vs. model programs (RTT/2) on the SDSC IBM SP2. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
6.51 Application programs (recv) vs. model programs (RTT/2) on the SDSC IBM SP2. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
6.52 Application programs (send) vs. model programs (RTT/2) on the HP Exemplar. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
6.53 Application programs (recv) vs. model programs (RTT/2) on the HP Exemplar. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109

6.54 Application programs (send) vs. model programs (RTT/2) on the SGI Origin 2000. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
6.55 Application programs (recv) vs. model programs (RTT/2) on the SGI Origin 2000. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110

# List of Tables

6.1 Socket communication on the IBM SP2 (Variable Data Size). . . . . . 42
6.2 Socket communication bandwidth on the IBM SP2 (Variable Data Size). 43
6.3 Socket communication on IBM SP2 (Constant Data Size). . . . . . . 47
6.4 Socket communication on the HP Exemplar (Variable Data Size). . . 52
6.5 Socket communication bandwidth on the HP Exemplar (Variable Data Size ). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
6.6 Socket communication on HP Exemplar (Constant Data Size). . . . . 55
6.7 Socket communication on the SGI Origin 2000 (Variable Data Size). . 61
6.8 Socket communication bandwidth on the SGI Origin 2000 (Variable Data S ize). . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
6.9 Socket communication on the SGI Origin 2000 (Constant Data Size). 64
6.10 Communication with MPI on the IBM SP2 (Variable Data Size). . . 73
6.11 MPI communication bandwidth on the IBM SP2 (Variable Data Size). 73
6.12 MPI-HPF communication using MPI on the IBM SP2. (Constant Data Size ) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
6.13 MPI-HPF communication with MPI on the IBM SP2 (Variable Data Size). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
6.14 MPI-HPF communication bandwidth using MPI on the IBM SP2 (Variable Data Size). . . . . . . . . . . . . . . . . . . . . . . . 82
6.15 MPI-HPF communication using MPI on the UH IBM SP2 (Constant Data Size) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
6.16 Communication with MPI on the HP Exemplar (Variable Data Size). 87
6.17 MPI communication bandwidth on the HP Exemplar (Variable Data Size). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
6.18 MPI-HPF communication using MPI on the HP Exemplar (Constant Data Size) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
6.19 MPI-HPF communication (using MPI) on Globus. . . . . . . . . . . . 98
6.20 MPI-HPF communication on the IBM SP2. . . . . . . . . . . . . . . 105
6.21 MPI-HPF communication on the HP Exemplar. . . . . . . . . . . . . 105
6.22 MPI-HPF communication on the SGI Origin 2000. . . . . . . . . . . . 105

6.23 FMD-Anderson with socket communication on the SDSC IBM SP2. . 110
6.24 Anderson-FMD with socket communication on the SDSC IBM SP2. . 111
6.25 FMD-Anderson with socket communication on the SGI Origin 2000. . 111
6.26 Anderson-FMD with socket communication on the SGI Origin 2000. . 112
6.27 FMD-Anderson with socket communication on the HP Exemplar. . . 112
6.28 Anderson-FMD with socket communication on the HP Exemplar. . . 113

# Chapter 1

# Introduction

## 1.1 Need for Parallel Computing

Engineers and scientists have realized the need to use large-scale computer simulations to design and build prototypes for products in development, and in developing an understanding of complex phenomena. Usually, these simulations require billions and sometimes even trillions or more of arithmetic operations per second to obtain results in a reasonable time frame. In addition, one generally needs large storage and memory requirements to solve these *grande* problems. These high-performance requirements can only be achieved through parallel computing with current technology.

Nowadays, parallel architectures can be classified as follows

- **Vector processing machines.** These architectures perform identical pipelined operations on vectors of data. Today some of these architectures have multiple processors accessing a shared-memory pool.

  Examples: NEC SX Series, Fujitsu (VP Series), Hitachi (S820 Series).

- **Distributed shared-memory machines.** These architectures are also called Cache-Coherent Nonuniform Memory Architectures (CC-NUMA). These machines are comprised of a collection of processors and memory modules connected by an interconnection network, which may either be bus-based or switch-based. Processors can view the entire available memory as a global memory and inter-processor communication is accomplished by using shared data.
  Examples: SGI Origin 2000, HP Exemplar.

- **Distributed memory machines.** Each processor in these architectures has its own private memory and multiple processors are connected by a high-speed interconnection network. Inter-processor communication is achieved by sending messages over the network.
  Examples: IBM SP2, Cray T3D, Intel Paragon XP/S.

- **Metacomputing Environments.** Sets of processors (in the form of parallel computers or networks of workstations) are connected by high-speed networks like the vBNS and the Abiline network. Communication is achieved by using messages which run over layers of communication protocols and security software.
  Examples: The Globus Metacomputing Toolkit [1], Legion [2].

## 1.2 Parallel Programming Paradigms

Parallel programs center around the concept of a process. A process is defined as an instance of a program or components thereof. Depending on the programming paradigm, a process may run on single or multiple processors. The former is the

core for programs making use of the Message Passing Interface (MPI) [3] paradigm. The latter paradigm is used for instance in High Performance Fortran (HPF) [4]. Conversely, most implementations of parallel programming paradigms, like MPI and HPF only allow a single process per processor. Multiple processes are often supported in shared-memory programming paradigms like OpenMP [5]. We now proceed to discuss three popular parallel programming models which are illustrated in Figure 1.1.

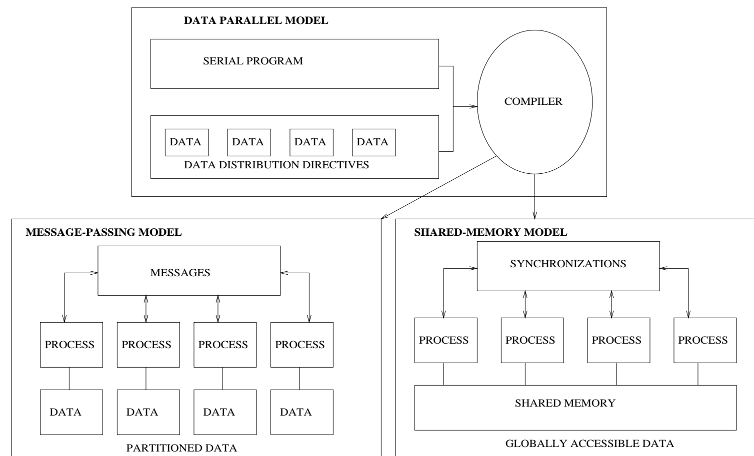


Figure 1.1: Parallel programming models.

### 1.2.1 The Message-Passing Model

Each process has its own private data. Communication with other processes takes place with the help of explicit messages. Processes alternate their execution between communication with other processes and computation on their own data.

With this model parallel programs can be programmed using the following two approaches:

- **The Single Program Multiple Data (SPMD) approach.** Using this approach all the processes in the system are instances of a single program.
- **The Multiple Program Multiple Data (MPMD) approach.** In this approach processes in the system can be instances of two or more programs.

The programmer has a great degree of control over the distribution of data and over process communication, at the expense of complicating code and making application programs hard to read and understand. In addition, the programmer has the responsibility of taking care of avoiding problems like deadlocks and managing process synchronization. Message-passing models are usually implemented as libraries with PVM (Parallel Virtual Machine) [6] and MPI (Message Passing Interface) [3] being the most common examples. The model is well suited for distributed-memory machines, but is also supported on distributed shared-memory type architectures and meta-computing environments.

### 1.2.2 The Shared-Memory Programming Model

In this model a shared memory space is available to all processes. Processes can be created statically at the beginning of the program execution or can be spawned dynamically as the program execution proceeds. Process coordination is managed by primitives that:

- specify variables that can be accessed from all processes.
- prevent processes from improper access of shared resources.
- provide a means for synchronizing the processes.

CC-NUMA architectures directly support this model, however, it can also be supported on distributed-memory machines through additional compilers or run-time systems. Examples of the shared-memory model include the OpenMP API (Application Program Interface), which is emerging as a widely accepted standard for this type of programming.

### 1.2.3 The Data Parallel Model

In the data parallel model, there is a global view of memory. Simple data structures, such as arrays, are distributed over processors. For execution models on multiprocessors, a separate process may be created on each processor. In such a case, all processors execute the same code on their local array sections. This execution model follows the Single Program Multiple Data (SPMD) approach, which is also commonly used in the MPI or the PVM paradigms. Synchronization is enforced, whenever communication is required between processors. Unlike the message-passing model where computations or tasks are parallelized, in the data parallel model data is parallelized and the code executed is the same across all processes. Data parallel models are frequently implemented in standard programming languages like Fortran or C with special directives for data distribution and parallelism. In addition, the data distribution and inter-process communication is taken care of by the compiler depending on what parallel directives the programmer has specified. Therefore, data parallel programming often frees the programmer from writing process management and synchronization code. On the negative side, the model favors simple data structures such as arrays; and hence programming with irregular applications is an issue [7]. Data parallel languages, such as High Performance Fortran (HPF), follow the principles of this model. HPF is basically a superset of Fortran 90/95, where the programmer can

specify various data distribution directives and use other HPF intrinsic functions. HPF compilers translate the source code into an equivalent SPMD message-passing or shared-memory code.

## 1.3 Our problem of calling HPF from MPI

The MPI standard is very popular in parallel programming and several well-tested and efficient implementations exist, including some public domain implementations. As a result, there are plenty of high-performance application programs and tools written using the MPI libraries. Commercial and public domain implementations of HPF are also available for a variety of parallel computing platforms. HPF is being accepted as the language for writing regular data parallel programs. Hence, it is desirable to have calling mechanisms from MPI to HPF and vice versa, as both programming techniques are important in their own respects. To achieve the above, HPF provides mechanisms by which an HPF program can call procedures written using other parallel programming models or other programming languages. However, as noted earlier, MPI implements a lower level programming model than HPF. Therefore, establishing a calling mechanism from MPI to HPF is difficult and even the latest MPI standard [8] does not specify such a calling mechanism. In this thesis, we aim to present high-performance, portable and efficient techniques for communication between codes using the MPI and HPF paradigms. Rather than using explicit procedure calling from MPI codes, our techniques are centered around inter-process communication between MPI and HPF processes as illustrated in Figure 1.2.

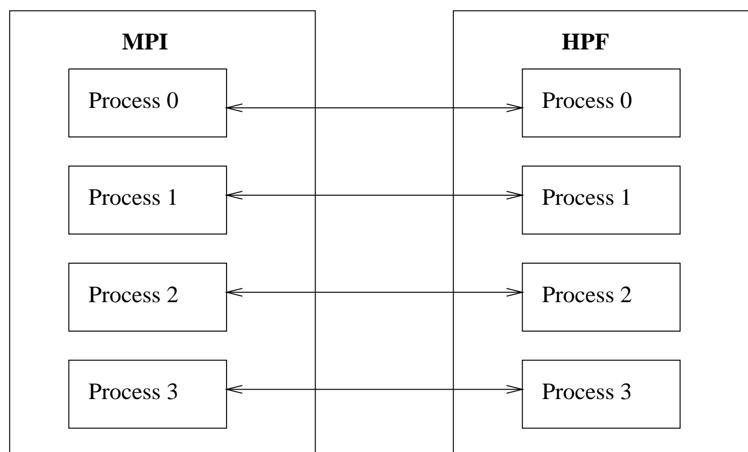


Figure 1.2: MPI-HPF inter-process communication.

## 1.4 Related Work

A fair amount of research has been done in incorporating some form of task parallelism in HPF. Gross et. al and Subhlok and Yang, [9, 10], discuss the advantages of having simple task parallel directives in data parallel codes written in HPF and also describe the design and implementation of compilers and models which support that kind of behavior. The HPF/MPI project [11] provides for concurrent execution of HPF tasks, which communicate by explicit message-passing in MPI. On another front, the KeLP-HPF project [12] makes use of an SPMD coordination layer to coordinate various HPF tasks in a high-level manner with no need for advanced HPF compilers/languages and changes to existing HPF codes. However, although all these projects have concentrated on the creation and management of multiple HPF tasks, they do not consider mixed language models, where suitable task parallel (MPI) and data parallel codes (HPF) can be used to do computations simultaneously. It is this area that we are trying to address and provide solutions for.

## 1.5 Organization of the thesis

This introductory chapter reviews some of the basic hardware and software concepts of parallel computing. We have then proceeded to explain the need for calling/communication mechanisms between HPF and MPI and compared it with previous work. In the following two chapters, we describe the details of programming with both MPI and HPF to give an idea why their communication is complex. Chapter 4 describes our approach, which is focused on the issues of high-performance, scalability and portability. We discuss three communication techniques: communication with UNIX sockets, communication with MPI, and shared memory communication. We then proceed in Chapter 5, to explain how some application codes benefit from our work. Performance results on various platforms are listed and analyzed in Chapter 6. Finally, we conclude our work and suggest future directions in Chapter 7.

# Chapter 2

# The Message Passing Interface (MPI)

## 2.1 Task Parallel Programming

Task Parallel Programming is the concurrent execution of different instruction streams (also known as tasks). The instruction streams can be employed on the same or multiple data sets. Typically, computation is parallelized in task parallel programs. A common example of task parallelism is pipelining, where various computations are parallelized per functional unit (or processors) and results are sent between functional units. Generally in task parallelism, the programmer has to manage load balancing among processors, minimizing communication costs, and coordinating communication among processors. The Message Passing Interface (MPI) supports this type of programming and in the following sections, we will study its features.

## 2.2 Programming with MPI

MPI is a software standard developed by the MPI forum. Although MPI was initially designed for MPP's and workstation clusters, nowadays it is available on a variety of High Performance Computing (HPC) platforms, including SMP's. MPI defines a variety of low-level communication routines for both point-to-point and collective communication between processes. The standardization of MPI enables developers to port their codes easily to other platforms. We now proceed to discuss the important features of MPI.

### 2.2.1 Process Handling

MPI does not specify how processes are mapped to physical processors. This and other information is assumed to be provided by the vendors. Groups of MPI processes participating in a particular computation are assumed to lie in a communication world (also known as a communicator). MPI-1 [13] assumes that all processes are statically allocated, that is, the number of processes is fixed at the beginning of program execution and no processes can be added or deleted during execution. However, the MPI-2 standard [8] allows processes to be created and terminated cooperatively during program execution. In addition, MPI-2 also allows for communication between different MPI applications, which may be executing using different communicators. In the following sections, we proceed to explain various patterns of communication that occur in a single communication domain.

### 2.2.2 Point-to-Point Communication

Point-to-point communication is the transmission of data between a pair of processes where one process sends and the other receives. Figure 2.1 illustrates an example program written in C with MPI library calls, which demonstrates simple MPI communication.

The **MPI_Init** call is invoked before calling any other MPI function. Its main purpose is to enable system initialization so that the MPI library can be used. Each process is able to get its unique rank by using the **MPI_Comm_rank** command. The size of the communicator (in this case **MPI_COMM_WORLD** ) is determined by using the **MPI_Comm_size** command. Processes can send and receive messages by using the **MPI_Send** and the **MPI_Recv** commands. To send a message, the sender specifies a communicator and a rank within the communicator to identify a destination process. In addition, the sender has to specify a data-type and the data to be sent as well as a tag value, which helps identify messages. Similarly the receiver specifies a rank, communicator, tag, data-type and receiving buffer to receive a message. Both **MPI_Send** and **MPI_Recv** are blocking calls. Therefore **MPI_Send** does not return until data is transferred either directly to the receive buffer or to an intermediate system buffer. Also, **MPI_Recv** returns only after the receive buffer contains the message. The MPI standard also defines non-blocking semantics (**MPI_ISend**, **MPI_IRecv**), which allow for overlap of computation with communication or the overlap of the transmissions of different messages. The non-blocking functions generally consist of a posting function that starts a particular operation and a test-for-completion function, which returns whether the operation has completed. MPI provides a substantial amount of point-to-point communication semantics without trying to compromise on performance. Issues regarding the exact

```
/* MPI example program */
#include <stdio.h>
#include <string.h>
#include "mpi.h"

#define DATA_SIZE 100
main(int argc, char* argv[]) {
    int         my_rank;       /* rank of process      */
    int         p;             /* number of processes  */
    int         source;        /* rank of sender       */
    int         dest;          /* rank of receiver     */
    int         tag = 0;       /* tag for messages     */
    char        message[100];  /* storage for message  */
    MPI_Status  status;        /* return status for    */
                               /* receive              */
    double data[DATA_SIZE];    /* buffer for sending/receiving */

    MPI_Init(&argc, &argv);  /* Start up MPI */

    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); /* Find out process rank  */

    MPI_Comm_size(MPI_COMM_WORLD, &p);  /* Find out number of processes */

     if(my_rank==0)
     {
        read_data(data);
        MPI_Send(data,DATA_SIZE,MPI_DOUBLE,p-1,tag,MPI_COMM_WORLD);
     }
     else
     {
        if(my_rank==p-1)
        MPI_Recv(data,DATA_SIZE,MPI_DOUBLE,0,tag,MPI_COMM_WORLD);
     }
     /* Shut down MPI */
     MPI_Finalize();
} /* main */
```

Figure 2.1: Example MPI program.

semantics of both blocking and non-blocking operations are handled by having modes that instruct the underlying protocol about how data is to be transferred.

### 2.2.3 Collective Communication

Collective communication involves the transmission of data or synchronization among processes in a communicator. Generally, two or more processes are involved in collective communication. The following are examples of collective communication:

- **Barrier synchronization.** This operation is used when all the processes in a communicator need to be synchronized.
- **Global Communication functions.** These functions implement the following operations
  - **Broadcast operation.** A single process sends the same data to all other processes in the communicator.
  - **Gather operations.** The *gather* operation is used when all the processes in the communicator send data to a single process that gets collected in a single data structure. One can also use an *allgather* operation where all processes in a communicator can receive the result.
  - **Scatter operation.** This operation is used when a data structure residing on a single process needs to be distributed across all the processes in the communicator.
  - **All-to-all communication.** Data is gathered/scattered from all processes to all processes in a communicator.

- **Global reductions.** Operations like sum, min, max, or other user-defined operations can be used on combined data from all the processes in the communicator. Reductions can be such that the result is returned to only one process or all processes in the communicator. Other variations include combined reduction and scatter operations and prefix operations, where reduction operations are carried out on data from an ordered subset of the processes.

### 2.2.4 Communicators and Topologies

MPI uses concepts such as communicators and topologies, which are not common in other message-passing systems. Multiple communicators are mainly used when in the application it is desirable to divide up the processes to allow different groups of processes to perform independent work. MPI defines two kinds of communicators

- **Intra-communicators:** These communicators consist of processes that can perform both point-to-point and collective communication amongst each other. An intra-communicator is composed of a group and a context, where a group is an ordered collection of processes based on process rank and a context is a system-defined object that uniquely identifies a communicator. Both groups and communicators cannot be directly accessed by the user, as their internal representation depends on the MPI implementation. Contexts, however, cannot be accessed by MPI functions at all, as their definition is implicitly done when a communicator is created.

- **Inter-communicators:** Communicators that are used for point-to-point communication between processes belonging to disjoint intra-communicators are known as inter-communicators.

It is possible to associate additional information with a communicator in addition to the group and context. A possible attribute for this cached information is a topology. A topology can provide a mechanism for addressing processes and also information about mapping of processes onto hardware. MPI provides special functions for creating and using grid and graph topologies.

# Chapter 3

# High Performance Fortran (HPF)

## 3.1 Data Parallel Programming

Data parallel programming involves parallelization by assigning data elements to various processors. A simple example is matrix multiplication where two $n$ x $n$ matrices $A$ and $B$ are multiplied to obtain a matrix $C = (c_{i,j})$. Each element of $C$ is computed by doing a dot product of the $i^{th}$ row of $A$ with the $j^{th}$ column of B. Therefore, computing each resultant element is essentially the same operation on different data, which can be appropriately distributed. High Performance Fortran (HPF) supports data parallel programming.

## 3.2 Programming with HPF

High Performance Fortran (HPF) is a set of extensions to Fortran 90/95, which enable programmers to write portable high-level data parallel programs. Notable HPF language features include directives for suggesting implementation strategies

and asserting facts to the compiler; direct extensions to the Fortran language; new library routines important for high performance and changes and restrictions to existing Fortran 90/95. In addition, the HPF standard [14] stresses that highly-efficient HPF programs on a particular platform should be able to achieve reasonably high efficiency on other parallel machines, so that programming efforts and platform-specific performance tuning are reduced. We observe that a number of commercial and public domain HPF compilers and tools are available [15] [16] and as a result there are a variety of data parallel applications written in HPF [17]. As HPF consists mainly of extensions to Fortran 90/95, we proceed to discuss some of the important extensions.

### 3.2.1 Data Mapping Features

HPF provides features for the programmer to control the distribution of data among processors in multi-processor systems (**DISTRIBUTE**) and the co-location of data (**ALIGN**). The desire to control data distributions is based on the expectation that processors can read/write their local memory faster than memory on another processor, and is pertinent, considering current parallel architectures. Dynamic directives like **REDISTRIBUTE** and **REALIGN** are executable statements, not just directives. The **TEMPLATE** directive is used to abstract spaces of indexed positions, while the **PROCESSORS** directive defines a set of abstract processors. In addition, HPF also provides directives to prescribe the alignment and distribution of procedure arguments.

### 3.2.2 Data Parallelization Features

Constructs, such as the **INDEPENDENT** directive describe operations that can be performed in parallel. The **FORALL** and **PURE** statements were initially a part of the HPF specification [18], but now are removed as they are now part of ISO Fortran [14].

The following HPF code-segment shows the use of data mapping directives and data parallel constructs.

```
        REAL a(16), b(32)
        INTEGER i
!HPF$ PROCESSORS procs(4)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: a
!HPF$ ALIGN a(i) with b(2*i-1)


        ...
!HPF$ INDEPENDENT
        DO i=1,16
                a(i) = 100 * b(2*i-1)
        END DO
```

Array $a$ is distributed in a block fashion on the abstract processor set *procs* and for all values of $i$, $a(i)$ and $b(2*i-1)$ are mapped on the same processor with the help of **DISTRIBUTE** and **ALIGN** directives as shown in Figure 3.1.

The **INDEPENDENT** directive informs the HPF compiler or pre-processor that it is safe to execute in parallel the **DO** loop that follows. Hence, as both the left-hand side and right-hand side variables are on the same processor, no communication

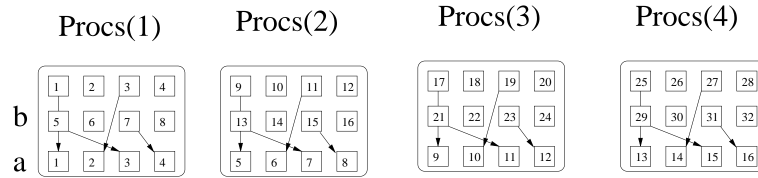


Figure 3.1: HPF data layout.

is required and the loop executes totally in parallel.

### 3.2.3 Extrinsic Procedures

As HPF is a high-level parallel programming language, it is sometimes desirable to express operations in other lower-level languages. For example, one may want to implement some codes using explicit message-passing to express non-uniform operations with respect to the data set or task parallelism. HPF provides extrinsic procedures that enable the programmer to call procedures written in other parallel programming paradigms, or other languages such as C.

### 3.2.4 Intrinsic and Library Procedures

In addition to Fortran 90/95's intrinsic functions, HPF has its own library module **HPF_LIBRARY** that contains a large number of additional subroutines and functions, which are further grouped into :

- **System Inquiry Functions:** These functions return values which describe the size and shape of the underlying processor array.

- **Mapping Inquiry Subroutines:** HPF subroutines which permit the program to determine array mappings at run time are grouped in this category.

- **Computational Functions:** These functions are grouped as follows:
  - Array location functions
  - Bit manipulation functions
  - Array reduction functions
  - Array combining scatter functions
  - Array prefix and suffix functions
  - Array sorting functions

# Chapter 4

# Analysis of Communication Techniques

## 4.1 Approach to the problem

In chapters 2 and 3, we have studied the features of both MPI and HPF for writing parallel programs. We observe that MPI provides a great deal of control for process communication and management, whereas HPF employs a much higher-level data organization and parallelization model. MPI programs are limited to knowing about the existence of other task parallel paradigms, unlike HPF, where one can call programs written in other parallel programming paradigms. Therefore, to enable calls to procedures written in HPF from MPI, one alternative is to develop a special library for MPI programs that would pass arguments to HPF subroutines identical to that of compiled HPF code [12]. This library would have to provide information to the HPF procedure about the shape, alignment and distribution of array arguments. In

addition, a mechanism must be created for assigning processor sets for the subroutine to execute independently. It is obvious that the implementation of these features is heavily dependent on the HPF compiler and the execution environment. A less complicated approach is to perform inter-process communication between MPI and HPF programs as shown previously in Figure 1.1. This communication serves mainly to pass data between the two programs. One can use the **HPF_LOCAL** extrinsic procedure to escape to the local process in HPF. It needs to be emphasized that the communication should not affect the scalability of existing MPI and HPF codes and should in itself necessarily be of high-performance, scalable and portable. We proceed to discuss the inter-process communication techniques we have employed for MPI-HPF communication.

## 4.2 Socket Communication

Berkeley Sockets are a form of inter-process communication provided by the 4.3 BSD operating system. Mechanisms for communication between processes on the same machine and different machines are provided by Berkeley sockets by using the Unix domain and the TCP/IP protocols respectively. Figure 4.1 shows how client and server programs communicate using these protocols.

Considering the above features, the fact that almost all Unix vendors have support for Berkeley sockets, and because of its easy to use API, this mechanism was considered for the communication between MPI and HPF processes. The MPI program calls client code and the HPF program invokes the server code or vice versa as shown in Figure 4.2.

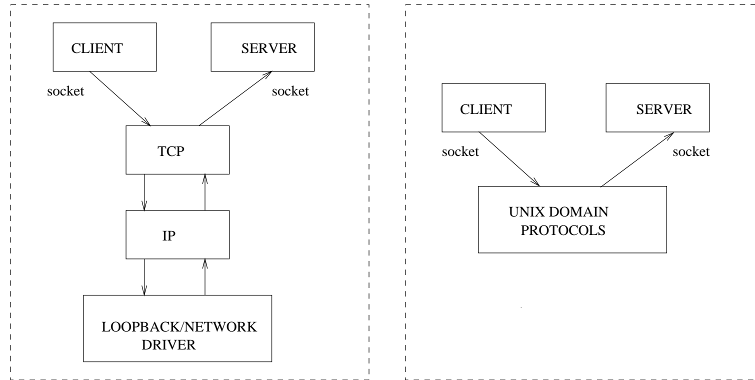


Figure 4.1: Client-server socket communication.

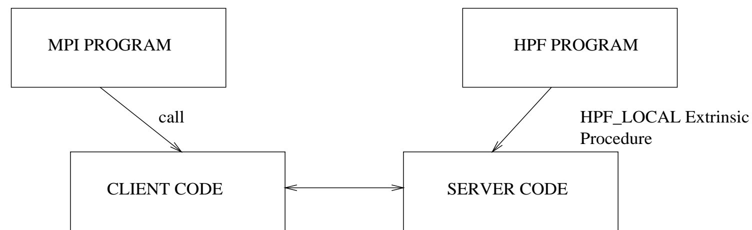


Figure 4.2: MPI-HPF communication with sockets.

We now proceed to discuss the finer details of the Berkeley communication protocols.

### 4.2.1 Unix Domain Protocols

These protocols are used to communicate between processes on the same machine. The implementation of these protocols is built in the Unix kernel, just like other communication protocols. The Unix domain protocols provide less processing than the TCP/IP communication protocols, as they know that communication is done between processes on the same host. Although other forms of Unix inter-process communication exist, these protocols have the advantage that they use the same API as that of networked programs. Some common applications, which use the Unix domain protocols for communicating on the same Internet host are:

1. The X Windows client and server programs.
2. The BSD print spooler, i.e., the *lpr* client and the *lpd* server.
3. Pipes in Berkeley derived kernels; the implementation is done using Unix sockets.
4. Reading control messages and articles from news readers in InterNetNews daemons is made through Unix sockets.

The Unix domain protocols are defined by the AF_UNIX address family in 4.3 BSD. Like other communication protocols (Example: TCP), 4.3 BSD provides a reliable connection-oriented interface and an unreliable connection-less interface to the Unix domain protocols. The connection-oriented protocol (also know as the stream protocol) provides flow control, whereas the connection-less protocol (also

known as the datagram protocol) does not provide flow control. Thus, when using datagram sockets the programmer has to see to it that a fast sender does not swamp a slower receiver with data. In addition, the programmer has to take care that datagrams are received in the intended order.

The sequence of steps involving communication with stream-oriented UNIX domain protocols is shown in Figure 4.3. The address that a client uses to identify the server is a pathname. When data transfer takes place, the kernel code for sending data appends data from user space directly to the receiving socket's receive buffer as there is no need for copying data to the sender process send buffer and then sending it (as done in the TCP protocol).

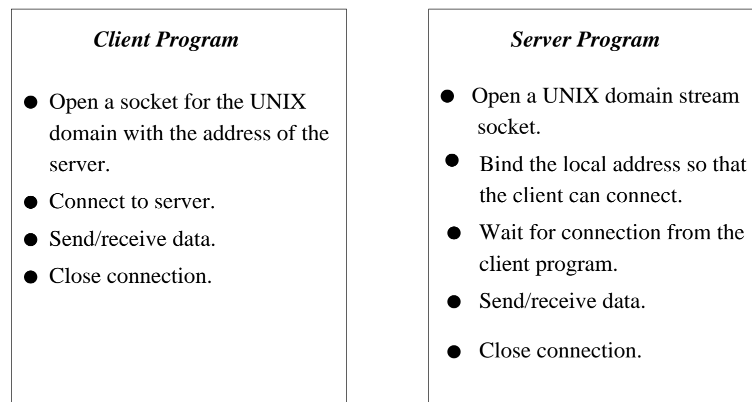


Figure 4.3: Client-server communication with UNIX domain sockets.

One should note that the sender process code in the kernel will block till the receiver has received a chunk of the data less than or equal to the socket buffer high water mark (this limit will vary from OS to OS). The system call will return the number of bytes written to the user program. Therefore, the programmer has to take care that the correct number of bytes is written/received.

For communication between the MPI and HPF processes, we have used the stream Unix domain protocol to allow for communication between the two corresponding processes on the same machine. Each MPI/HPF process recognizes the corresponding HPF/MPI process with the same process rank by including the process rank as a part of the address pathname.

### 4.2.2 Internet protocols

4.3 BSD uses the TCP/IP protocols for communication between processes on different machines (Internet hosts). Just like the UNIX domain sockets there are reliable (TCP) and unreliable (UDP) protocols. Although, our MPI-HPF communication is currently intended for processes on the same Internet host, we have used the reliable connection-oriented protocol (TCP) for our communication and compared its performance with that of the Unix domain protocols on various operating systems. The Internet protocols are defined by the AF_INET address family in 4.3 BSD.

## 4.3 Communication using MPI

One can take advantage of the fact that HPF can call MPI subroutines to solve our communication problem. This is done by using an MPMD (Multiple Program Multiple Data) model for running parallel programs. Based on the assumption that the HPF compiler uses MPI for communication, one can split the default communicator **MPI_COMM_WORLD** into two; one half runs HPF processes and the other half runs MPI processes. This model is illustrated in Figure 4.4.

However, one has to take care that all HPF data distributions are local only to its intended set of processors. This can be achieved with the help of the following

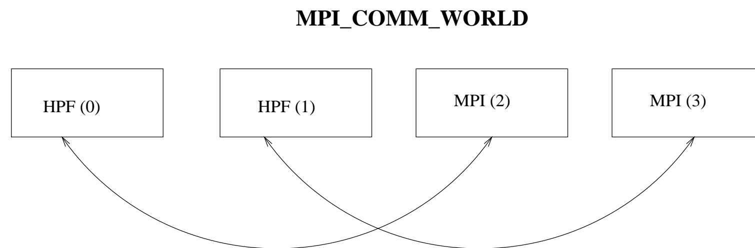


Figure 4.4: MPI-HPF communication with MPI in a four process system.

HPF directives.

```
!Specify the first partition consisting of half the number of processors.

!HPF$ PROCESSORS line( NUMBER_OF_PROCESSORS()/2 )

! Distribute arrays on the HPF partition of processors

!HPF$ DISTRIBUTE(BLOCK) ONTO line :: myArray
```

In addition, existing MPI codes will have to be modified to use a different MPI communicator world than the default **MPI_COMM_WORLD**, as it will be used for coordinating between MPI and HPF processes. This can be achieved by using standard MPI functions to create new groups and communicators. Each MPI/HPF process needs to calculate the rank of the corresponding HPF/MPI process before proceeding with the data transfer.

## 4.4 Shared Memory

In the socket communication technique, there are two memory copy operations before data finally is transferred to the HPF/MPI side. One memory copy is from the data segment of the sending process to the receive socket buffers and the second copy is from the receive socket buffers to the data segment of the receiving process.

Shared memory is another way of doing the same type of communication with possibly less overhead than UNIX sockets. This is because all socket I/O goes through certain predefined routines regardless of protocol (i.e., TCP, UDP, Unix domain). This generality hinders performance, although it helps in selecting various protocols. Hence, simple memory copies with synchronization between two processes using semaphores could improve performance. Before doing the data transfer though, one has to use system calls to establish memory mappings that allow the processes to share the memory. For this, just like the sockets code where we have a unique pathname per MPI-HPF process pair, we need to generate a special shared memory key per process pair and create a shared memory segment using that key.

# Chapter 5

# Application Codes

## 5.1 Fast Molecular Dynamics (FMD)

FMD is a parallel molecular dynamics package developed under the Computational Chemistry and Materials Science (CCM) Common High Performance Software Support Initiative (CHSSI) effort of the Department of Defense High Performance Computing Modernization Program's (DoD HPCMP) and is designed for high-performance simulations in chemistry and materials science. The source code for FMD is derived in part from the NAMD molecular dynamics package from the University of Illinois [19]. Particle simulations consist of the computation of atomic trajectories, which is done by numerically solving equations of motion using an empirical force field; currently CHARMM [20]. This force field defines interactions between atoms in terms of bonds, bond angles, dihedral angles, improper dihedral angles, van der Waals, and electrostatic interactions. Execution of FMD takes place by reading existing molecular structures and producing simulated results of final molecular structures and trajectories. These molecular structures are described by files containing atomic

coordinates and velocities (PDB/HDF format), protein structures (PSF format), and parameters.

FMD is implemented using C++ with MPI calls and is parallelized using spatial decomposition. Space occupied by the molecular model is divided up into patches, which are then assigned to processors (MPI processes) as shown in Figure 5.1. The total number of patches and hence, atoms per processor may not be the same across all processors. During the simulation, the number of patches per process and the location of patches can be adjusted to provide load balancing across all processes.

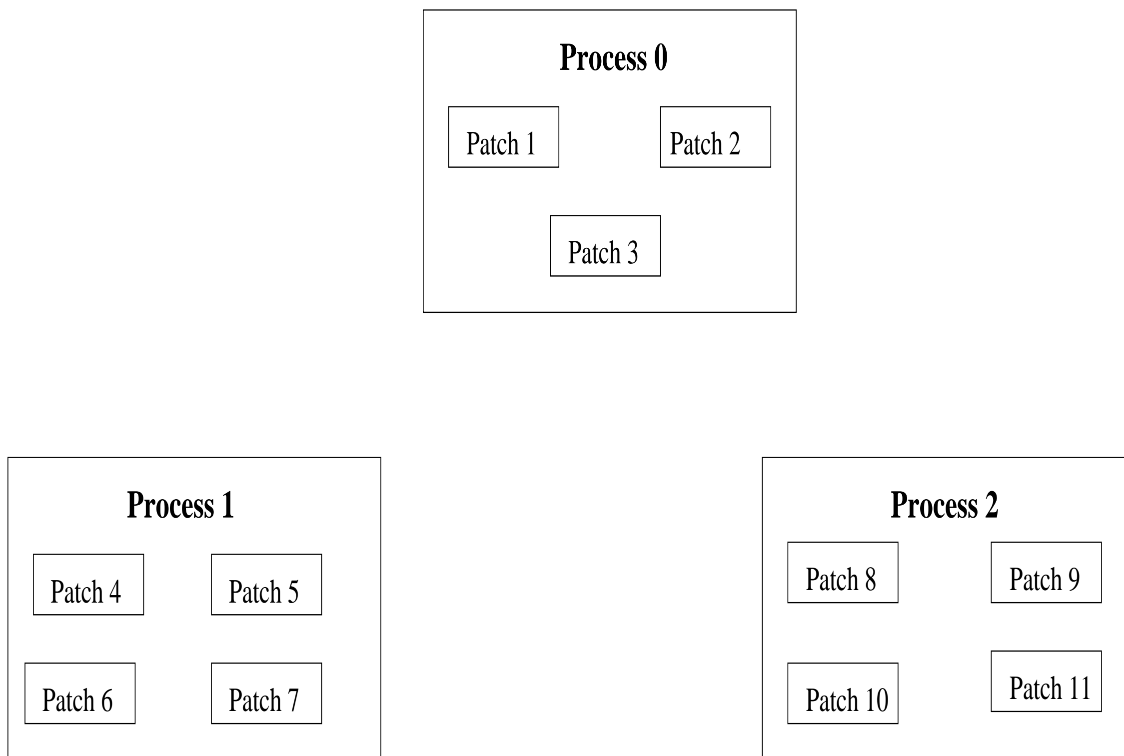


Figure 5.1: Example FMD patch assignments to processors.

## 5.2 The Fast Multipole Method (FMM) and the Anderson Method

The direct approach to the pairwise calculation of electrostatic interaction (Coulomb's law) has a complexity of $O(N^2)$. Although FMD does have a fully parallel implementation of the direct method, it is not suitable for molecular systems of a substantial size ($> 1000$ atoms), on account of the cost scaling. Therefore, FMD aims to provide for the selection of two fast multipole algorithms, which reduce the complexity from $O(N^2)$ to $O(N)$.

- **The Fast Multipole Method in Three Dimensions (FMM3D).** This method is an implementation [21] of the Fast Multipole Algorithm (FMA) of Greengard and Rokhlin [22] in three-dimensions [23]. The FMA uses power series to handle groups of long-range interactions and the direct method for handling the short-range interactions. FMM3D is a non-adaptive code, implemented in C and Fortran 77 with MPI libraries.

- **Hierarchical Adaptive and Non-adaptive $N-$body codes based on Anderson's Method.** Anderson's $O(N)$ $N-$body algorithm [24] is derived from computational elements based on Poisson's formula. Based on this algorithm, we have three-dimensional adaptive [25] and non-adaptive [26] data parallel codes written in HPF.

## 5.3 Calling the Anderson code from FMD

Calling the FMM code from FMD is not a problem, as both codes are message-passing codes. Before calling FMM, FMD needs to gather coordinates and charges

from the patches on each node and pass them to FMM. After the calculations, the results are distributed back to the patches. However, to interface the HPF codes for the Anderson method to FMD, we will have to use our MPI-HPF communication techniques. As shown in Figure 5.2, FMD has to send coordinates and charges to the Anderson code, which after doing calculations, sends potential and forces back to FMD.

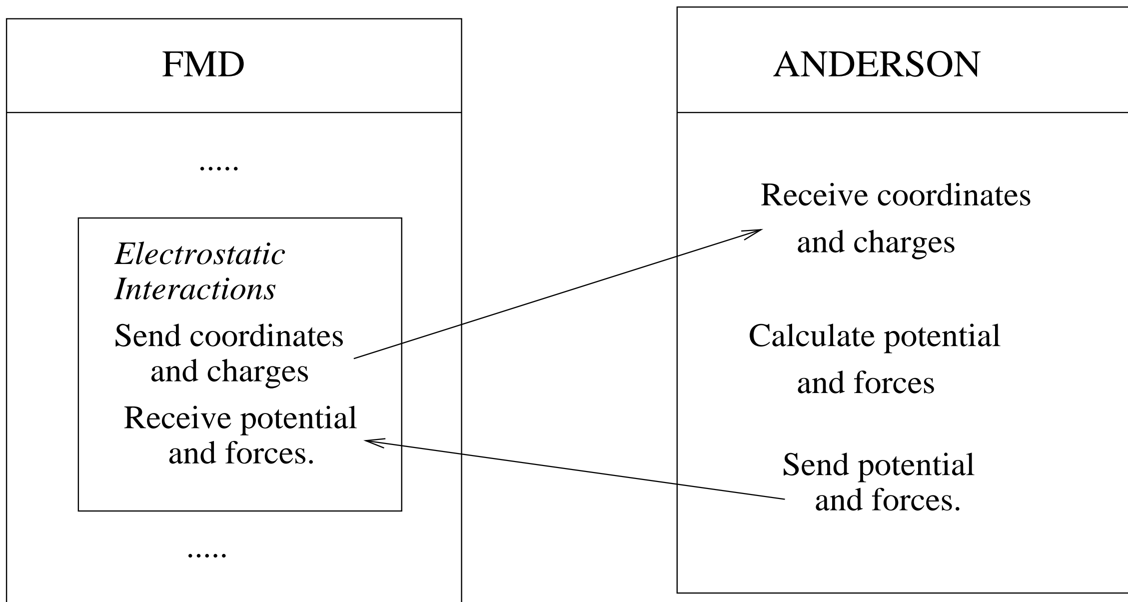


Figure 5.2: FMD-Anderson communication.

Since the total number of atoms per FMD process are unequal, and because of the fact that our current version of the HPF compiler [27] does not support irregular data distributions we have to pad the receiving regular HPF arrays with dummy elements as shown in Figure 5.3. This is done by allocating arrays of size $max_subgrid * number_of_processors$, where $max_subgrid$ is the maximum number of atoms on an FMD process. To proceed to the HPF computations though, we need

to reshuffle the arrays to obtain regular HPF arrays per process. This is done using *copy_scatter* operations with appropriate *mask* and *index* arrays to move the non-dummy elements to the unpadded arrays. HPF arrays are padded too, unless the size is evenly divisible by the number of processors. However, the HPF array padding is only made at one edge of each dimension. The reshuffling of the data in the HPF array is also likely to reduce the maximum number of elements per processor, hence improving performance of computations on the arrays.

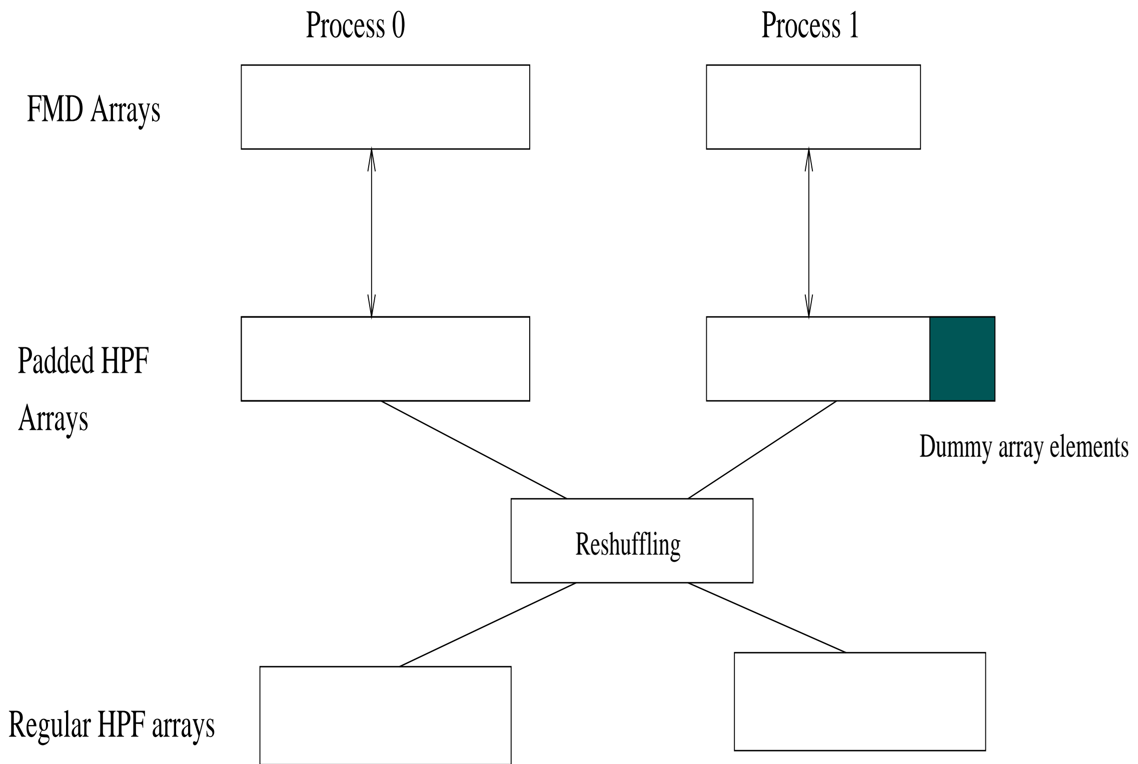


Figure 5.3: Emulating irregular array distributions.

# Chapter 6

# Performance Analysis

In this chapter, we discuss the cost of the three techniques described previously for MPI-HPF communication. Prototype MPI and HPF programs were used and round-trip times were measured to evaluate the performance and scalability of each technique. We also present an analysis of how our techniques perform with the application codes presented in Chapter 5. Initially though, we proceed to describe the hardware platforms used in the thesis.

## 6.1 Parallel and Distributed Computing Platforms used for performance evaluation

### 6.1.1 The IBM SP2 at UH

The IBM SP2 at the University of Houston (UH) has 64 RS/6000 processors, each running its own copy of the IBM Unix clone AIX 4.3. The architecture of an IBM SP processor is illustrated in Figure 6.1.

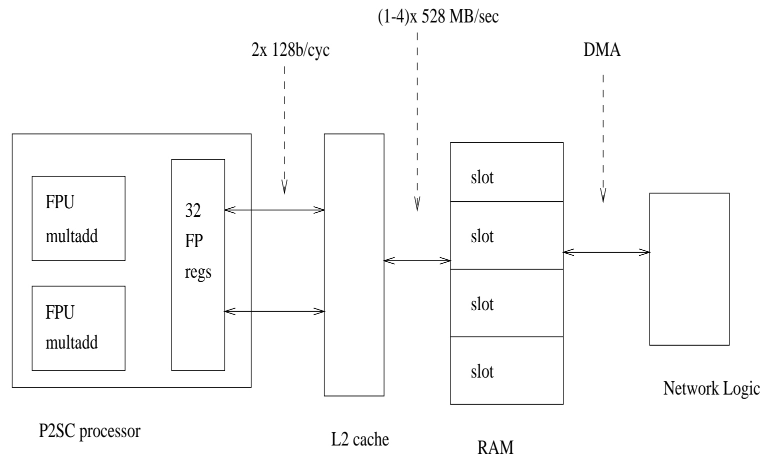


Figure 6.1: Architecture of an IBM SP processor.

The processors used are the Power2 Super Chips (P2SC), which can be either *thin* or *wide* depending on the difference in data bus width. At UH, the *thin* nodes operate at 120 MHz, while the *wide* nodes operate at 135 MHz. Memory per processor for the UH SP2 for *thin* nodes goes from 128 MB to 256 MB, whereas for the *wide* nodes it varies from 512 MB to 2 GB. The nodes are connected by Ethernet and a high performance switch, which has a peak bandwidth of 110 MB/sec/channel. All our codes were run on the thin nodes.

### 6.1.2 The IBM SP2 at SDSC

This SP2 at the San Diego Supercomputing Center (SDSC) has 128 160 Mhz Power2 Thin Nodes,each running AIX 4.2. The memory per processor is 256 MB. This machine also is capable of a peak bi-directional data transfer rate of 110 MB/second between each node pair.

### 6.1.3 The HP Exemplar at CACR

The HP Exemplar X-Class server installed at the Center for Advanced Computing Research (CACR), has 256 HP PA8000 RISC processors each running at 180MHz. Each processor has a 1 MB direct-mapped data cache and a 1 MB instruction cache. The 256-CPU system is composed of 16 nodes, where each node is an SMP arrangement of 16 processors as shown in Figure 6.2. Memory per node is 4 GB.

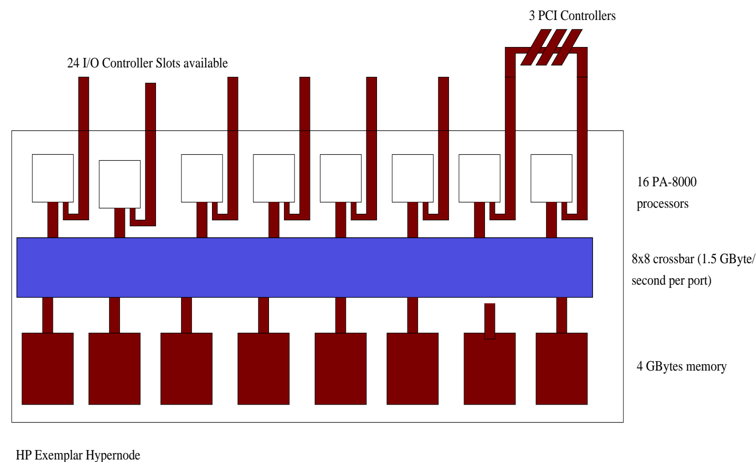


Figure 6.2: HP Exemplar node architecture.

The OS used is SPP-UX version 5.2. The entire system supports global addresses and is fully cache-coherent through hardware support. The nodes are connected using HP's coherent toroidal interconnect. Remote hypernode memory bandwidth is rated at 480 MB/s and local hypernode memory bandwidth is 1.5 GB/s.

### 6.1.4 The SGI Origin 2000 at the National Center for Supercomputing Applications (NCSA)

The Origin 2000 is a Scalable Shared Memory Multiprocessor (S2MP) and is also known as a Distributed Shared Memory (DSM) system. NCSA's current configuration consists of two 32 processor machines, one 64 processor machine and five 128 processor machines. A node consists of two R10000 CPU's (195 MHz/250 MHz), local caches, external cache for each CPU, local memory, I/O and interconnect interfaces and a HUB as shown in Figure 6.3.

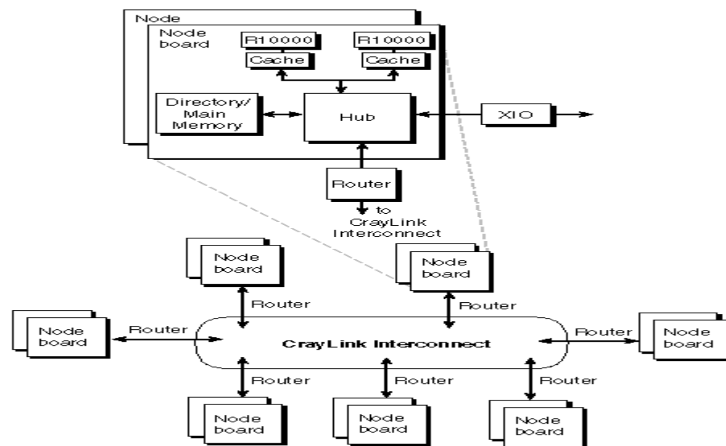


Figure 6.3: Modules in an SGI Origin 2000 system.

The HUB links the two CPU's, local memory, system interconnect, and the I/O subsystem. All our codes have been run on the machines which have 250 MHz R10000 processors running the IRIX 6.5.5 Operating System and having memory per node between 1 GB to 1.18 GB.

Local memory is memory on the requesting CPU's node board and remote memory is on another node board. The HUB to local memory peak bandwidth is 780

MB/s. Access to memory on a remote node takes longer in part because the request must be processed through two HUBS. Routers are used to connect two nodes to the Cray Link Interconnect system, which is responsible for linking two or more routers together. The Peak Interconnect bandwidth is 1.56 GB/s. A four processor system consists of two inter-connected two processor systems. Similarly, an eight processor system consists of two inter-connected four processor systems and so on. All our codes have been run on the machines which have 250 MHz R10000 processors running the IRIX 6.5.1 Operating System.

### 6.1.5 The Globus Metacomputing Toolkit

The Globus project is developing a software infrastructure for computations that integrate geographically distributed computational and information resources. The GUSTO testbed is being used to evaluate the performance of programs which use the Globus toolkit. The implementation uses a communications protocol called Nexus, which can run over various communication protocols. We have used the IBM SP2 at UH and the IBM SP2 at SDSC, as both sites participate in the GUSTO testbed.

## 6.2 Experimental Setup

### 6.2.1 Model Programs

The model programs were variants of the ping program. A message of a certain length was sent from the MPI program to the HPF program using our communication techniques and the Round Trip Time (RTT) and bandwidth were measured. 100 runs of this kind of communication were taken per processor pair.

Two cases were studied with respect to global and local data transfer. The terminology is based on the local data set size under scaling the number of processors.

- **Variable Data Size.** 8 MB of data was used for the programs running on a single processor and then this data was divided up evenly per MPI-HPF process pair as the number of processors increased. In this case

  *Speedup for n procs = (Time taken on 1 proc)/(Time taken on n procs)*
  *Parallel Efficiency for n procs = (Speedup for n procs)*100/n*

- **Constant Data Size.** A range of data from 128 bytes to 64 MB was kept constant per MPI-HPF process pair as the number of processors were increased.

  *Parallel Efficiency for n procs = (Time taken on 1 proc)*100/(Time taken on n procs)*

As 100 runs were taken of each ping-pong operation, three timing values were obtained for each processor, namely: *average value, minimum value and maximum value*. The *Average time* considered in the following sections indicates the longest average time over all processors. The standard deviation of the 100 timings is also reported.

### 6.2.2 Application Codes

Timings for data transfer in both the FMD as well as Anderson codes were recorded. 20 runs of the data transfer were taken to observe fluctuations in timings. Therefore, like the model program case, three timing values were obtained per processor: *average value, minimum value and maximum value.* The *Average time* considered indicates the longest average time over all processors. We have

*Speedup for n procs = (Time taken on 1 proc)/(Time taken on n procs)*
*Parallel Efficiency for n procs = (Speedup for n procs)*100/n*

In addition, both send and receive times were measured when sending/receieving from the FMD code to/from Anderson code and these times were compared with those of the model programs.

## 6.3 Communication with Sockets

With the UNIX domain protocols we use a pathname for mapping between the two processes. We evaluated the performance for both local and remote pathnames with respect to a particular processor/node. After a comparison of which protocol (UNIX domain or TCP/IP) performed better, the best one was used in all subsequent analysis.

### 6.3.1 The IBM SP2

Communicating MPI and HPF processes were placed on the same processor and therefore the programs were run using the *ip* mode on the IBM SP, which allow for two processes running on the same processor. Figure 6.4 shows that the UNIX domain protocols are almost twice as fast as the TCP/IP protocols. Furthermore having the socket name on the local or remote disk does not affect the performance in a great way as the difference between timings is not more than 10 %.

Thus, in the following we only consider UNIX domain sockets with pathnames on local disk.

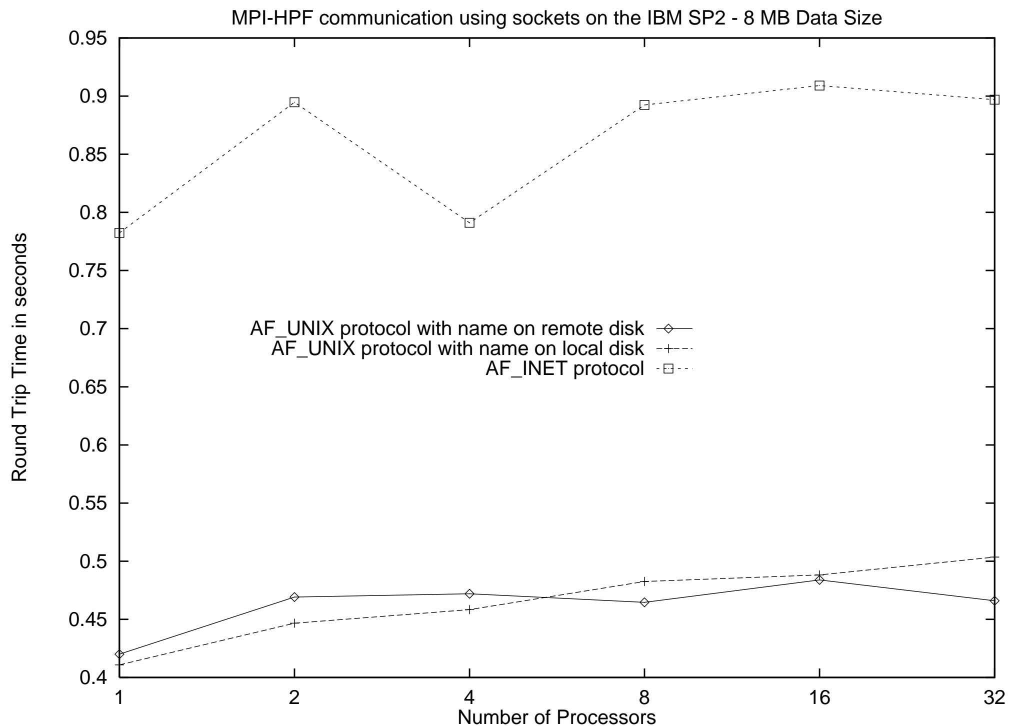


Figure 6.4: Various types of socket communication on the IBM SP2.

From Table 6.1 we see that for the Variable Data Size case we observe reasonably

MPI-HPF Round Trip Times with socket communication on IBM SP2
( Array size per proc: 8 MB /number_of_processors )

| No. of Procs | Bytes (one way) | Avg Time in secs (longest) | Standard Deviation | Parallel Efficiency | Real Efficiency |
|---|---|---|---|---|---|
| 1 | 8 MB | 0.443 | 0.0280 | 100.00 % | 24.83 % |
| 2 | 4 MB | 0.237 | 0.0162 | 93.45 % | 22.78 % |
| 4 | 2 MB | 0.121 | 0.0157 | 91.52 % | 22.97 % |
| 8 | 1 MB | 0.061 | 0.0087 | 90.77 % | 22.62 % |
| 16 | 512 KB | 0.031 | 0.0122 | 87.61 % | 22.15 % |
| 32 | 256 KB | 0.018 | 0.0163 | 75.23 % | 19.45 % |

Table 6.1: Socket communication on the IBM SP2 (Variable Data Size).

good speedup up to 32 processors. Real Efficiency is calculated using the following formula:

$$Real_Efficiency = \frac{Memory_copy_time}{(Socket_rtt/2)} * 100. \tag{6.1}$$

From Figure 6.5, we see that the memory copy is about four times faster than socket communication. Table 6.2 shows both peak and average MPI-HPF socket communication bandwidth. Figure 6.6 shows that variations in timings increase in the 16 and 32 processor case. We believe this is due to possible contention for socket buffers per processor. Figure 6.7 shows the curve for memory copy standard deviation rises substantially after 1 MB and then becomes flat till 4 MB and again increases when the data size is 8 MB. This behavior can be linked to the fact that the cache page size is 4 KB [28] and hence variations in memory copy timings increase by a factor of four.

The socket buffer contention effect is illustrated in Figure 6.8, which shows that there is a linear increase in the number of socket buffers with an increase in the

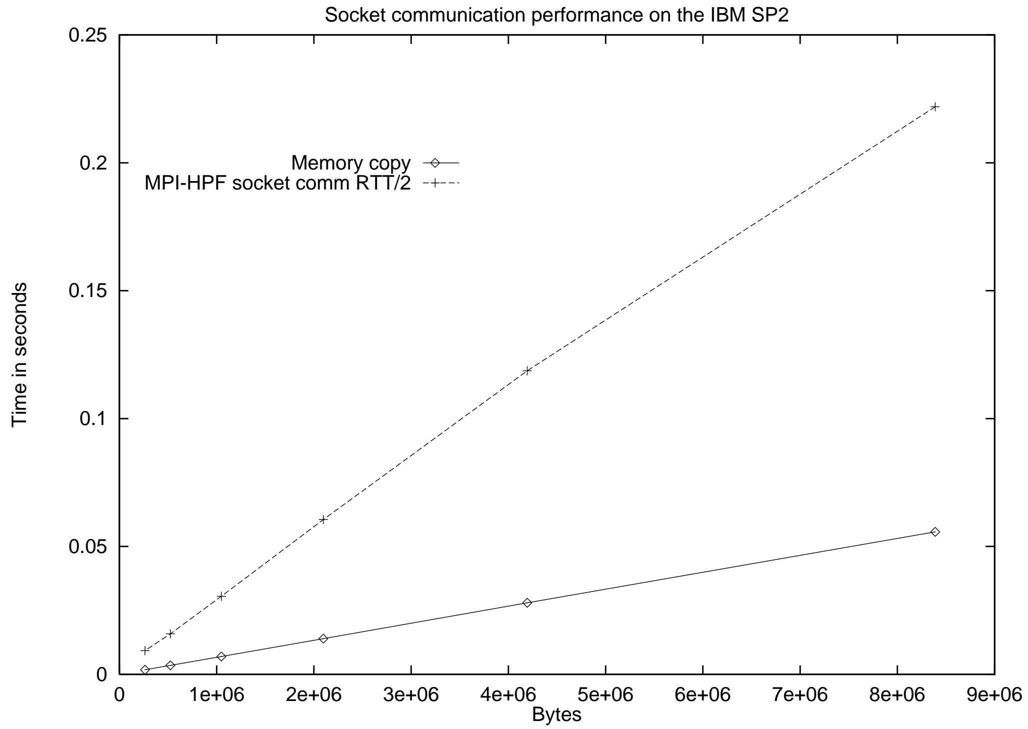


Figure 6.5: Memory copy vs. socket communication (Variable Data Size) on the IBM SP2.

Bandwidth for socket communication on the IBM SP2 with Variable Data Size

| No. of Procs | Bytes | Peak Bandwidth (Proc Num) | Avg. Bandwidth (Proc Num) |
|---|---|---|---|
| 1 | 8 MB | 39.34 MB/s (0) | 37.80 MB/s (0) |
| 2 | 4 MB | 39.98 MB/s (1) | 35.34 MB/s (0) |
| 4 | 2 MB | 40.27 MB/s (3) | 34.65 MB/s (0) |
| 8 | 1 MB | 42.31 MB/s (7) | 34.33 MB/s (0) |
| 16 | 512 KB | 42.13 MB/s (12) | 33.07 MB/s (0) |
| 32 | 256 KB | 42.03 MB/s (30) | 28.34 MB/s (0) |

Table 6.2: Socket communication bandwidth on the IBM SP2 (Variable Data Size).

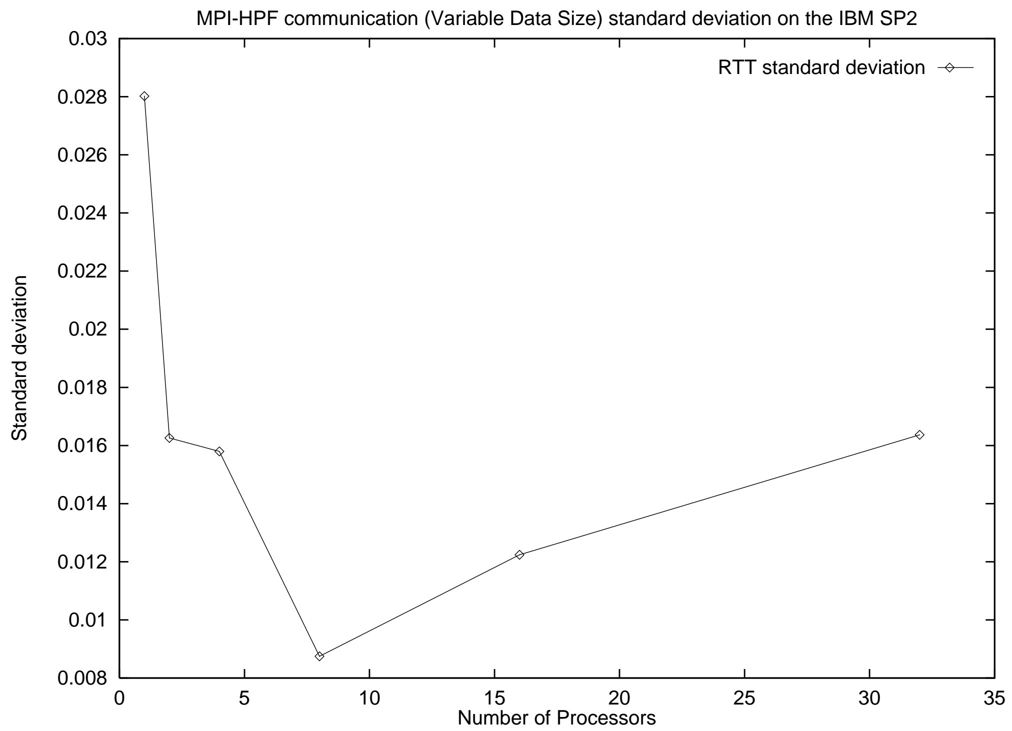


Figure 6.6: MPI-HPF RTT standard deviation (Variable Data Size per processor) on the IBM SP2.

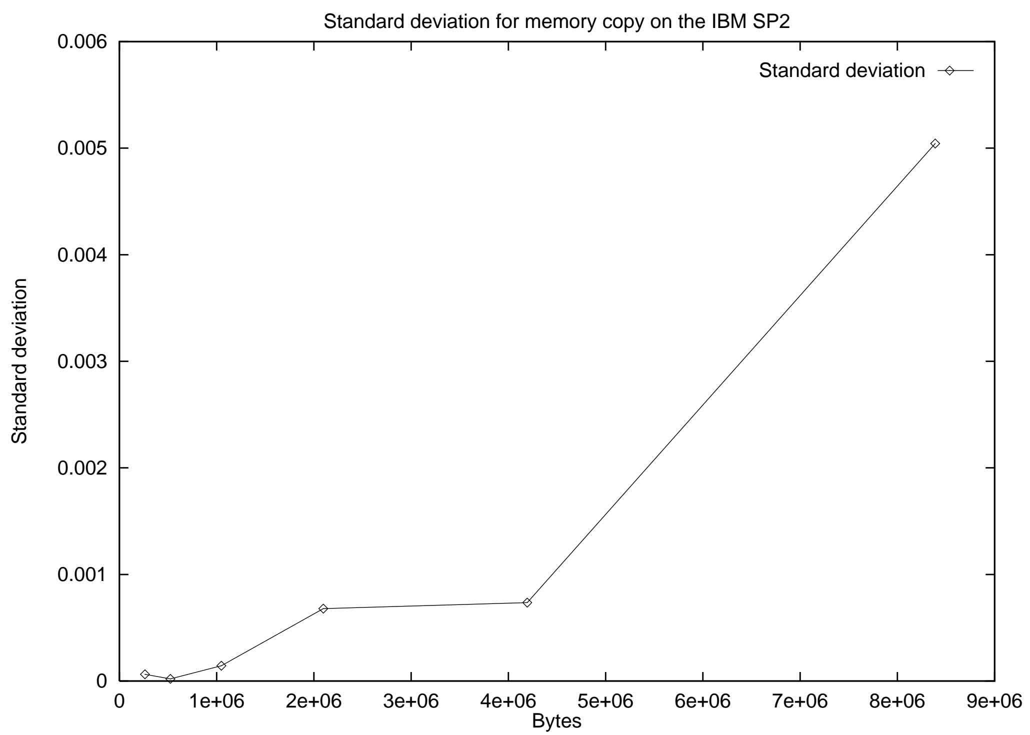


Figure 6.7: Memory copy standard deviation on the IBM SP2.

number of processors. Figure 6.8 shows that processor 0 allocates the largest number of buffers and hence, also explains Table 6.2, which shows that the lowest average bandwidth is always on processor 0.

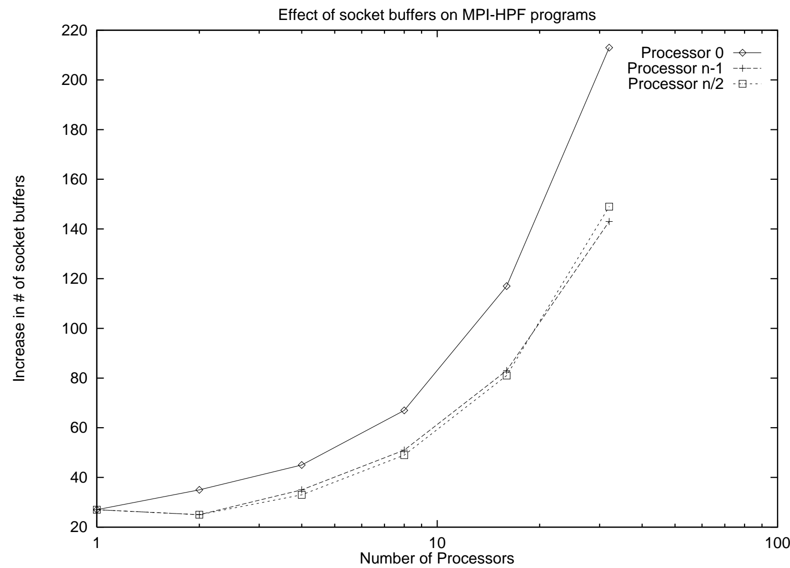


Figure 6.8: Increase in socket buffers on the IBM SP2.

For the constant data size per processor case (8 MB data size) shown in Table 6.3, parallel efficiency is better and the variations in timings are less (Figure 6.9). This can be attributed to the fact that the data sizes are significantly larger in the constant data size case. Thus, the overhead is relatively less compared to the data transfer time.

Figures 6.10 and 6.11 show communication for constant data sizes from zero bytes to 16 KB and from 16 KB to 24 MB respectively. The plot is split into two, as one can study the behavior of the code, while using smaller byte sizes in greater detail. We see that for smaller data sizes (128 bytes, 512 bytes, 2048 bytes), average time does not increase linearly with increase in data size.

MPI-HPF Round Trip Times with socket communication on the IBM SP2
(8 MB per MPI-HPF process)

| No. of Procs | Avg Time in secs (largest) | Standard Deviation | Parallel Efficiency | Real Efficiency |
|---|---|---|---|---|
| 1 | 0.443 | 0.0280 | 100.00 % | 22.94 % |
| 2 | 0.474 | 0.0266 | 93.45 % | 21.05 % |
| 4 | 0.481 | 0.0295 | 92.09 % | 20.74 % |
| 8 | 0.482 | 0.0300 | 91.90 % | 20.70 % |
| 16 | 0.469 | 0.0286 | 94.45 % | 21.27 % |
| 32 | 0.481 | 0.0380 | 92.09 % | 20.74 % |

Table 6.3: Socket communication on IBM SP2 (Constant Data Size).

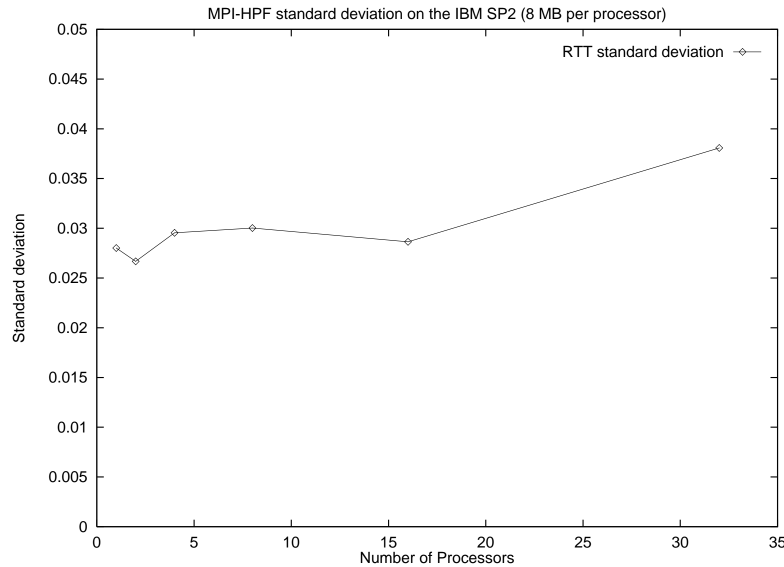


Figure 6.9: MPI-HPF RTT standard deviation (8 MB per processor) on the IBM SP2.

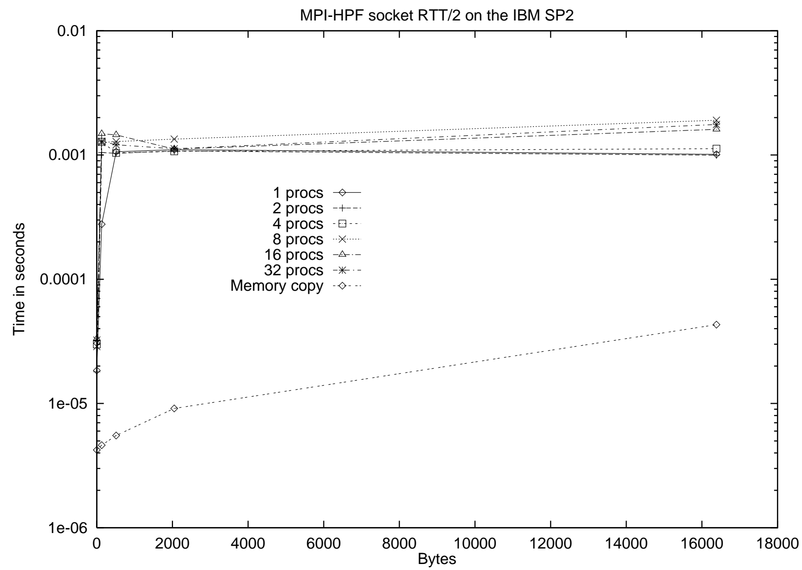


Figure 6.10: MPI-HPF socket RTT/2 for Constant Data Sizes per processor on the IBM SP2.

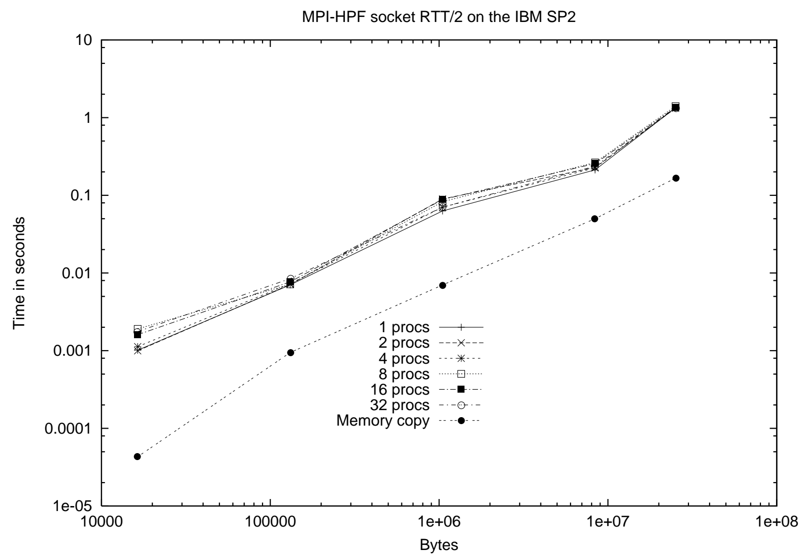


Figure 6.11: MPI-HPF socket RTT/2 for Constant Data Sizes per processor on the IBM SP2.

This behavior can be explained by Figures 6.12 and 6.13, which shows, for example, that while measuring round-trip times for data size of 128 bytes on a single processor a 100 times, we see that there are a few spikes (not more than 10) which increase the average time considerably. These spikes could be due to the fact that the OS needs to do a context switch between the two MPI/HPF processes thus affecting average timing for small data sizes. Peak real efficiency of 22.94 % was observed with one processor for a data size of 8 MB. From Figures 6.10 and 6.11 we can conclude that increasing the number of processors with constant data size increases the data transfer time. Figure 6.11 also verifies the timings in Table 6.1, where the MPI-HPF round-trip times in Table 6.1 are approximately double that of of those (RTT/2) in Figure 6.11 for the single processor case (8 MB data size) and the eight processor case (1 MB data size).

This happens especially for data sizes that are small. Thus in Table 6.1, parallel efficiency decreases as we decrease data size.

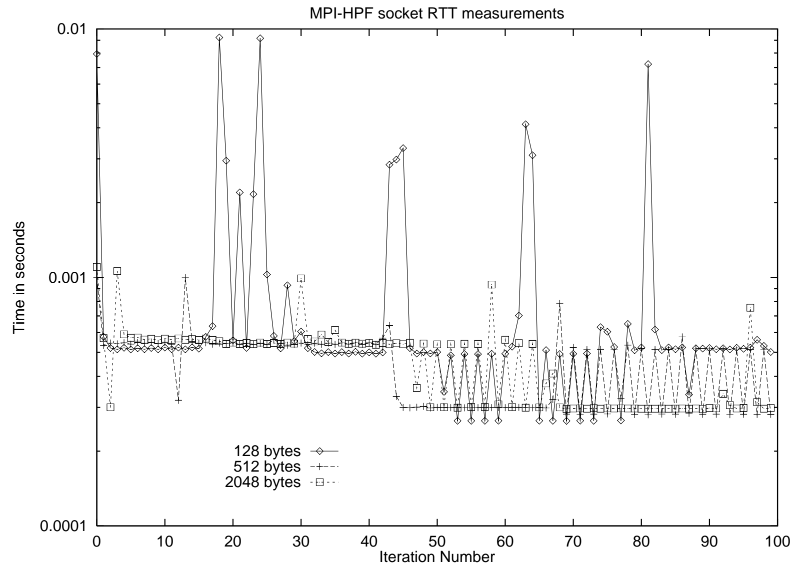


Figure 6.12: MPI-HPF socket RTT variations with Constant Data Size (128-2048 bytes) on one processor of the IBM SP2.

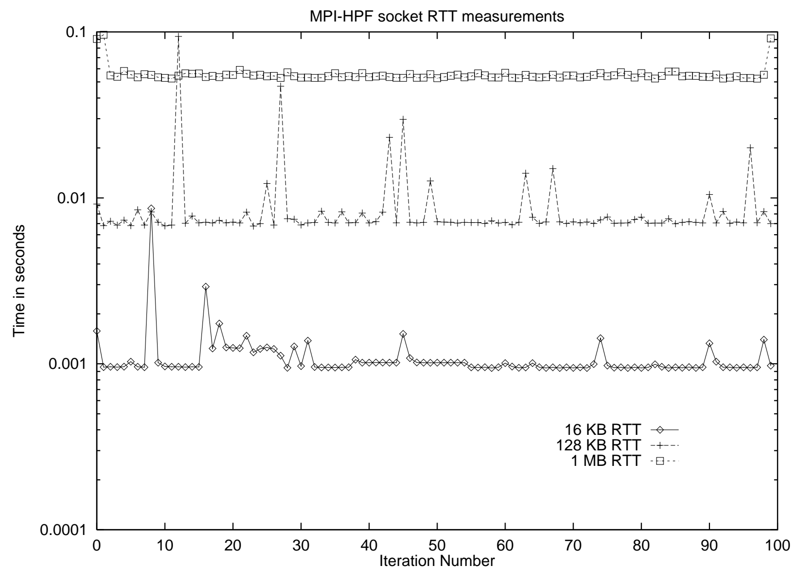


Figure 6.13: MPI-HPF socket RTT variations with Constant Data Size (16 KB-1 MB) on one processor of the IBM SP2.

### 6.3.2 The HP Exemplar

On the HP Exemplar we were not able to control the running of two processes (at the same time) on a single processor. Under SPP-UX, there is no way to control what processor a process is scheduled on. It's entirely up to the operating system to decide where to place a process. Also, during a process' lifetime, the operating system may schedule it among multiple processors.

The solution to this problem was to ask the job scheduler for the required number of processors and then run a serial program which would spawn two "mpirun" processes. These processes would be run on the set of processors given by the operating system. Using the pathname and rank the MPI and HPF processes were able to communicate with each other. The SPP-UX operating system was able to handle inter-process communication across different processors.

Figure 6.14 shows that although there is a less than 9 % difference in execution time for UNIX domain sockets with the name on the local or a remote disk, the Internet protocols are more than twice as fast. This behavior has been confirmed by engineers from HP.

So hereafter we refer to only the performance of Internet protocol sockets.

Real Efficiency is calculated as follows:

$$Real_Efficiency = \frac{Memory_copy_time}{(Socket_rtt/2)} * 100. \tag{6.2}$$

Table 6.4 shows that we observe reasonably good speedup up to eight processors. For 16 processors the operating system has to run the 32 MPI and HPF processes on the node having 16 processors in addition to system processes, whereas for lower processor number cases (less than 16), system processes can be run on the idle

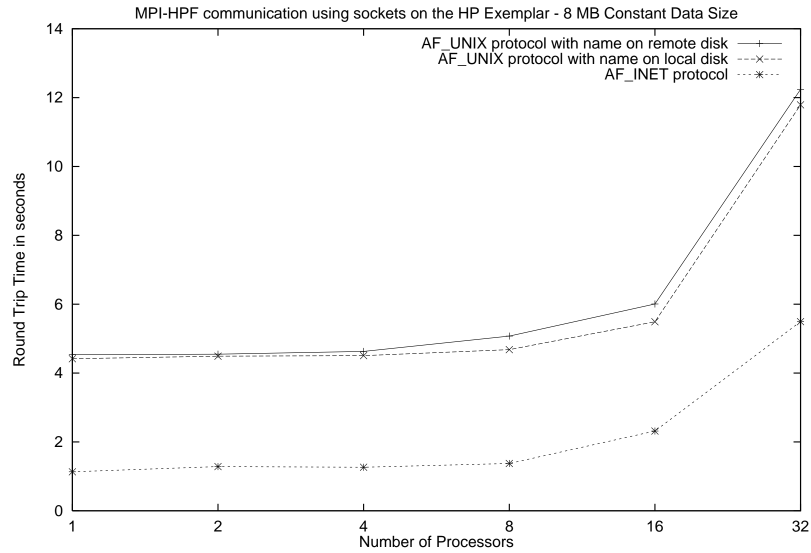


Figure 6.14: Various types of socket communication on the HP-Exemplar.

MPI-HPF Round Trip Times with Socket communication on HP-Exemplar
( Array size: 8 MB /number_of_processors )

| No. of Procs | Bytes (one way) | Avg Time in secs (longest) | Standard Deviation | Parallel Efficiency | Real Efficiency |
|---|---|---|---|---|---|
| 1 | 8 MB | 1.18 | 0.363 | 100.00 % | 10.74 % |
| 2 | 4 MB | 0.59 | 0.130 | 98.50 % | 11.05 % |
| 4 | 2 MB | 0.40 | 0.086 | 72.66 % | 9.16 % |
| 8 | 1 MB | 0.23 | 0.062 | 63.85 % | 9.00 % |
| 16 | 512 KB | 0.15 | 0.082 | 46.67 % | 4.05 % |
| 32 | 256 KB | 0.20 | 0.301 | 17.98 % | 0.54 % |

Table 6.4: Socket communication on the HP Exemplar (Variable Data Size).

processors.

Table 6.5 shows the peak and average socket communication bandwidth and Figure 6.15 shows a plot of the memory copy vs. MPI-HPF communication with sockets.

Bandwidth for socket communication on the HP Exemplar with Variable Data Size

| No. of Procs | Bytes | Peak Bandwidth (Proc Num) | Avg. Bandwidth (Proc Num) |
|---|---|---|---|
| 1 | 8 MB | 10.40 MB/s (0) | 7.08 MB/s (0) |
| 2 | 4 MB | 9.10 MB/s (0) | 7.02 MB/s (0) |
| 4 | 2 MB | 7.26 MB/s (1) | 5.16 MB/s (2) |
| 8 | 1 MB | 6.80 MB/s (3) | 4.52 MB/s (6) |
| 16 | 512 KB | 6.72 MB/s (10) | 3.30 MB/s (0) |
| 32 | 256 KB | 7.03 MB/s (1) | 1.26 MB/s (2) |

Table 6.5: Socket communication bandwidth on the HP Exemplar (Variable Data Size ).

For the 32 processor case the longest time taken is between processes on different hypernodes and hence the operating system has to instantiate inter-node communication and therefore the time taken increases substantially. We note that even though we calculate the Real Efficiency based on the hypernode peak memory bandwidth of 480 MB/s, the Real Efficiency is less than 1 %. On closer inspection of the running of processes, we observed that MPI processes that were running corresponding HPF processes residing on another hypernode. Therefore, when the processes at each hypernode try to communicate with the processes on the other hypernode interference amongst processes for using communication resources increases the time taken substantially. Thus, we see that Parallel Efficiency drops in the 16 and 32 processor cases and variations in timings increase (Figure 6.16).

From Table 6.6, we see that time taken for transfer of the same amount of data

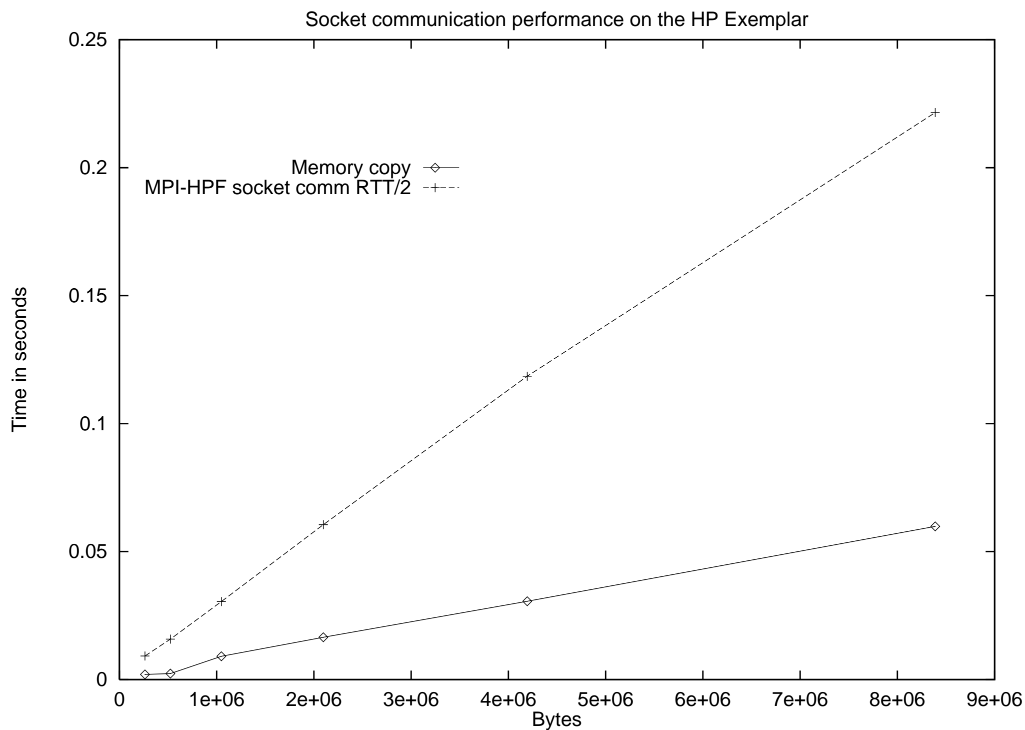


Figure 6.15: Memory copy vs. socket communication on the HP Exemplar.

does not increase significantly till the 16 and 32 processor cases. This behavior can also be attributed to the reasons listed for the variable data size.

From Figure 6.17 and Figure 6.18, we observe that the variations in timings are less in the constant data size case (8 MB data size) than in the variable data size case because of the fact that the overhead has a lesser effect in the constant data size case as data sizes are significantly larger.

In addition, Figures 6.17 and 6.18 show that there are large variations in timings in the 32 processor case as communication takes place across hypernodes and other traffic may cause delays from time to time.

Figures 6.19 and 6.20 show communication for constant data sizes from zero

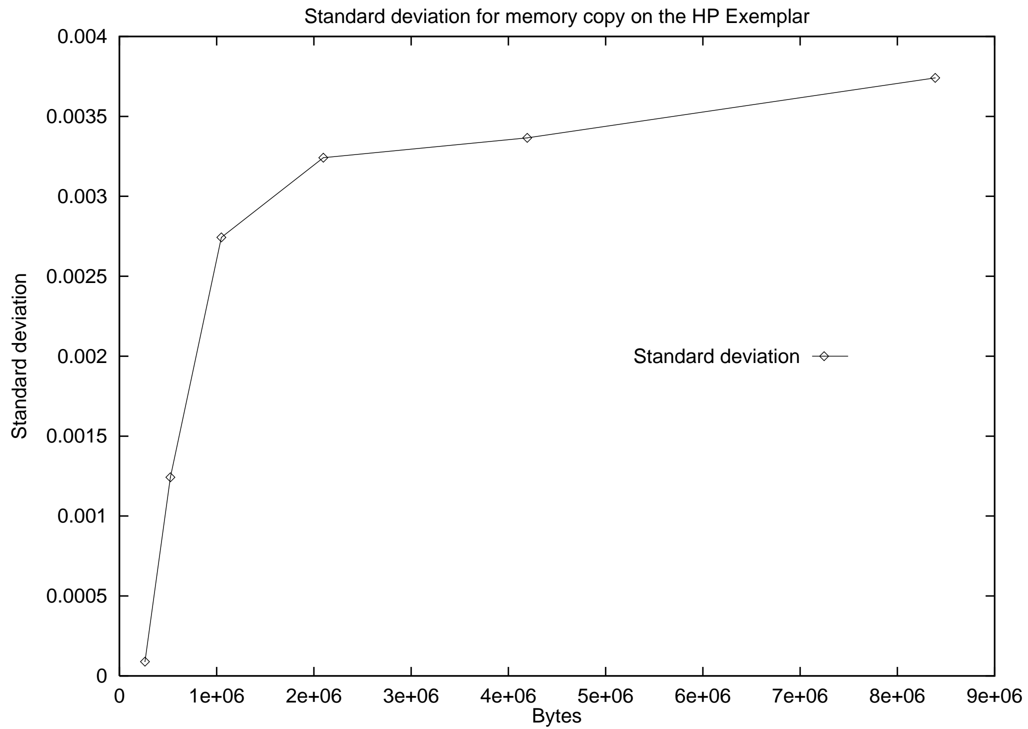


Figure 6.16: Memory copy standard deviation on the HP Exemplar.

MPI-HPF Round Trip Times with socket communication on the HP Exemplar
(8 MB per MPI-HPF Process)

| No. of Procs | Avg. Time in secs (longest) | Standard Deviation | Parallel Efficiency | Real Efficiency |
|---|---|---|---|---|
| 1 | 1.18 | 0.363 | 100.00 % | 11.43 % |
| 2 | 1.63 | 0.593 | 72.39 % | 7.77 % |
| 4 | 1.88 | 0.400 | 62.76 % | 6.74 % |
| 8 | 1.82 | 0.362 | 64.83 % | 6.96 % |
| 16 | 2.29 | 0.218 | 51.52 % | 5.53 % |
| 32 | 5.62 | 2.610 | 20.99 % | 1.27 % |

Table 6.6: Socket communication on HP Exemplar (Constant Data Size).

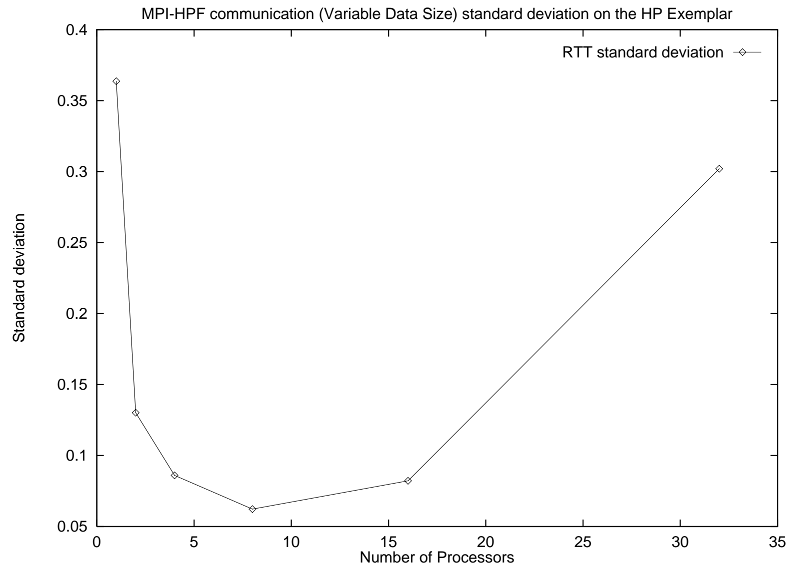


Figure 6.17: MPI-HPF socket RTT standard deviation (Variable Data Size) on the HP Exemplar.

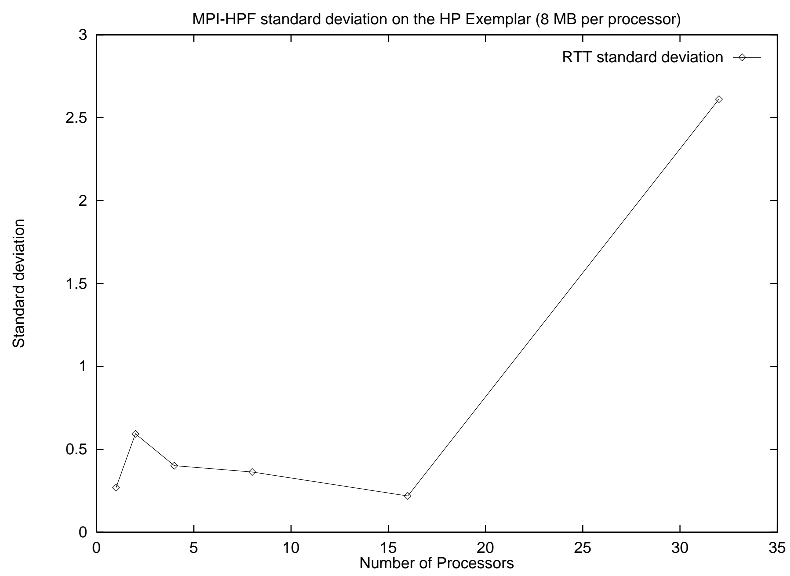


Figure 6.18: MPI-HPF socket RTT standard deviation (8 MB per processor) on the HP Exemplar.

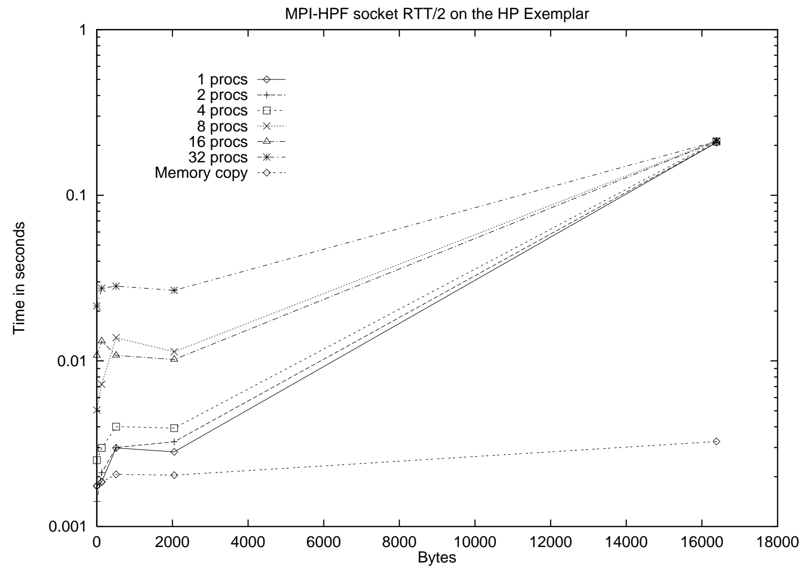


Figure 6.19: MPI-HPF socket RTT/2 for Constant Data Sizes per processor on the HP Exemplar.

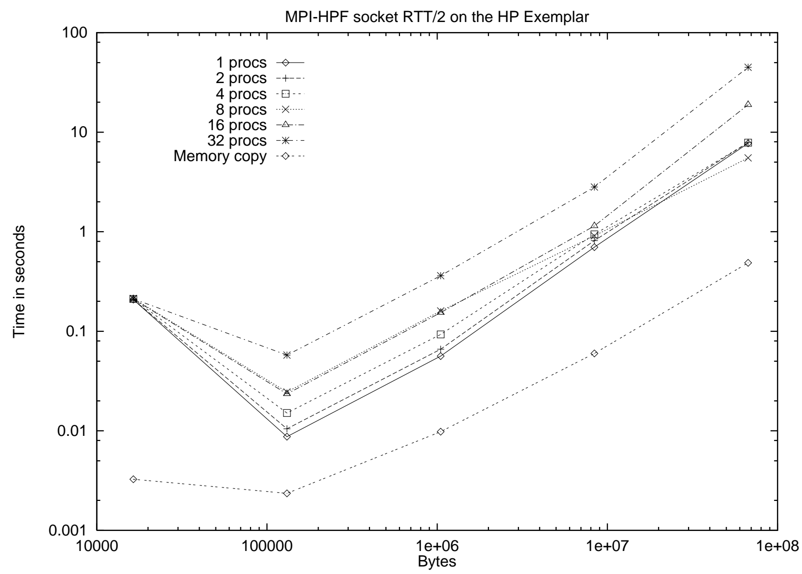


Figure 6.20: MPI-HPF socket RTT/2 for Constant Data Sizes per processor on the HP Exemplar.

bytes to 16 KB and from 16 KB to 64 MB respectively. The Round Trip Time (RTT) increases almost by two orders of magnitude when the data size is 16 KB and then reduces back by two orders of magnitude when the data size is around 32 KB. Discussions with HP have revealed that 16 KB affects the RTT because it is the boundary of the buffer used. Below 16 KB there is no new allocation of bigger buffers and above it there are. So the increase in time is a delay due to extra memory allocation.

For smaller data size cases (128 bytes, 512 bytes, 2 KB), we see large fluctuations in timings for multiple runs as seen in Figure 6.21.

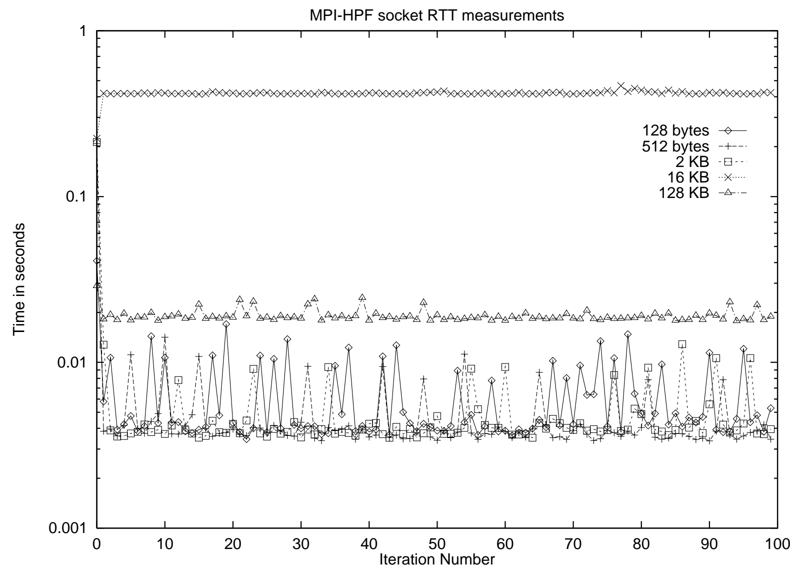


Figure 6.21: MPI-HPF socket RTT variations with Constant Data Size (128 bytes -128 KB) on one processor of the HP Exemplar.

In this case, these large fluctuations could be attributed to the dynamic rescheduling of processes across processors by the operating system. Peak real efficiency of 99.51% was observed with one processor for a data size of 128 bytes.

From Figures 6.19 and 6.20 we can conclude that increasing the number of processors with constant data size affects the data transfer time especially as the size of data decreases. For the 32 processor case as we effectively run 64 processes (32 MPI and 32 HPF processes) the longest data transfer time is generally between two hyper-nodes, as per the process scheduling techniques of the operating system. The above behavior is reflected in Table 6.4, which shows decreasing Parallel Efficiency with decreasing data size per processor. However, by comparing Table 6.4 and Table 6.6, we see that bigger data sizes reduce Parallel Efficiency. This could be explained by the fact that as more data is transmitted, the probability of the communicating processes getting rescheduled by the operating system increases, thus introducing additional overhead in our timings.

Figure 6.20 also verifies the timings in Table 6.4, where the MPI-HPF round-trip times in Table 6.4 are approximately double that of of those (RTT/2) in Figure 6.20 for the single processor case (8 MB data size) and the eight processor case (1 MB data size).

### 6.3.3 The SGI Origin 2000

The SGI Origin 2000 allows for only a single process on a processor at a time in batch mode. Hence, we run our MPI-HPF programs on double the number of processors compared to the previous cases (IBM, HP). In this case too, like for the HP Exemplar, we wrote a program to invoke the script that would run both the MPI and HPF programs. This program would be given as input to the job scheduler along with the required number of processors.

All jobs submitted were run in time-shared mode. Thus, other processes might have been running on the machine at the "same time".

Figure 6.22 shows that the UNIX domain protocols are more than twice as fast as the Internet protocols on the SGI Origin 2000. Thus, we discuss the results only for the UNIX domain protocols.

Real Efficiency is calculated as follows:

$$Real_Efficiency = \frac{Memory_copy_time}{(Socket_rtt/2)} * 100. \tag{6.3}$$

Table 6.7 shows that we achieve good parallel efficiency up to 32 processors. Note that because of the restrictions on the process to processor mapping for the SGI Origin 2000, the processor column in Table 6.7 has twice the number of processors compared to Tables 6.1 and 6.4 for comparable cases. Figure 6.23 shows how the MPI-HPF socket communication compares to a memory copy operation.

Parallel Efficiency greater than 100 % in the four processor case can be explained by the fact that the communicating processors could be closer in the four processor case and hence the increase in efficiency. The communicating processes may be on the same or different node boards or may have to communicate across different SMP clusters. Observations of multiple runs of the same experiment shows that there is

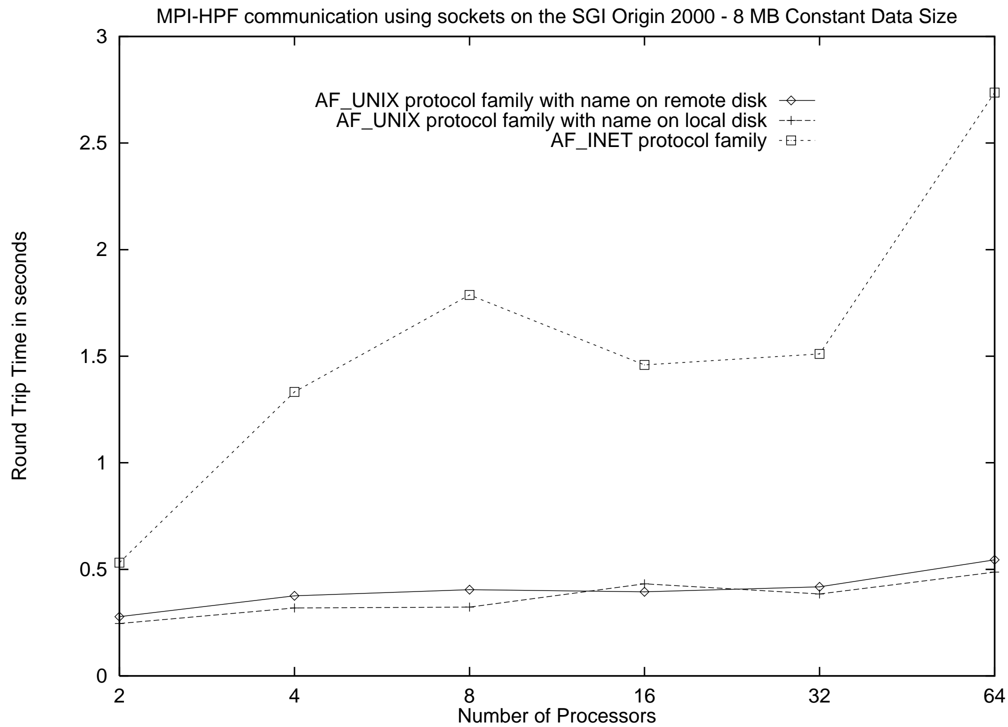


Figure 6.22: Various types of socket communication on the SGI Origin 2000.

MPI-HPF Round Trip Times with socket communication on the SGI Origin 2000
( Array size per proc: 8 MB /number_of_processors )

| No. of Procs | Bytes (one way) | Avg Time in secs (longest) | Standard Deviation | Parallel Efficiency | Real Efficiency |
|---|---|---|---|---|---|
| 2 | 8 MB | 0.220 | 0.0156 | 100.00 % | 48.72 % |
| 4 | 4 MB | 0.109 | 0.0073 | 100.91 % | 50.82 % |
| 8 | 2 MB | 0.062 | 0.0033 | 87.85 % | 18.08 % |
| 16 | 1 MB | 0.030 | 0.0039 | 89.57 % | 13.68 % |
| 32 | 512 KB | 0.018 | 0.0028 | 73.92 % | 10.75 % |
| 64 | 256 KB | 0.011 | 0.0084 | 59.26 % | 9.54 % |

Table 6.7: Socket communication on the SGI Origin 2000 (Variable Data Size).

not more than a 5-10 % difference in the values over different runs.

Figure 6.24 shows that the standard deviation increases for the 16, 32 and 64 processor cases, although the number of bytes being transferred decreases with an increase in the number of processors. This effect could again be attributed to the fact that each 128 node Origin 2000 machine is made of SMP clusters of 4, 8, 16, 32 and 64 processor systems and hence variations in timings increase as we go across clusters. The clustering effect also explains the decline in real efficiency shown in Figure 6.23, as we have compared our communication with a simple memory copy.

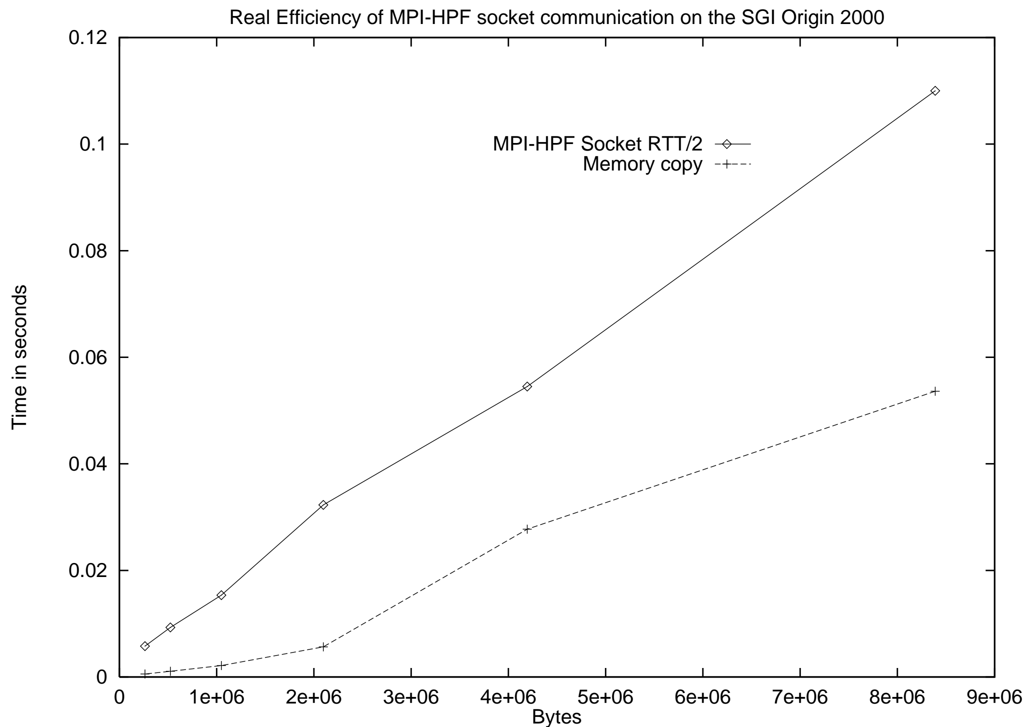


Figure 6.23: Memory copy vs. socket communication on the SGI Origin 2000.

Table 6.8 shows both peak and average MPI-HPF socket communication bandwidth. For the constant data size case (8 MB data size), Table 6.9 shows that the

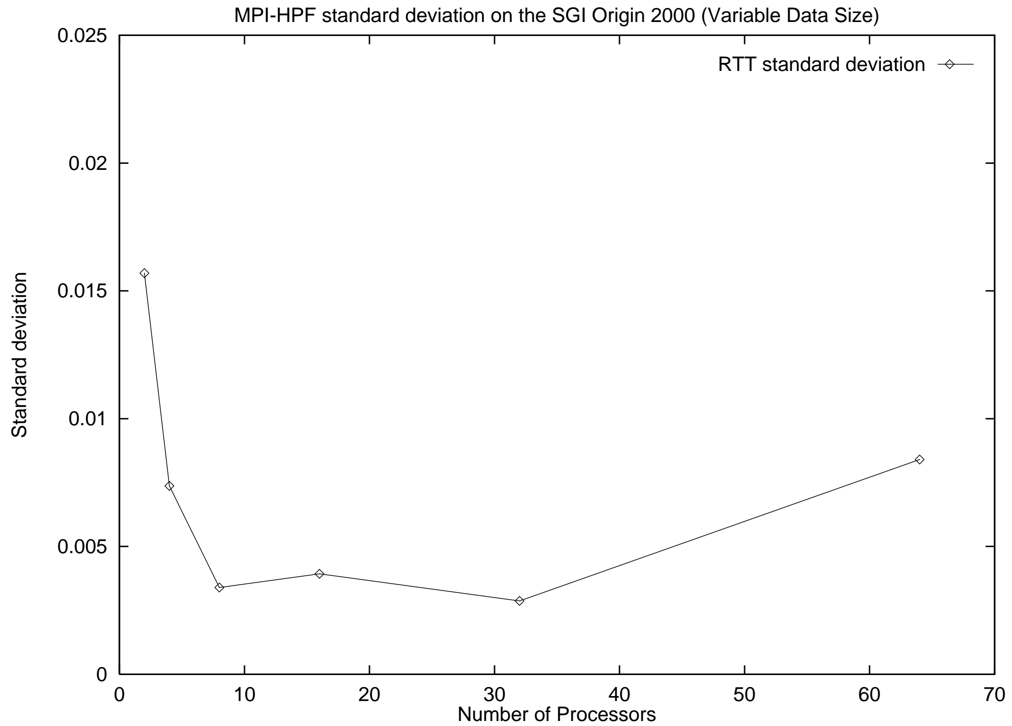


Figure 6.24: MPI-HPF socket RTT standard deviation (Variable Data Size) on the SGI Origin 2000.

Bandwidth for socket communication on the SGI Origin 2000 with Variable Data Size

| No. of Procs | Bytes | Peak Bandwidth (Proc Num) | Avg. Bandwidth (Proc Num) |
|---|---|---|---|
| 2 | 8 MB | 77.07 MB/s (0) | 76.26 MB/s (0) |
| 4 | 4 MB | 84.38 MB/s (0) | 76.95 MB/s (1) |
| 8 | 2 MB | 71.90 MB/s (3) | 67.00 MB/s (0) |
| 16 | 1 MB | 83.22 MB/s (2) | 68.31 MB/s (4) |
| 32 | 512 KB | 92.12 MB/s (15) | 56.37 MB/s (11) |
| 64 | 256 KB | 77.97 MB/s (23) | 45.19 MB/s (13) |

Table 6.8: Socket communication bandwidth on the SGI Origin 2000 (Variable Data S ize).

MPI-HPF Round Trip Times with socket communication on the SGI Origin 2000
(8 MB per MPI-HPF process)

| No. of Procs | Avg Time in secs (longest) | Standard Deviation | Parallel Efficiency | Real Efficiency |
|---|---|---|---|---|
| 2 | 0.220 | 0.0156 | 100.00 % | 48.72 % |
| 4 | 0.234 | 0.0150 | 94.01 % | 45.81 % |
| 8 | 0.295 | 0.0241 | 74.57 % | 36.33 % |
| 16 | 0.252 | 0.0207 | 87.30 % | 42.53 % |
| 32 | 0.325 | 0.0226 | 67.69 % | 32.98 % |
| 64 | 0.317 | 0.1820 | 69.40 % | 33.81 % |

Table 6.9: Socket communication on the SGI Origin 2000 (Constant Data Size).

efficiency is better than that in the variable data size case for 64 processors, because overhead is not significant compared to the data transfer time. In addition, the variations in timings are less than in the variable data size case, as seen from Figures 6.24 and 6.25. Each R10000 processor in the SGI Origin 2000 can have a 1 MB or 4 MB secondary cache in addition to a 32 KB on-chip set associative primary instruction and data caches. For a memory copy with a data size of 4 MB and a secondary cache of size 4 MB, the time to swap data in and out of the cache may increase the variations in timings. This could be a possible explanation for the anomalous behavior in Figure 6.25 for the 4 MB memory copy.

Figure 6.24 shows that the standard deviation decreases with the number of processors as expected, since the data sent per processor decreases with increasing numbers of processors. However, after 8 processors the standard deviation is almost constant till 32 processors and then shoots up significantly at 64 processors.

This can also be explained by the fact that the SGI Origin 2000 is made up of clusters of processors as explained earlier. The figure shows us that the clustering effect does become significant when we use more than 8 processors. Figure 6.26

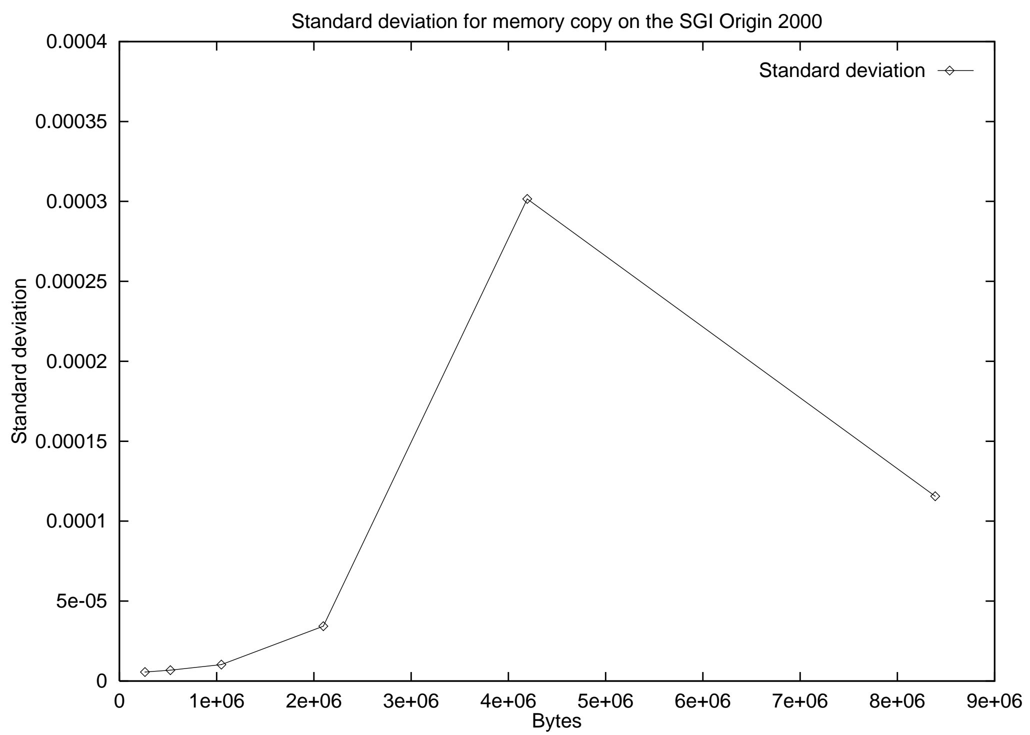


Figure 6.25: Memory copy standard deviation on the SGI Origin 2000.

shows the MPI-HPF socket standard deviation.

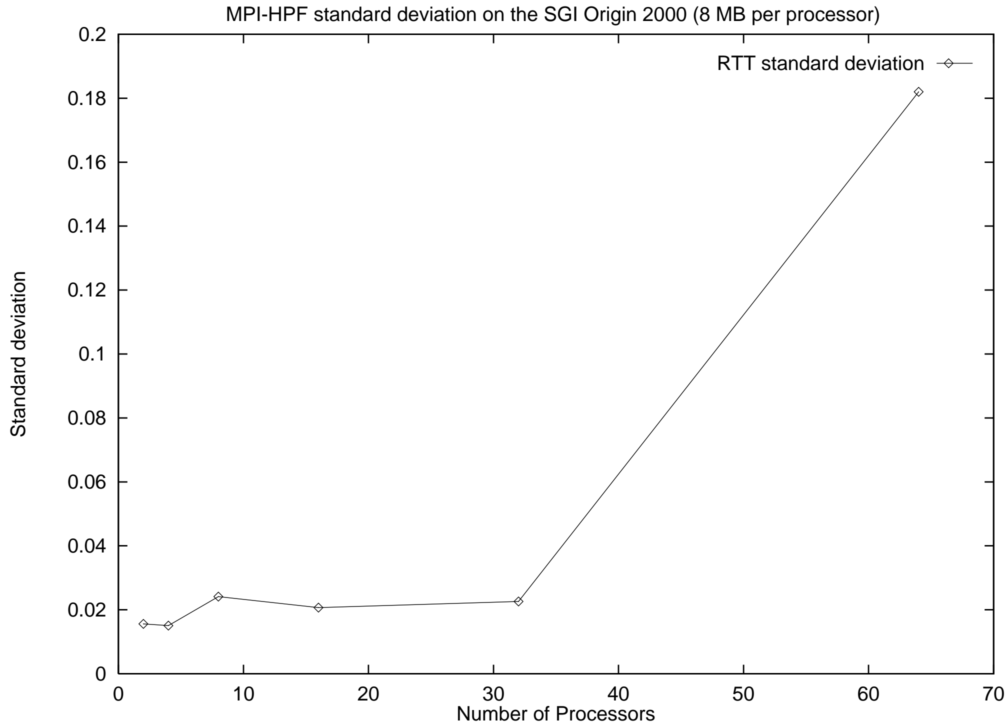


Figure 6.26: MPI-HPF socket RTT standard deviation (8 MB per processor) on the SGI Origin 2000.

Figures 6.27 and 6.28 show communication for constant data sizes from zero bytes to 16 KB and from 16 KB to 64 MB respectively. These two figures illustrate that for the 2, 4, 8, 16 and 32 processor cases, increase in number of processors may not necessarily increase or decrease the data transfer time. This could be because of the fact that communicating processors may have shorter/longer paths in the same cluster, across different processor cases. The 64 processor case however always takes more time for data transfer of the same data size.

Figure 6.29 shows that for two processors communicating with a data size of

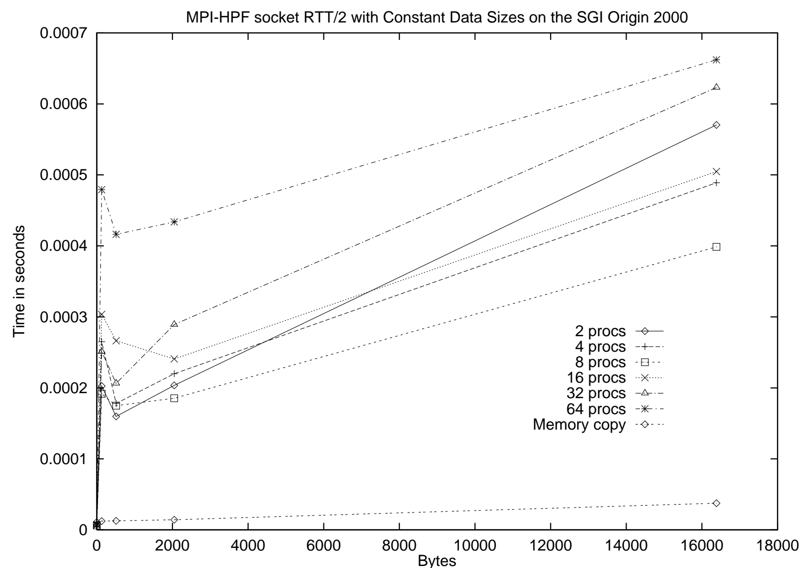


Figure 6.27: MPI-HPF socket RTT/2 for Constant Data Sizes per processor on the SGI Origin 2000.

128 bytes, the fluctuations in timings across multiple runs causes the average data transfer time to go up. This behavior is similar to that of the HP Exemplar and can be attributed to the dynamic rescheduling of processes across processors.

Figure 6.28 also verifies the timings in Table 6.7, where the MPI-HPF round-trip times in Table 6.7 are approximately double that of of those (RTT/2) in Figure 6.28 for the two processor case (8 MB data size) and the 16 processor case (1 MB data size).

A peak Real Efficiency of 67.06 % was observed with two processors for a data size of 8 MB.

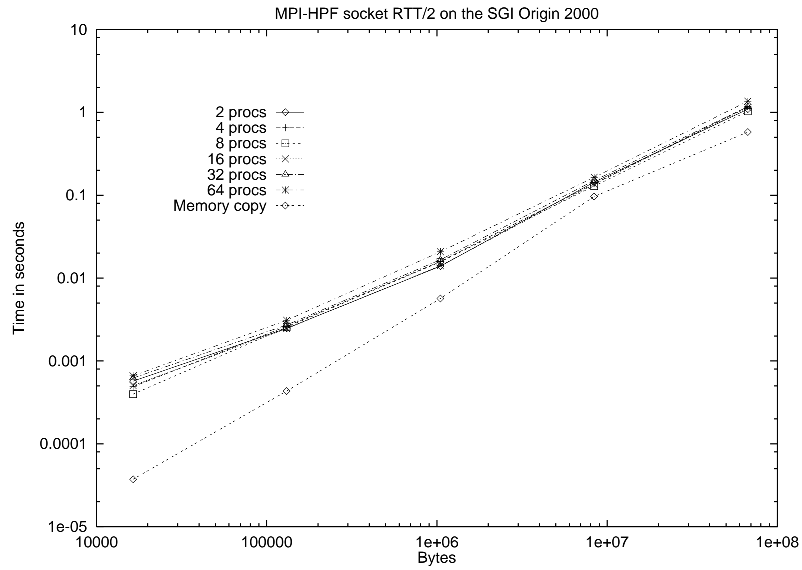


Figure 6.28: MPI-HPF socket RTT/2 for Constant Data Sizes per processor on the SGI Origin 2000.

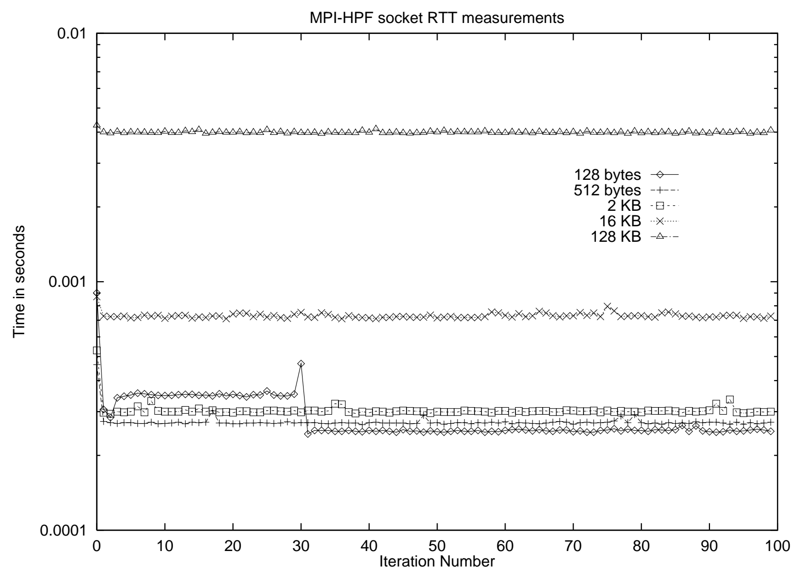


Figure 6.29: MPI-HPF socket RTT variations with Constant Data Size (128 bytes-128 KB) on two processors of the SGI Origin 2000.

### 6.3.4 Summary of Socket Communication

After studying the performance of MPI-HPF socket communication on different architectures, we can say that this type of communication will be most suitable in environments, where any pair of communicating MPI-HPF processes will cause zero or minimal interference to other communicating MPI-HPF processes. The IBM SP2 provides the above kind of environment, as our proposed model of MPI-HPF communication is limited to the processor on which the MPI and HPF processes are running. In addition, one can control how many user-level processes one wants to run on a particular processor on the IBM SP2. On the HP Exemplar, since process to processor scheduling is done entirely by the Operating System, communicating pairs of processes interfere with each other both within and across hyper-nodes. The SGI Origin 2000 also controls placement of processes on processors. However the rules are much stricter as only one process is allowed to run on a processor. In addition, as the SGI Origin 2000 is made up of clusters in powers of two, if communicating processes are on the same cluster then they will affect only other processes on that cluster.

For the Variable Data Size case, Tables 6.1, 6.4 and 6.9 show that the IBM SP2 has the best parallel efficiency and reasonably consistent Real Efficiency (within 3 %) upto 32 processors. It is followed by the SGI Origin 2000, which has good parallel efficiency upto the 32 processors. However, Real Efficiency in the SGI Origin 2000 decreases considerably (by about 39 %) from 2 processors to 64 processors. The HP Exemplar has a rapid decrease in both Parallel and Real Efficiency after 2 processors, as increasing numbers of communicating processes affect each other and hence the longest time.

For the Constant Data Size case, Tables 6.3, 6.6 and 6.7 show that the IBM SP2 is again the winner in terms of Parallel Efficiency and consistency of Real Efficiency across all processor cases. Here, the HP Exemplar shows an improvement over the Variable Data Size case in terms of Parallel Efficiency but Real Efficiency still drops with increasing number of processors. The SGI Origin 2000 shows an increase in Parallel Efficiency for 64 processors and has better consistency in Real Efficiency (within 15 % from 2 to 64 processors) than in the Variable Data Size case.

## 6.4 Communication with MPI

During the study of this mechanism, in addition to studying communication for fixed and variable data sizes, the MPI-HPF communication was also compared with equivalent MPI communication, where appropriate.

1. **MPI Communication between one pair of processes.** 8 MB was sent from one MPI process to the other with one process per processor. In order to study the performance sensitivity to the number of processors from which the communicating pair was selected, we varied the number of processors even though communication only took place between a single pair.

2. **MPI Communication with all processes communicating pairwise.** This case is a simulation of the MPI-HPF communication using MPI. The process set is split into two. Each process in one half sends data to precisely one process in the other half, which receive data from precisely one process in the first half. Then, the processes in the second half send back the data to the process in the first half from which it received data and all the processes in the first half receive data from its "partner". The objective of this experiment was to study potential network contention and its effect on MPI performance.

### 6.4.1 The IBM SP2

We have two cases under consideration for doing MPI-HPF communication.

**Communicating Processes on different Processors.**

The *poe* command on the IBM SP2 provides means for invoking normal MPI MPMD programs. Therefore, this facility was used to run our MPI-HPF communication, and the *poe* command were run using IBM's LoadLeveler. The user space protocol was used for running each MPMD job.

At first we discuss the MPI-HPF communication (with all MPI and HPF processes communicating) for constant and variable data sizes (Tables 6.10, 6.11, 6.12 and Figures 6.30, 6.31, 6.32, and 6.33) and then we compare these results with MPI communication between one pair of processes and MPI communication between all processes communicating pairwise (Figures 6.34 and 6.35).

We observe good efficiency up to 32 processors, as seen in Tables 6.10 and 6.12 for both the variable data size case and the constant data size case. Real Efficiency in this case is calculated by taking note of the fact that the peak performance of the SP2 high-performance switch is 110 MB/s as follows:

$$Real_Efficiency = \frac{Number_of_bytes}{110 * 10^6 * (MPI_rtt/2)} * 100. \quad (6.4)$$

Table 6.11 shows both peak and average bandwidth for this type of MPI-HPF communication.

The High-Performance Switch topology [29] of the IBM SP2 ensures that the available bandwidth between any pair of communicating nodes remains constant irrespective of where in the topology the two nodes lie. Thus, in Table 6.10 for the

MPI-HPF RTT with MPI communication on IBM SP2 (communicating processes on different processors)
( Array size: 8 MB /number_of_processors )

| No. of Procs | Bytes (one way) | Avg Time in secs (longest) | Standard Deviation | Parallel Efficiency | Real Efficiency |
|---|---|---|---|---|---|
| 2 | 8 MB | 0.218 | 0.0125 | 100.00 % | 66.05 % |
| 4 | 4 MB | 0.110 | 0.0060 | 99.09 % | 65.45 % |
| 8 | 2 MB | 0.054 | 0.0015 | 100.92 % | 66.66 % |
| 16 | 1 MB | 0.029 | 0.0045 | 93.96 % | 62.06 % |
| 32 | 512 KB | 0.019 | 0.0003 | 70.23 % | 46.39 % |

Table 6.10: Communication with MPI on the IBM SP2 (Variable Data Size).

Bandwidth for MPI communication on the IBM SP2 (communicating processes on different processors)
( Array size: 8 MB /number_of_processors )

| No. of Procs | Bytes | Peak Bandwidth (Proc Num) | Avg. Bandwidth (Proc Num) |
|---|---|---|---|
| 2 | 8 MB | 78.85 MB/s (1) | 76.95 MB/s (1) |
| 4 | 4 MB | 78.96 MB/s (3) | 76.26 MB/s (3) |
| 8 | 2 MB | 79.18 MB/s (6) | 75.70 MB/s (5) |
| 16 | 1 MB | 77.04 MB/s (12) | 72.31 MB/s (8) |
| 32 | 512 KB | 72.69 MB/s (30) | 54.05 MB/s (22) |

Table 6.11: MPI communication bandwidth on the IBM SP2 (Variable Data Size).

Variable Data Size case we observe good parallel efficiency and reasonably consistent Real Efficiency. For the 8 processor case we observe Parallel Efficiency greater than 100 %. Observation of multiple runs of the same experiment shows that there is not more than a 5-10 % difference in the timings over different runs. Similarly in Table 6.12 for Constant Data Size, we observe good Parallel Efficiency and consistent Real Efficiency (upto 16 processors). In this case we observe greater than 100 % efficiency for the 4 processor case.

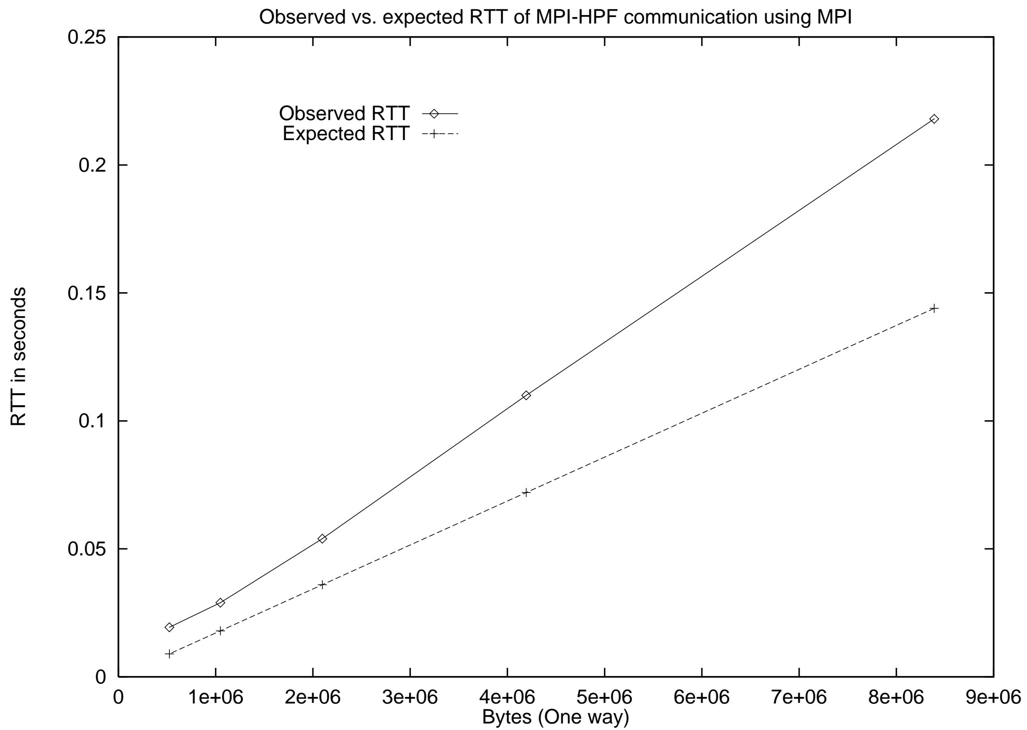


Figure 6.30: Observed vs. expected RTT for MPI-HPF communication using MPI (communicating processes on different processors).

Figure 6.30 compares expected and observed timings. The expected values are calculated by considering the peak bandwidth available between any two nodes of

the IBM SP2. We see that the observed RTT deviates more from the expected RTT as we increase the data size, possibly because of allocation of more system buffers.

Figure 6.31 shows that our timings are fairly stable. We observe a sharp increase in standard deviation at the 16 processor case.

The 64 processor IBM SP2 is made up of 16 processor frames, each of which houses up to 16 nodes and one switch board. These frames are connected via interframe cables to get a four-stage switch network. In the 16 processor case, all the inter-processor communication is within a single switch board, it could explain possible increase in timing variations.

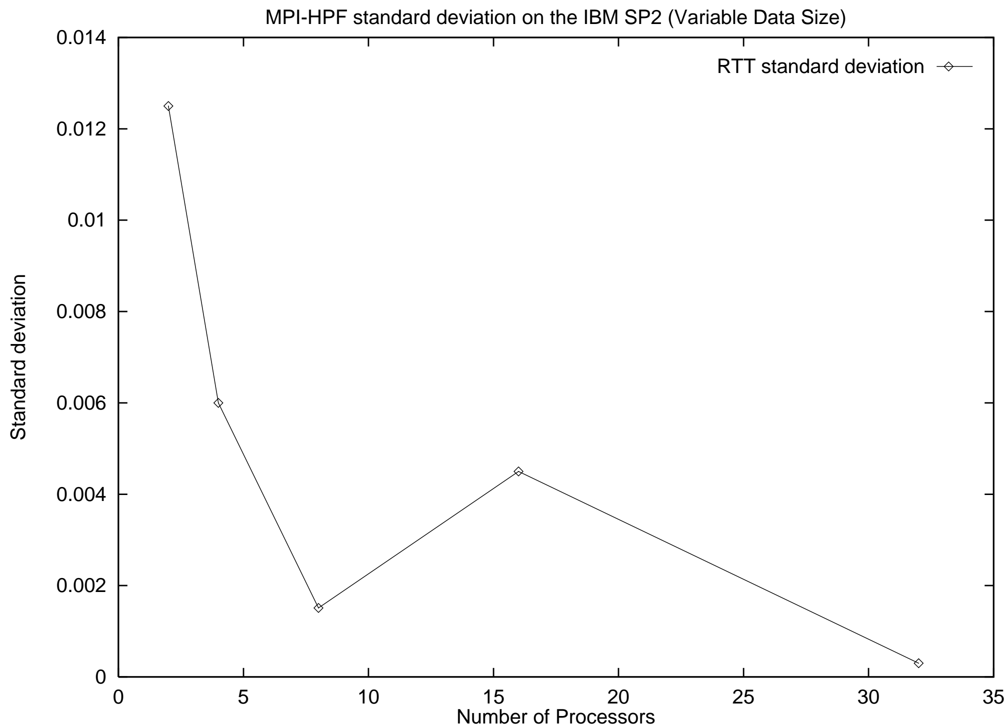


Figure 6.31: Standard deviation for MPI-HPF RTT using MPI with communicating processes on different processors on the IBM SP2.

MPI-HPF RTT with MPI communication on the IBM SP2
(communicating processes on different processors)
(8 MB per MPI-HPF Process)

| No. of Procs | Avg Time in secs (longest) | Standard Deviation | Parallel Efficiency | Real Efficiency |
|---|---|---|---|---|
| 2 | 0.218 | 0.0086 | 100.00 % | 66.05 % |
| 4 | 0.217 | 0.0120 | 100.36 % | 66.27 % |
| 8 | 0.218 | 0.0112 | 99.08 % | 65.99 % |
| 16 | 0.218 | 0.0100 | 99.72 % | 65.87 % |
| 32 | 0.291 | 0.0050 | 74.75 % | 49.36 % |

Table 6.12: MPI-HPF communication using MPI on the IBM SP2. (Constant Data Size )

Figures 6.32 and 6.33 show communication costs for constant data sizes from zero bytes to 16 KB and from 16 KB to 48 MB respectively. Peak Real Efficiency of 77.91 % was observed with two processors for a data size of 48 MB. The best RTT is calculated by considering the maximum hardware bandwidth. From Figures 6.32 and 6.33 we can conclude that MPI-HPF communication with MPI becomes significantly closer to the best performance one can expect as the data sizes are increased. This behavior is also observed in Table 6.10 where Real Efficiency increases by almost 20 % when we go from 512 KB to 8 MB. Significant disparities in performance between different processor cases are especially noticable for small data size cases (less than 2 KB) and for large data size cases (50 MB). Figure 6.33 also verifies the timings in Table 6.10, where the MPI-HPF round-trip times in Table 6.10 are approximately equal to that of of those in Figure 6.33 for the two processor case (8 MB data size) and the sixteen processor case (1 MB data size).

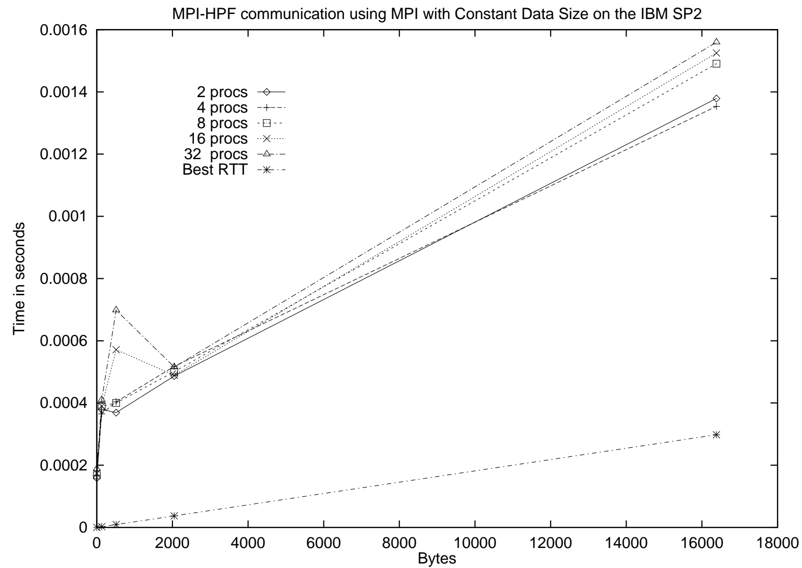


Figure 6.32: MPI-HPF RTT/2 using MPI with communicating processes on different processors for Constant Data Sizes per processor on the IBM SP2.

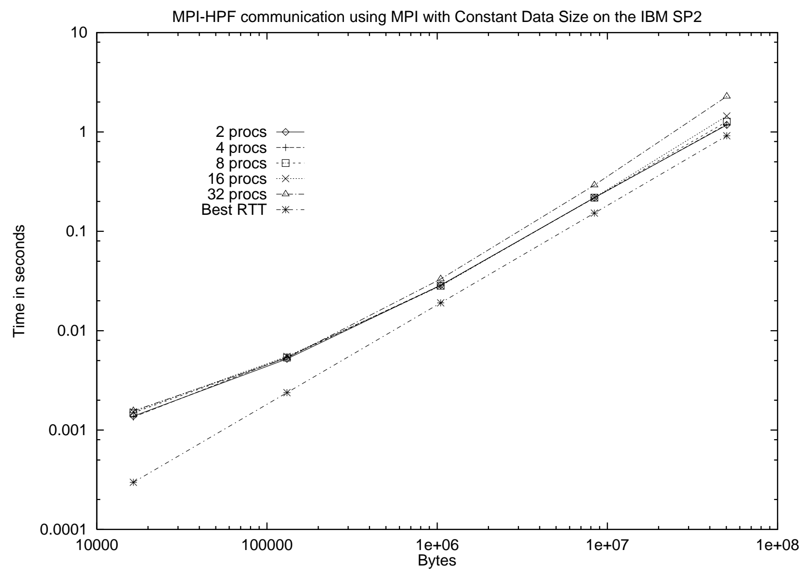


Figure 6.33: MPI-HPF RTT/2 using MPI with communicating processes on different processors for Constant Data Sizes per processor on the UH IBM SP2.

Figures 6.34 compares MPI-HPF communication with MPI communication between one pair of processes and MPI communication with all processes communicating for variable data sizes. We do not observe significant sensitivity to number of processors nor do we see network contention affecting performance. For the Constant Data Size case shown in Figure 6.35, there is maximum of a 25 % difference between MPI communication between process pairs and with all processes communicating. In both Figure 6.34 and 6.35 however, MPI-HPF communication takes more than both the MPI Communication cases, though not by very significant amounts (less than 18 %).

A closer look at the HPF intermediate code reveals that before and after each extrinsic procedure the *pghpf* compiler does some additional copying and hence the explanation for increase in time over the MPI communication.

**Communicating Processes on the same Processor.**

In this case the programs were run by giving a *hostfile* and a *commandfile* as input to *poe* to schedule the MPI ranks desired on particular hosts. Through this mechanism we could assure that the communicating MPI and HPF processes were scheduled to run on the same processor. It was not possible to run this with IBM's Loadleveler, as it overrides our desired hostfile settings. These programs were run in *ip* mode.

In this case, Real Efficiency is calculated by comparing with memory copy timings:

$$Real_Efficiency = \frac{Memory_copy_time}{(Socket_rtt/2)} * 100. \qquad (6.5)$$

From Tables 6.13 and 6.15 we conclude that MPI-HPF communication achieves

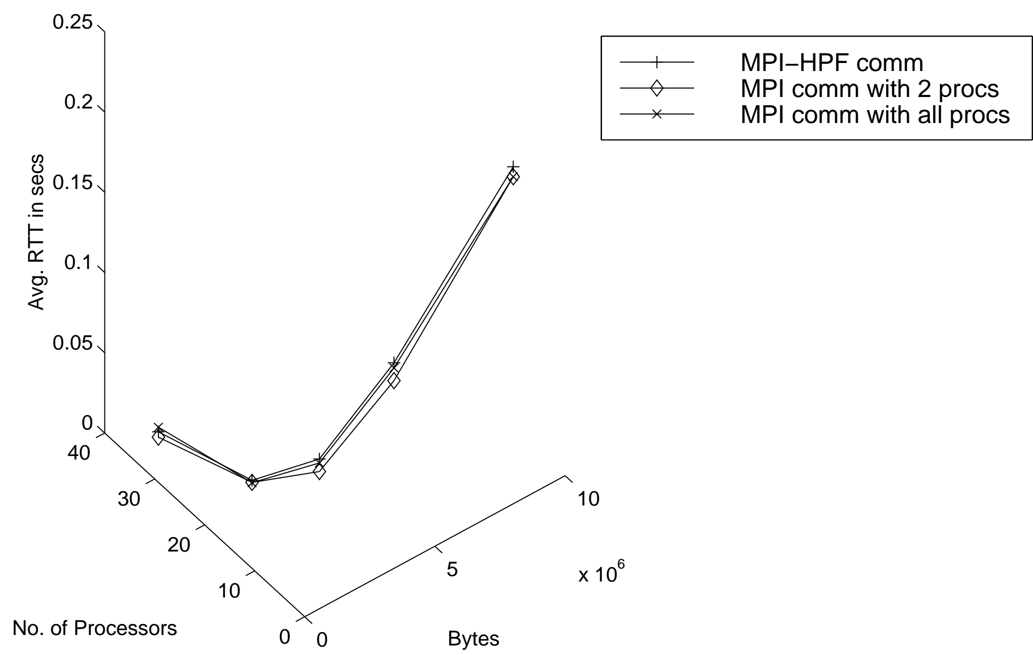


Figure 6.34: MPI communication vs. MPI-HPF communication with Variable Data Size on the UH IBM SP2.

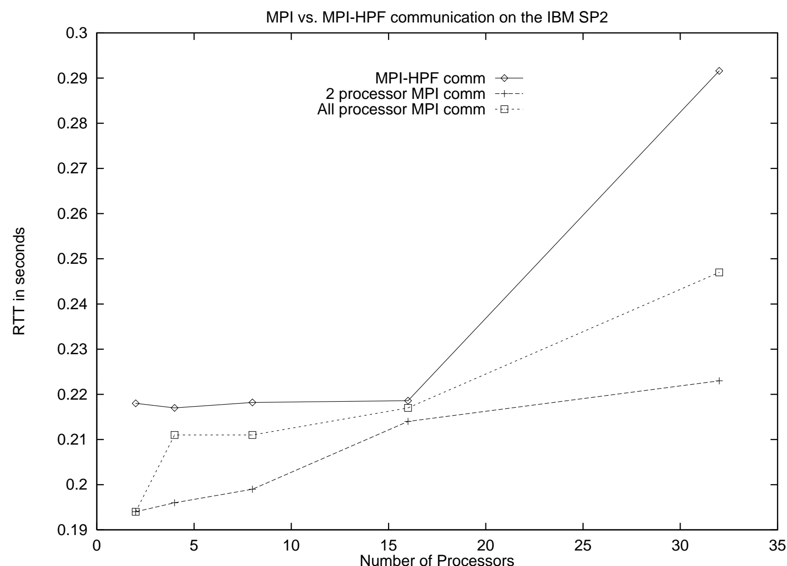


Figure 6.35: MPI communication vs. MPI-HPF communication with Constant Data Size on the IBM SP2.

reasonably high Parallel Efficiency and consistent Real Efficiency. Figure 6.37 shows that the timings recorded are reasonably stable. It is interesting to note though that the bandwidth achieved in this case (Table 6.14) is much less than that of the socket communication (Table 6.2). This is because the MPI implementation on the IBM SP2, uses TCP/IP for communication in *ip* mode, even though communicating processes are on the same processor.

Figures 6.38 and 6.39 show communication for constant data sizes from zero bytes to 16 KB and from 16 KB to 24 MB respectively. A peak Real Efficiency of 9.61 % was observed with one processor for a data size of 24 MB. Figure 6.39 also verifies the timings in Table 6.13, where the MPI-HPF round-trip times in Table 6.13 are approximately equal to that of of those in Figure 6.39 for the one processor case (8 MB data size) and the eight processor case (1 MB data size).

MPI-HPF RTT with MPI comm on IBM SP2
(communicating processes on the same processor)
(Array size: 8 MB /number_of_processors )

| No. of Procs | Bytes (one way) | Avg Time in secs (longest) | Standard Deviation | Parallel Efficiency | Real Efficiency |
|---|---|---|---|---|---|
| 1 | 8 MB | 1.098 | 0.0555 | 100.00 % | 9.08 % |
| 2 | 4 MB | 0.535 | 0.0277 | 102.61 % | 9.24 % |
| 4 | 2 MB | 0.275 | 0.0210 | 99.81 % | 9.00 % |
| 8 | 1 MB | 0.143 | 0.0130 | 95.97 % | 8.66 % |
| 16 | 512 KB | 0.077 | 0.0184 | 89.12 % | 8.01 % |

Table 6.13: MPI-HPF communication with MPI on the IBM SP2 (Variable Data Size).

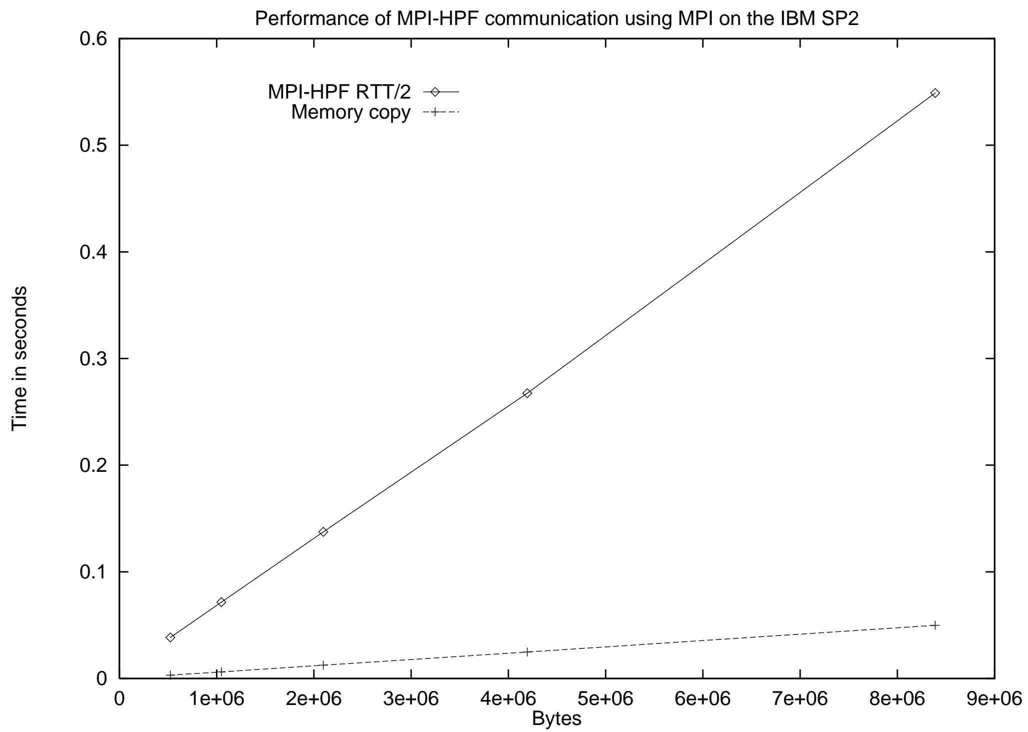


Figure 6.36: MPI-HPF communication with MPI (communicating processes on the same processor).

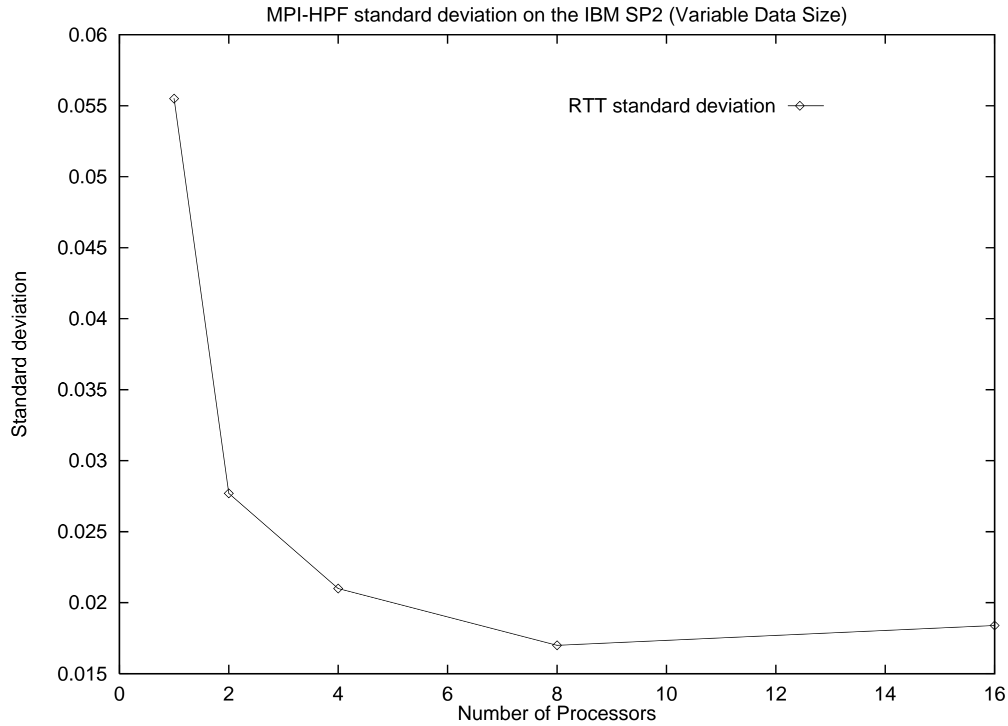


Figure 6.37: MPI-HPF RTT (using MPI) standard deviation on the IBM SP2 (communicating processes on the same processor).

Bandwidth for MPI communication on the IBM SP2 (communicating processes on the s ame processor)
( Array size: 8 MB /number_of_processors )

| No. of Procs | Bytes | Peak Bandwidth (MPI Proc Num) | Avg. Bandwidth (MPI Proc Num) |
|---|---|---|---|
| 1 | 8 MB | 16.98 MB/s (1) | 7.63 MB/s (1) |
| 2 | 4 MB | 17.84 MB/s (3) | 7.83 MB/s (2) |
| 4 | 2 MB | 17.76 MB/s (5) | 7.62 MB/s (4) |
| 8 | 1 MB | 17.84 MB/s (9) | 7.65 MB/s (14) |
| 16 | 512 KB | 18.42 MB/s (29) | 6.80 MB/s (16) |

Table 6.14: MPI-HPF communication bandwidth using MPI on the IBM SP2 (Variable Data Size).

MPI-HPF RTT with MPI communication on the IBM SP2
(communicating processes on the same processor)
(8 MB per MPI-HPF Process)

| No. of Procs | Avg Time in secs (longest) | Standard Deviation | Parallel Efficiency | Real Efficiency |
|---|---|---|---|---|
| 1 | 1.098 | 0.0555 | 100.00 % | 9.08 % |
| 2 | 1.083 | 0.0425 | 100.38 % | 9.21 % |
| 4 | 1.088 | 0.0460 | 100.91 % | 9.16 % |
| 8 | 1.100 | 0.0500 | 99.81 % | 9.06 % |
| 16 | 1.105 | 0.0510 | 99.36 % | 9.02 % |

Table 6.15: MPI-HPF communication using MPI on the UH IBM SP2 (Constant Data Size) .

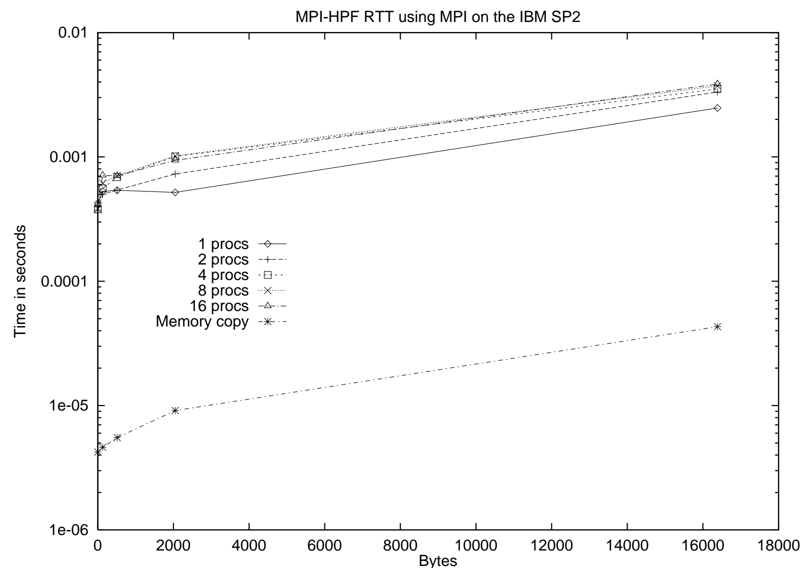


Figure 6.38: MPI-HPF RTT/2 (using MPI with communicating processes on the same processor) for Constant Data Sizes per processor on the IBM SP2.

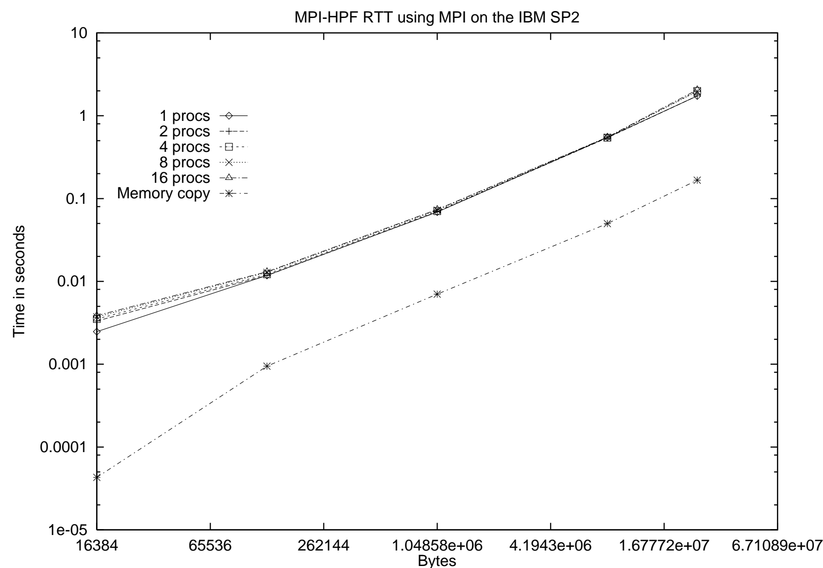


Figure 6.39: MPI-HPF RTT/2 (using MPI with communicating processes on the same processor) for Constant Data Sizes per processor on the UH IBM SP2.

#### Comparison of the two modes of MPI-HPF communication using MPI on the IBM SP2

The MPI-HPF communication using MPI with communicating processes on the same processor is similar to socket communication and thus we can run double the number of processes on the given processor set. MPI-HPF communication using MPI with processes on different processors however use only one process per processor. Therefore if a particular MPI/HPF program is waiting for the ouput of the other HPF/MPI program then the first set of processors remain idle.

Comparing Table 6.10 and 6.13 we see that Real Efficiency is much better (approximately a factor of six) in Table 6.10 than in Table 6.13. This could be because while using MPI-HPF communication with communicating processes on different processors, MPI uses the user-space protocol which is an optimized, lightweight protocol

that does not require a kernel call. Therefore, when we compare our observed timings with expected timings (peak hardware bandwidth), we get good Real Efficiency numbers. When using MPI-HPF communication with communicating processes on the same processor, however, MPI uses TCP/IP for communication between the two processes. Hence, when comparing observed and expected performance (memory copy operation), we lose out on Real Efficiency. However, discussions with IBM have revealed that the latest MPI implementations will provide options for data transfer between processes on the same processor using shared memory primitives.

### 6.4.2 The HP Exemplar

HP's implementation of MPI uses shared memory for communication between processes on the same hypernode and TCP/IP for communication between processes on different hypernodes [30]. In addition, hardware data movers and high performance data transfers are used in MPI communication. According to [30], over 800 MB/s MPI bandwidth should be expected if the two communicating processes are on the same hypernode and close to 170 MB/s MPI bandwidth if the two communicating processes are on different hypernodes. As the MPI libraries take care of underlying communication protocols between processors and provide a higher-level interface to the user, the fact that the MPI bandwidth is lesser than the peak hardware bandwidth comes as no surprise.

At first we discuss the MPI-HPF communication (with all MPI and HPF processes communicating) for constant and variable data sizes (Tables 6.16, 6.17, 6.18 and Figures 6.40, 6.41) and then we compare these results with MPI communication between one pair of processes and MPI communication between all processes communicating pairwise (Figures 6.42 and 6.43).

Since peak hardware memory bandwidth is 1.5 GB/s within hypernodes, our formula for Real Efficiency for communication becomes:

$$Real_Efficiency = \frac{Number_of_bytes}{1.5 * 10^9 * (MPI_rtt/2)} * 100. \tag{6.6}$$

For communication between hypernodes, since the peak hardware bandwidth is 480 MB/s, our formula for Real Efficiency becomes:

$$Real_Efficiency = \frac{Number_of_bytes}{480 * 10^6 * (MPI_rtt/2)} * 100. \tag{6.7}$$

From Table 6.16 we see that Parallel Efficiency reduces from eight processors onward. Real Efficiency also deteriorates after four and eight processors. This is depicted graphically in Figure 6.40, which shows the difference between observed and expected RTT. The expected RTT line in Figure 6.40 is calculated by substituting all the known values in the two equations above. For the 32 processor case, the two communicating programs reside on different hypernodes and therefore there is a significant reduction in Parallel Efficiency.

MPI-HPF RTT with MPI communication on the HP Exemplar
( Array size: 8 MB /number_of_processors )

| No. of Procs | Bytes (one way) | Avg Time in secs (longest) | Standard Deviation | Parallel Efficiency | Real Efficiency |
|---|---|---|---|---|---|
| 2 | 8 MB | 0.0227 | 0.0038 | 100.00 % | 46.57 % |
| 4 | 4 MB | 0.0164 | 0.0044 | 69.20 % | 32.27 % |
| 8 | 2 MB | 0.0178 | 0.0018 | 31.73 % | 14.81 % |
| 16 | 1 MB | 0.0266 | 0.0387 | 10.66 % | 4.96 % |
| 32 | 512 KB | 0.0487 | 0.0389 | 2.95 % | 4.44 % |

Table 6.16: Communication with MPI on the HP Exemplar (Variable Data Size).

Table 6.17 shows that we reach close to the published bandwidth upto four processors, but then observe rapid deterioration after the number of processors are increased. Figure 6.41 shows high variations in timings for higher number of processors (16, 32) for the Variable Data Size case. Table 6.18 shows MPI-HPF communication with constant data size. Reasons for the above observed behavior can be studied by comparing

1. MPI Communication between one pair of processes

2. MPI Communication with all processes communicating

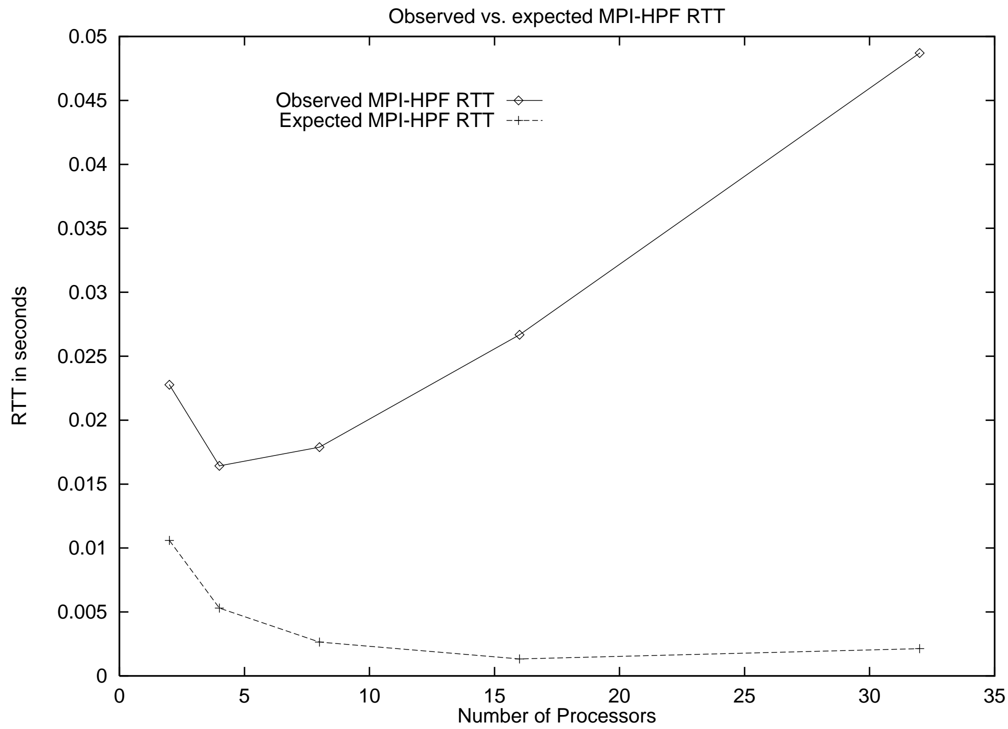


Figure 6.40: Observed vs. expected MPI-HPF communication with MPI on the HP Exemplar.

Bandwidth for MPI communication on the HP Exemplar
( Array size: 8 MB /number_of_processors)

| No. of Procs | Bytes | Peak Bandwidth (Proc Num) | Avg. Bandwidth (Proc Num) |
|---|---|---|---|
| 2 | 8 MB | 761.42 MB/s (1) | 737.09 MB/s (1) |
| 4 | 4 MB | 634.49 MB/s (2) | 510.77 MB/s (3) |
| 8 | 2 MB | 471.21 MB/s (7) | 234.52 MB/s (4) |
| 16 | 1 MB | 333.88 MB/s (11) | 78.60 MB/s (12) |
| 32 | 512 KB | 26.19 MB/s (26) | 21.52 MB/s (27) |

Table 6.17: MPI communication bandwidth on the HP Exemplar (Variable Data Size).

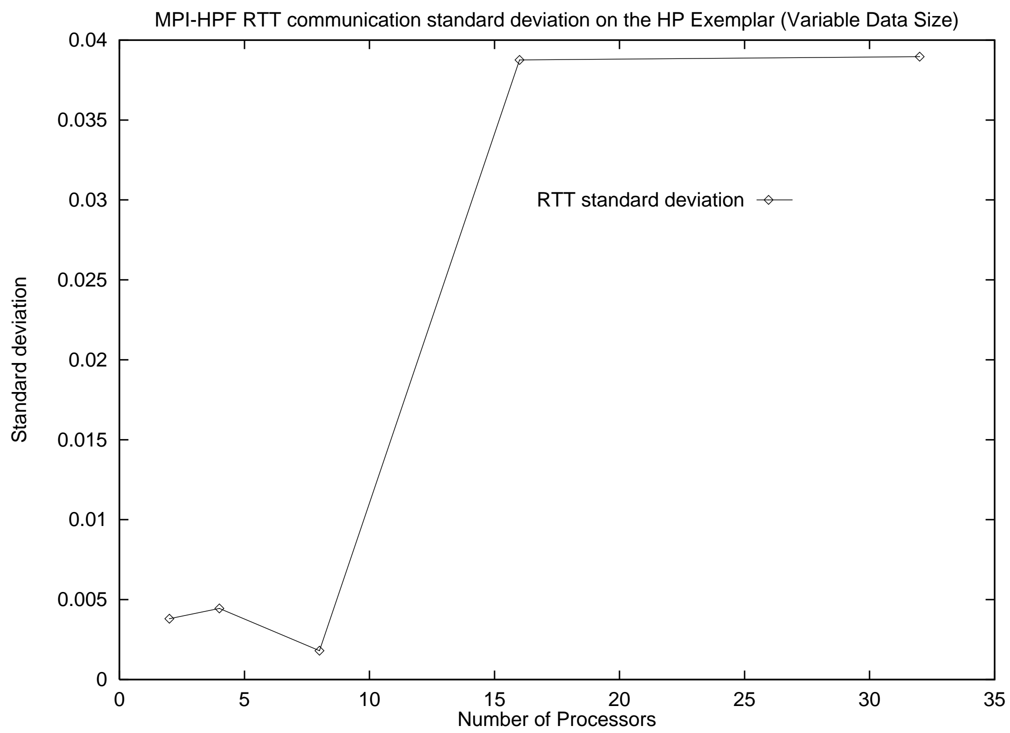


Figure 6.41: MPI-HPF RTT (using MPI) standard deviation on the HP Exemplar. (Variable Data Size)

3. MPI-HPF Communication

MPI-HPF RTT with MPI Communication on the HP Exemplar
(8 MB per MPI-HPF Process)

| No. of Procs | Avg Time in secs (longest) | Standard Deviation | Parallel Efficiency | Real Efficiency |
|---|---|---|---|---|
| 2 | 0.022 | 0.0071 | 100.00 % | 46.03 % |
| 4 | 0.030 | 0.0091 | 78.02 % | 35.04 % |
| 8 | 0.059 | 0.0202 | 32.27 % | 17.94 % |
| 16 | 0.112 | 0.0837 | 8.48 % | 9.42 % |
| 32 | 1.850 | 0.5780 | 1.22 % | 1.80 % |

Table 6.18: MPI-HPF communication using MPI on the HP Exemplar (Constant Data Size) .

The above three methods were compared using both Variable Data Sizes (Figure 6.42) and Constant Data Sizes (Figure 6.43) from 2-32 processors.

By looking at the first two cases, it is evident that when all processes are communicating at the same time, effective data transfer time increases. This could be the explanation for poor scaling of the MPI-HPF communication. For the variable data size case (Figure 6.42), we see that the MPI-HPF code takes from 13 % to 65 % longer than the MPI code with all processes communicating across the 2, 4, 8, and 16 processor cases. For the 32 processor case the MPI-HPF code takes about 30 % longer than the corresponding MPI timings.

For the constant data size case (Figure 6.43), there is a 2 % to 25 % difference in timing between the MPI-HPF code and the MPI code with all processes communicating across 2, 4, 8, 16 and 32 processor cases.

Again this difference can be explained by the addition of *extra* copying code by the HPF compiler before and after calling extrinsic procedures.

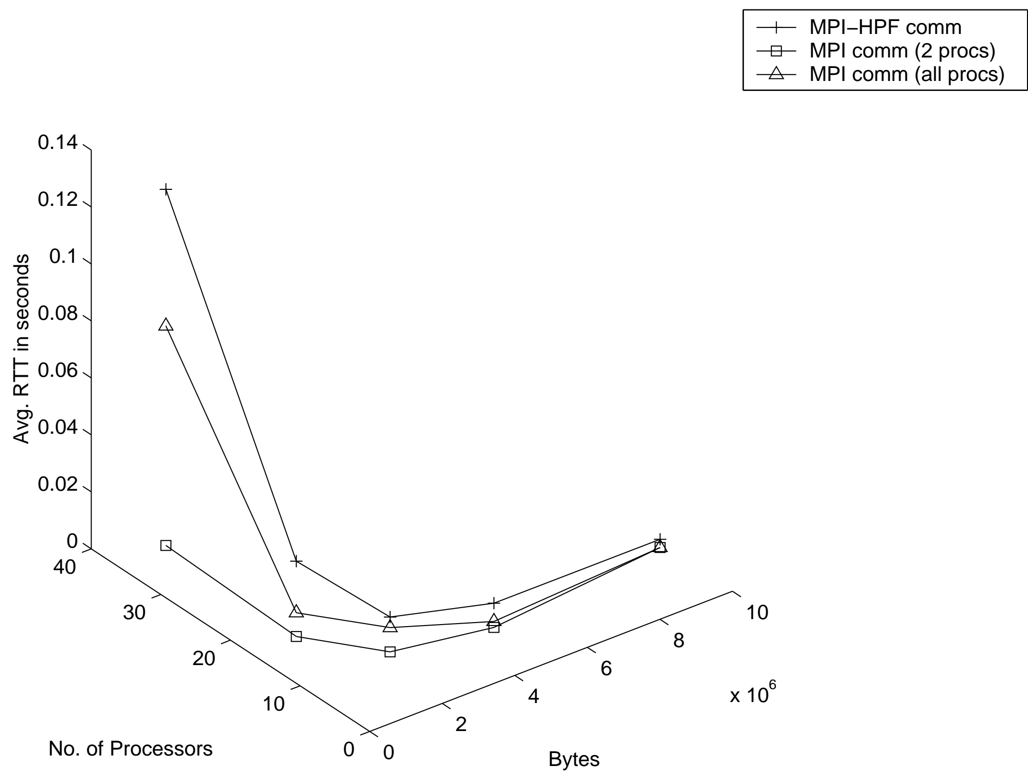


Figure 6.42: MPI communication vs. MPI-HPF communication with Variable Data Size per processor on the HP Exemplar.

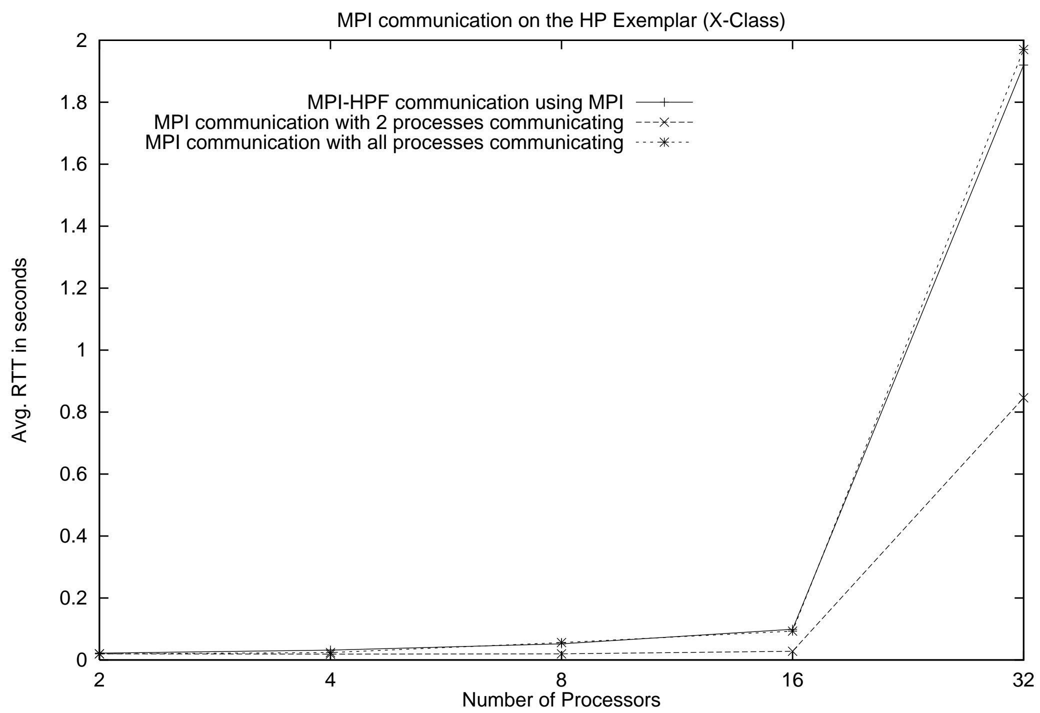


Figure 6.43: MPI communication vs. MPI-HPF communication with Constant Data Size 8 MB per processor on the HP Exemplar.

Figures 6.44 and 6.45 show communication for constant data sizes from zero bytes to 16 KB and from 16 KB to 64 MB respectively. Peak Real Efficiency of 56.37 % was observed with two processors for a data size of 64 MB. Figure 6.45 also verifies the timings in Table 6.16, where the MPI-HPF round-trip times in Table 6.16 are approximately equal to that of of those in Figure 6.45 for the two processor case (8 MB data size) and the sixteen processor case (1 MB data size).

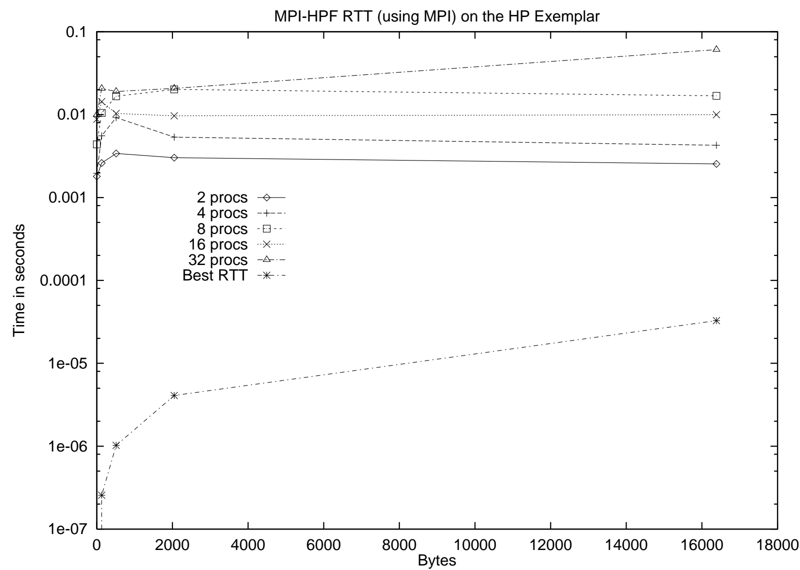


Figure 6.44: MPI-HPF RTT (using MPI) for Constant Data Sizes per processor on the HP Exemplar.

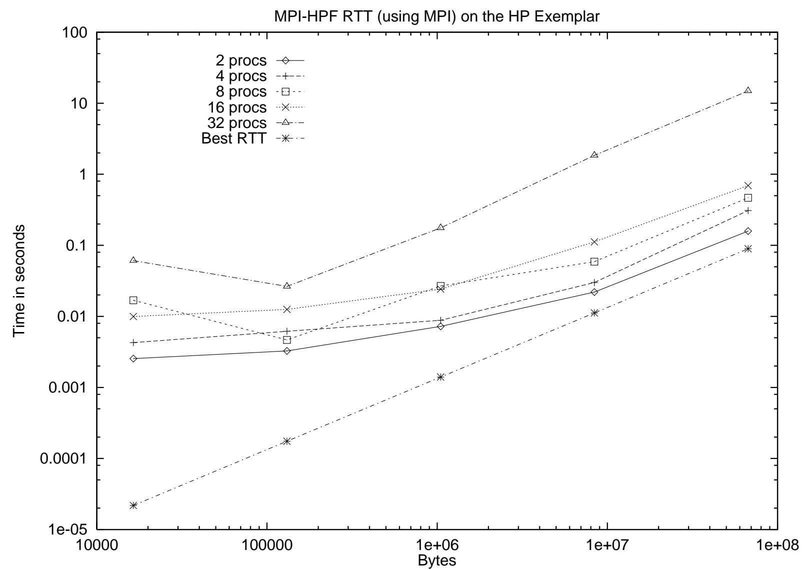


Figure 6.45: MPI-HPF RTT (using MPI) for Constant Data Sizes per processor on the HP Exemplar.

### 6.4.3 The SGI Origin 2000

On the SGI Origin 2000, the MPI-HPF communication with MPI proved to be quite disappointing as the timings for Constant Data Size linearly increased with an increase in the number of processors. To get a better understanding of what is happening, the performance of the MPI code with all processors communicating was investigated in both SPMD and MPMD mode. From Figure 6.46, it seems as if running in the MPMD mode is causing the performance problem. When running in MPMD mode the MPI implementation assumes that communication is going out of the machine and starts reading/writing to the HIPPI bypass, which connects the subsystems of the NCSA SGI array together. Therefore, for local MPMD communication, all the processes try to write to the serial HIPPI port simultaneously, thus increasing the time with the number of processors.

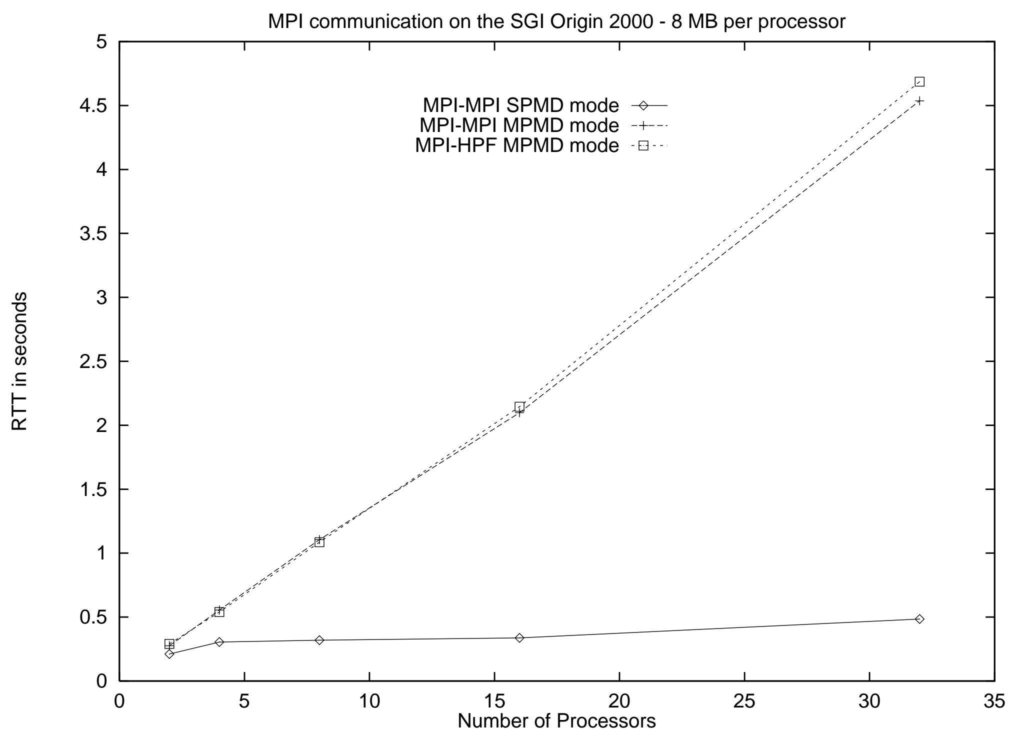


Figure 6.46: Various types of MPI communication on the SGI Origin 2000.

### 6.4.4 Globus

The MPMD model of communication is used for the Globus toolkit, since Globus views even SPMD programs as MPMD. It needs different code images or executables per site (depending upon the site's operating system and architecture).

The MPI and HPF programs were run between the UH SP2 and the San Diego SP2. The HPF code was first translated to intermediate F77 code with message passing calls and was then linked with globus MPI libraries. Only the Variable Data size case was tested on Globus.

With regard to bandwidth, a traceroute between UH and SDSC shows there are eight hops between the two sites and that the connection goes through the vBNS.

traceroute to sp129.sdsc.edu (132.249.40.201) from 129.7.136.17 (129.7.136.17)

1 n021s (129.7.136.21)

2 brutus-atm5-00-1.gw.uh.edu

3 LINK2UH.GIGAPOP.GEN.TX.US

4 VBNS.GIGAPOP.GEN.TX.US

5 cs-atm0-0-18.sdsc.vbns.net

6 medusa.sdsc.edu

7 tigerfish.sdsc.edu

8 sp129.sdsc.edu

The weakest link has a capacity of 45 Mbit/sec and hence we achieve approximately 10 % of the hardware bandwidth as seen in Table 6.19.

Round Trip Times with MPI communication on Globus

| No. of Procs | Bytes (one way) | Min Time (in seconds) | Parallel Efficiency | Bandwidth (Peak) |
|---|---|---|---|---|
| 2 | 8 MB | 22.96 | 100.00 % | 0.689 MB/s |
| 4 | 4 MB | 17.16 | 64.65 % | 0.421 MB/s |
| 8 | 2 MB | 14.39 | 39.92 % | 0.208 MB/s |

Table 6.19: MPI-HPF communication (using MPI) on Globus.

### 6.4.5 Summary of MPI Communication

As in the case of the MPI-HPF socket communication, communication using MPI will be successful on architectures where any pair of communicating MPI-HPF processes will cause zero or minimal interference to other communicating MPI-HPF processes. For the IBM SP2, since available bandwidth between any pair of communicating nodes remains constant when the communicating processes are on different processors, we have observed that we get good parallel efficiency. For the case where communicating processes are on the same processor, interference from other processes will not be present and good parallel efficiency is observed here too. With regard to the HP Exemplar, we have seen that the performance of MPI communication is sensitive to the total number of processes communicating. Therefore we do not observe good parallel efficiency although in the best cases MPI is able to achieve high bandwidth (within 53 % of hardware peak). The SGI Origin 2000 has not proven to be suitable for MPMD communication and hence at present it is not advisable to use this communication technique for MPI-HPF communication.

## 6.5 Shared Memory Communication

Figures 6.47, 6.48, and 6.49 show a comparison of the communication between MPI-HPF and equivalent MPI programs, which simulate the behavior of the MPI-HPF communication, on the IBM SP2, the HP Exemplar and the SGI Origin 2000. We see that the MPI-HPF communication time increases linearly with the number of processors, unlike the MPI-MPI communication. We conjecture that the performance problem is because of the synchronization between the MPI and HPF programs. The MPI program needs to wait for the HPF program after it has sent data before it receives data from the HPF program. A closer look at the timings of various parts of the code reveal that it is this waiting period that increases with increase in the number of processors. As this behavior is only observed for MPI-HPF communication and is generic to all platforms, the problem could also be due to some properties of the *pghpf* compiler.

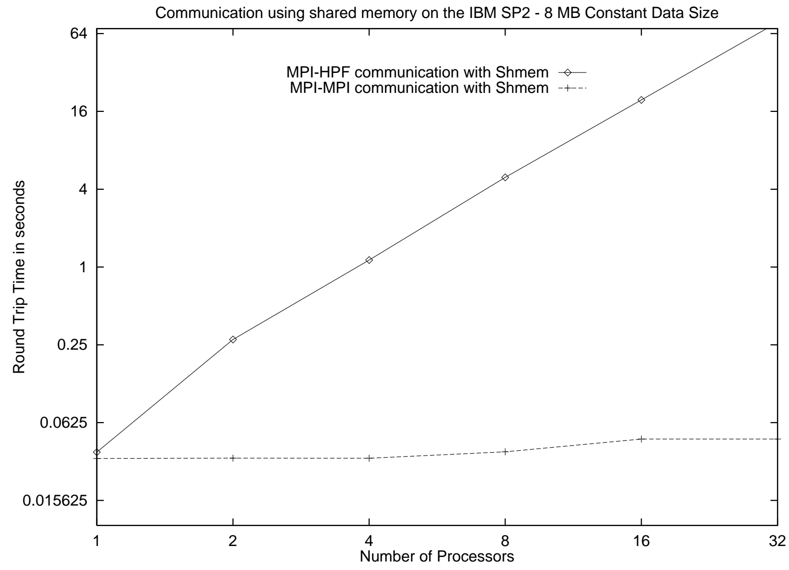


Figure 6.47: Shared memory communication on the IBM SP2.(Constant Data Size)

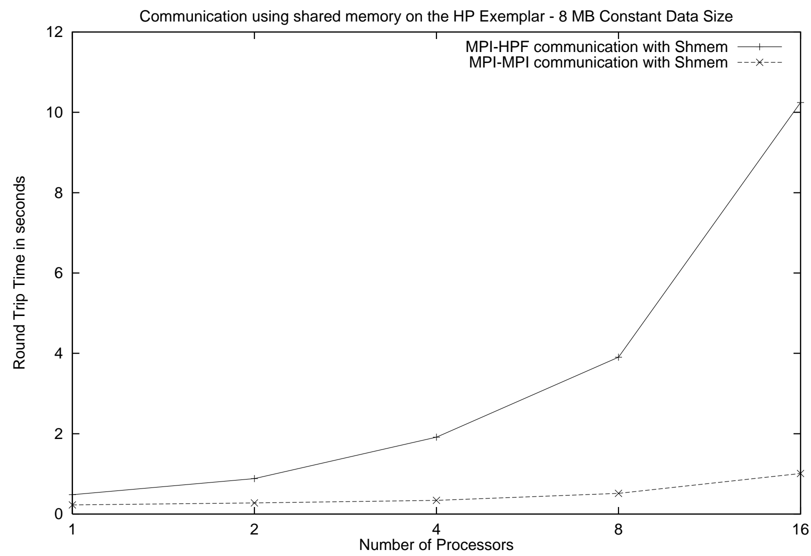


Figure 6.48: Shared memory communication on the HP Exemplar.(Constant Data Size)

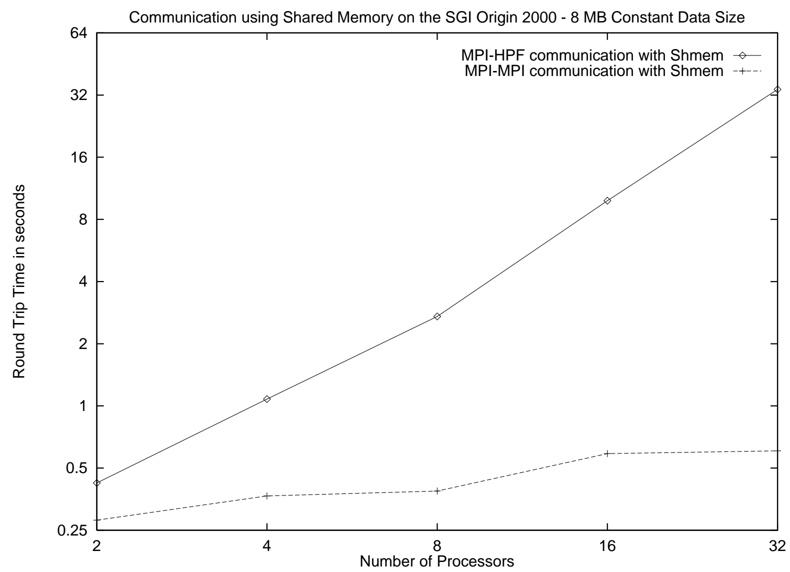


Figure 6.49: Shared memory communication on the SGI Origin 2000.(Constant Data Size)

## 6.6 Summary of model program mechanisms

- **Socket communication.** For the IBM SP2, single processor timings for data sizes from zero bytes to 2 KB show considerable variations and hence Parallel Efficiency is affected. Parallel Efficiency (upto 32 processors) starts improving from data sizes greater than 16 KB. Therefore it is advisable to use socket communication on the IBM SP2 upto 32 processors with data sizes greater than 16 KB.

  For the HP Exemplar single processor MPI-HPF data transfer timings are relatively stabilized after data sizes of 512 bytes. Data sizes near 16 KB cause a sharp increase in timing because the buffer boundary conditions cause a sharp increase in timing than expected. For data sizes greater than 16 KB, Parallel Efficiency (upto 16 processors) improves. However, beyond a data size of 16 MB Parallel Efficiency reduces for 16 processors. We do not advise to use more than 16 communicating pairs of MPI and HPF processes as communication may take place across hypernodes thus increasing time taken for data transfer and reducing Parallel Efficiency. Therefore it is advisable to use MPI-HPF socket communication on the HP Exemplar upto 16 processors and for data sizes between 16 KB to 16 MB.

  For the SGI Origin 2000, single processor MPI-HPF data transfer timings are relatively stabilized after data sizes greater than or equal to 512 bytes. Parallel Efficiency (upto 64 processors) also starts improving after this data size. Therefore it is advisable to use MPI-HPF socket communication on the SGI Origin 2000 upto 64 processors and for data sizes greater than 512 bytes.

- **Communication using MPI.** For the IBM SP2, when communicating processes are on different processors, MPI-HPF data transfer times get stabilized after data sizes of 2 KB. Thereafter, Parallel Efficiency upto 32 processors improves till data sizes of 16 MB. From 16 MB to 64 MB we see that Parallel Efficiency for 32 processors decreases. When the communicating processes are on the same processor, data transfer times for MPI-HPF communication gets stabilized after a data size of 512 bytes. Thereafter, Parallel Efficiency upto 16 processors keeps improving steadily.

  In the case of the HP Exemplar, MPI-HPF data transfer times stabilize after data size of 512 bytes. Parallel Efficiency however, reduces considerably after 4 processors. Therefore, we do not advise to use more than 4 communicating pairs of MPI and HPF processes.

  For the SGI Origin 2000, MPI-HPF communication using SGI's MPI implementation should not be used at this point.

- **Communication using Shared Memory.** MPI-HPF communication using shared memory is not yet fully understood at this point, so this mechanism of communication should not be used.

Tables 6.20, 6.21 and 6.22 summarize the preferred MPI-HPF communication techniques and data sizes for each platform.

Preferred Communication Mechanisms on the IBM SP2

| Data Set Size | Number of Processors | Preferred Mechanism |
|---|---|---|
| 2 KB to 16 KB | 1 to 16 | MPI Communication (two processes per processor) |
| $> 16$ KB | 1 to 32 | Socket communication |
| 2 KB to 16 MB | 1 to 32 | MPI Communication (one process per processor) |

Table 6.20: MPI-HPF communication on the IBM SP2.

Preferred Communication Mechanisms on the HP Exemplar

| Data Set Size | Number of Processors | Preferred Mechanism |
|---|---|---|
| 512 to 16 KB | 1 to 4 | MPI Communication |
| 16 KB to 16 MB | 1 to 16 | Socket communication |

Table 6.21: MPI-HPF communication on the HP Exemplar.

Preferred Communication Mechanisms on the SGI Origin 2000

| Data Set Size | Number of Processors | Preferred Mechanism |
|---|---|---|
| $> 512$ B | 1 to 64 | Socket communication |

Table 6.22: MPI-HPF communication on the SGI Origin 2000.

## 6.7 Performance of Application Codes

The FMD and Anderson codes were run on three supercomputing platforms, wth sockets used for the communication. For the MPI-HPF communication with MPI changes to existing HPF codes are required. In attempting to use this mechanism in our application code, we found that the *pghpf* compiler does not support the MPMD model for some intrinsic HPF functions (**GRADE_UP** ) and F90 reduction intrinsics. Therefore, our application codes are interfaced using sockets at present. As was described in Chapter 5, the FMD code sends coordinates and charges to the Anderson code, which sends back calculated potential and force vectors.

When comparing the results of our application programs with that of the model programs on the IBM SP2 and SGI Origin 2000 systems, we observe from Figure 6.50, 6.51, 6.54 and 6.55 that our application codes exhibit behavior similar to that of our model programs. Sending data may take less time than receiving data as the reciever may have to wait for the sender to start sending before it can receive. The sender, however can start writing data to system buffers, even though the receiver may not be ready to receive.

For HP Exemplar systems both sending and receiving application codes take longer time than the model programs as shown in Figures 6.52 and 6.53. This may be due to the fact that both the application programs (FMD and Anderson) are taking up more memory compared to the model programs and hence the scheduling of processes across processors by the OS changes.

For the IBM SP2, we observe that the timings between the Anderson-FMD and the FMD-Anderson data transfers are not significantly different, except in the 32 processor case. In addition, efficiency is reasonable up to 16 processors. On the

SGI Origin 2000, we see that efficiency in both codes drops at the 64 processor case, because communication may take place across more number of clusters than the other cases. The pairs of communicating processes are on different clusters On the Exemplar, Parallel Efficiency drops at the 16 processor case and there are large variations in timings for the FMD-Anderson case. The efficiency is much better however in the Anderson-FMD case.

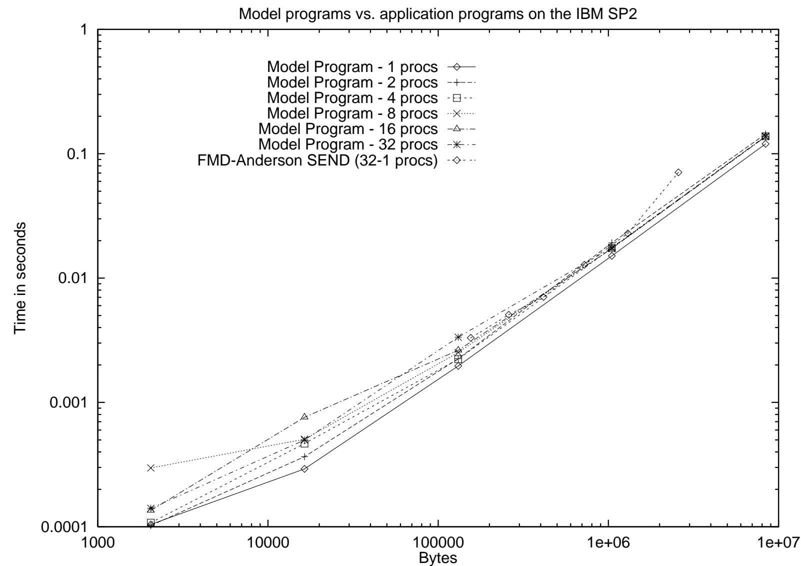


Figure 6.50: Application programs (send) vs. model programs (RTT/2) on the SDSC IBM SP2.

Tables 6.23, 6.24, 6.25, 6.26, 6.27 and 6.28 show the performance of the FMD-Anderson codes using our MPI-HPF socket communication and associated parallel efficiency.

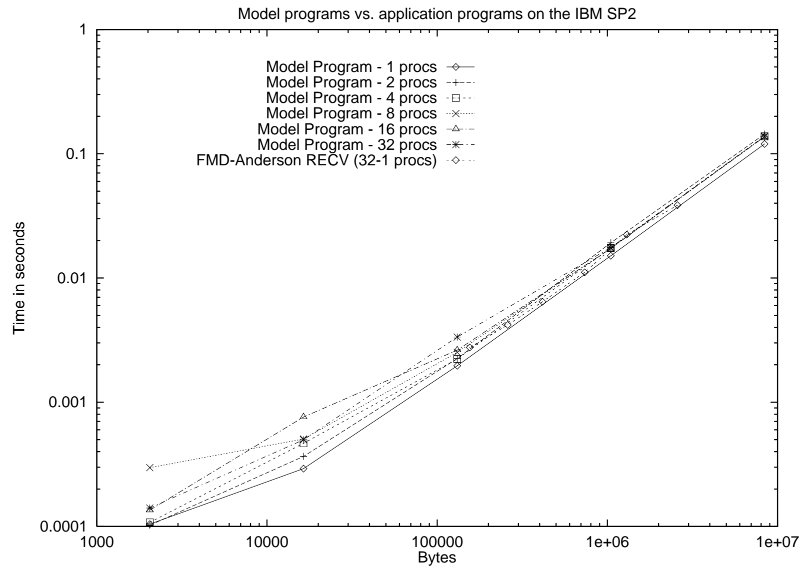


Figure 6.51: Application programs (recv) vs. model programs (RTT/2) on the SDSC IBM SP2.

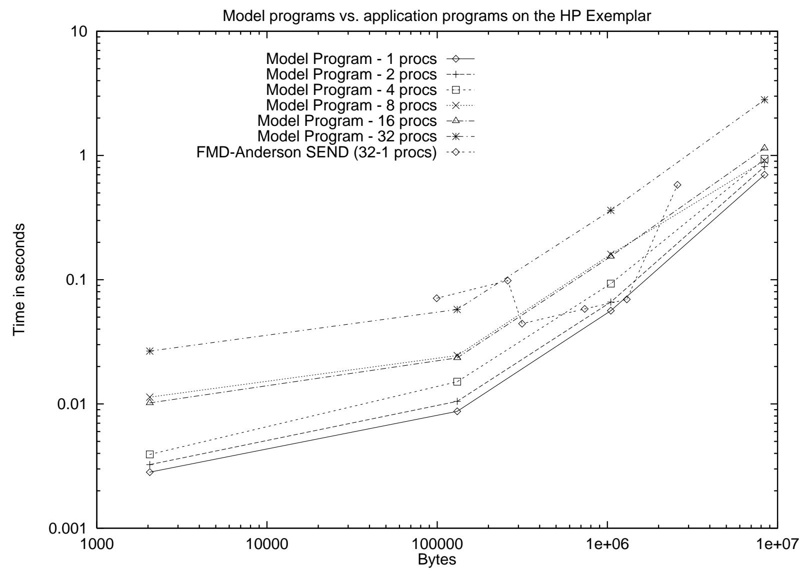


Figure 6.52: Application programs (send) vs. model programs (RTT/2) on the HP Exemplar.

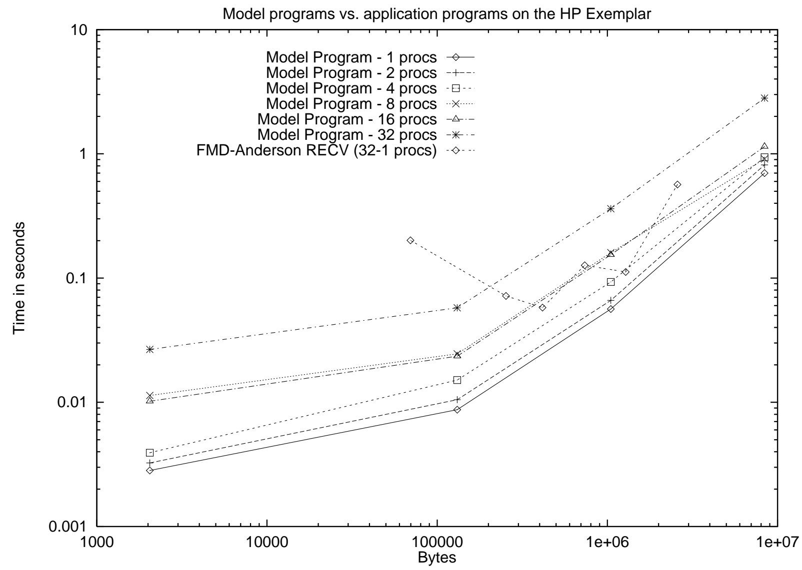


Figure 6.53: Application programs (recv) vs. model programs (RTT/2) on the HP Exemplar.

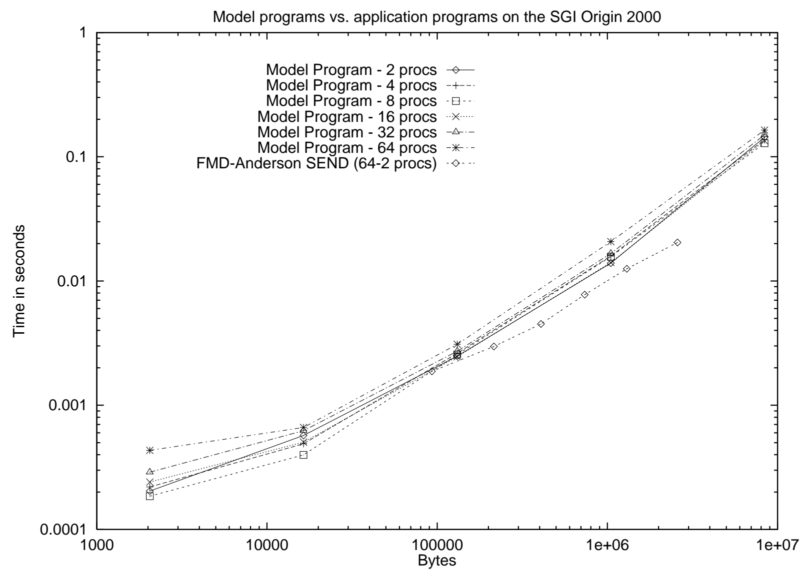


Figure 6.54: Application programs (send) vs. model programs (RTT/2) on the SGI Origin 2000.

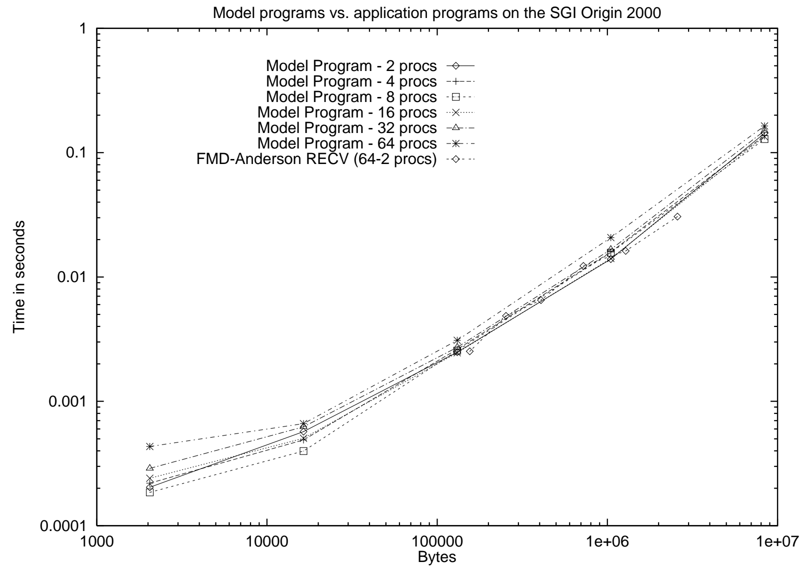


Figure 6.55: Application programs (recv) vs. model programs (RTT/2) on the SGI Origin 2000.

FMD-Anderson data transfer using socket communication with 80712 atoms on the IBM SP2

| No. of Procs | FMD side timing (in seconds) | Standard Deviation | Parallel Efficiency |
|---|---|---|---|
| 1 | 0.087 | 0.0615 | 100.00 % |
| 2 | 0.045 | 0.0102 | 96.67 % |
| 4 | 0.022 | 0.0036 | 98.86 % |
| 8 | 0.013 | 0.0020 | 81.15 % |
| 16 | 0.008 | 0.0012 | 61.78 % |
| 32 | 0.006 | 0.0189 | 43.15 % |

Table 6.23: FMD-Anderson with socket communication on the SDSC IBM SP2.

Anderson-FMD data transfer using socket communication with 80712 atoms on the IBM SP2

| No. of Procs | Anderson side timing (in seconds) | Standard Deviation | Parallel Efficiency |
|---|---|---|---|
| 1 | 0.0750 | 0.0050 | 100.00 % |
| 2 | 0.0440 | 0.0033 | 85.22 % |
| 4 | 0.0210 | 0.0017 | 89.28 % |
| 8 | 0.0120 | 0.0012 | 78.12 % |
| 16 | 0.0083 | 0.0008 | 56.47 % |
| 32 | 0.0089 | 0.0290 | 26.33 % |

Table 6.24: Anderson-FMD with socket communication on the SDSC IBM SP2.

FMD-Anderson data transfer using socket communication with 80712 atoms on the SGI Origin 2000

| No. of Procs | FMD side timing (in seconds) | Standard Deviation | Parallel Efficiency |
|---|---|---|---|
| 2 | 0.0494 | 0.0054 | 100.00 % |
| 4 | 0.0305 | 0.0034 | 80.98 % |
| 8 | 0.0188 | 0.0032 | 65.69 % |
| 16 | 0.0104 | 0.0015 | 59.37 % |
| 32 | 0.0066 | 0.0036 | 46.78 % |
| 64 | 0.0051 | 0.0039 | 30.26 % |

Table 6.25: FMD-Anderson with socket communication on the SGI Origin 2000.

Anderson-FMD data transfer using socket communication with 80712 atoms on the IBM SP2

| No. of Procs | Anderson side timing (in seconds) | Standard Deviation | Parallel Efficiency |
|---|---|---|---|
| 2 | 0.045 | 0.0065 | 100.00 % |
| 4 | 0.030 | 0.0037 | 75.00 % |
| 8 | 0.017 | 0.0031 | 66.17 % |
| 16 | 0.008 | 0.0021 | 63.92 % |
| 32 | 0.005 | 0.0031 | 47.66 % |
| 64 | 0.003 | 0.0010 | 40.17 % |

Table 6.26: Anderson-FMD with socket communication on the SGI Origin 2000.

FMD-Anderson data transfer using socket communication with 80712 atoms on the HP Exemplar

| No. of Procs | FMD side timing (in seconds) | Standard Deviation | Parallel Efficiency |
|---|---|---|---|
| 1 | 1.20 | 0.041 | 100.00 % |
| 2 | 0.66 | 0.056 | 90.90 % |
| 4 | 0.40 | 0.061 | 75.00 % |
| 8 | 0.33 | 0.072 | 45.45 % |
| 16 | 0.35 | 0.114 | 21.42 % |
| 32 | 0.36 | 0.161 | 10.41 % |

Table 6.27: FMD-Anderson with socket communication on the HP Exemplar.

Anderson-FMD data transfer using socket communication with 80712 atoms on the HP Exemplar

| No. of Procs | Anderson side timing (in seconds) | Standard Deviation | Parallel Efficiency |
|---|---|---|---|
| 1 | 1.88 | 1.39 | 100.00 % |
| 2 | 0.80 | 0.275 | 117.50 % |
| 4 | 0.44 | 0.065 | 106.81 % |
| 8 | 0.29 | 0.044 | 81.03 % |
| 16 | 0.18 | 0.036 | 64.91 % |
| 32 | 0.15 | 0.016 | 37.42 % |

Table 6.28: Anderson-FMD with socket communication on the HP Exemplar.

# Chapter 7

# Summary/Conclusions and Future Work

## 7.1 Summary

We have presented mechanisms for communication between programs written using MPI and HPF. To the best of our knowledge, this is one of the first efforts in doing inter-process communication between parallel programs written using different paradigms. We have demonstrated that this concept can be implemented reasonably efficiently and without much programming effort or dependence on execution environments, as we use features commonly provided by most compilers and parallel and distributed operating environments. We have documented communication efficiency and compared it to expected machine performance for different processor numbers on diverse parallel (IBM SP2, HP Exemplar, SGI Origin 2000) and distributed computing (Globus) platforms. In addition we have pointed out possible hardware and software performance bottlenecks in different scenarios. Comparison between results

of the application programs and our model programs have been made and their results discussed.

## 7.2 Conclusions and Recommendations

- We conclude that the socket communication mechanism for MPI-HPF communication is the most portable at present. We have shown how this mechanism can be used on both distributed memory and SMP architectures in an efficient manner. However, we do look forward to using smart socket libraries, which do single memory copies for communication between processes on the same machine.

- In the process of running experiments on various parallel machines, we observe that distributed memory machines like the IBM SP2 are more predictable in terms of performance. This is because unlike the other two SMP's, where total machine throughput is considered while scheduling processes; the IBM SP2 generally provide each process with an almost exclusive CPU to run on and therefore performance is more predictable.

- Though our approach to the MPI-HPF communication problem is centered towards running two different programs, we look forward to the development of portable libraries and software to enable a single program model of calling HPF code from MPI.

- As we have attempted to provide a mixed parallel programming language model, we hope to see more research in the analysis of algorithms that exploit the strengths of both task parallel (MPI) and data parallel (HPF) programming

paradigms.

## 7.3 Ongoing and Future Work

The shared memory communication technique needs to be studied more in depth, so as to understand exactly where performance drops and the dependence of this problem on the *pghpf* compiler. It would also be interesting to test our MPI-HPF communication using with different MPI implementations on the SGI Origin 2000.

Throughout this thesis, we have used the *pghpf* compiler. Hence it would be desirable to test and evaluate our mechanisms with other HPF compilers too.

Based on this work, it would also be interesting to look at communication between HPF and other parallel programming paradigms such as OpenMP.

# References

[1] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.

[2] Andrew S. Grimshaw and Wm. A. Wulf. Legion–A View from 50,000 Feet. *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, August 1996.

[3] Marc Snir, Stewe W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI - The Complete Reference*. MIT Press, Cambridge, MA, 1996.

[4] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, MA, 1994.

[5] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1), January/March 1998.

[6] A. Geist, A. Beguelin, J. Dongarra, R. Manchek, W. Jiang, and V. Sunderam. *PVM: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, 1994.

[7] Yu Hu. *Efficient Data Parallel Implementations of Highly Irregular Problems*. PhD thesis, Harvard University, Cambridge, MA, April 1997.

[8] MPI Forum. MPI-2: Extensions to the Message-Passing Interface. July 1997.

[9] T. Gross, D. O' Halloron, and J. Subhlok. Task Parallelism in a High Performance Fortran Framework. *IEEE Parallel and Distributed Technology*, 2(3):16–26, Fall 1994.

[10] J. Subhlok and B. Yang. A new model for integrated nested task and data parallel programming. *In Proc. of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 1997.

[11] I.Foster, D. Kohr Jr., R. Krishnaiyer, and A. Choudhary. Double standards: Bringing task parallelism to HPF via the Message Passing Interface. *Proc. Supercomputing '96*, November 1996.

[12] J. Merlin, S. Baden, S. Fink, and B. Chapman. Multiple Data Parallelism with HPF and KeLP. *J. Future Generation Computer Systems*, 15(3):393–405, May 1999.

[13] MPI Forum. MPI: A Message-Passing Interface Standard. June 1995.

[14] High Performance Fortran Forum. High Performance Fortran Language Specification Version 2.0. January 1997.

[15] European Center for Parallelism of Barcelona. High Performance Fortran Compilers Survey. *http://www.ac.upc.es/HPFSurvey*.

[16] High Performance Fortran Forum. HPF: Compilers. *http://dacnet.rice.edu/Depts/CRPC/HPFF/compilers/index.cfm*.

[17] High Performance Fortran Forum. HPF: Applications. *http://dacnet.rice.edu/Depts/CRPC/HPFF/applications/index.cfm*.

[18] High Performance Fortran Forum. High Performance Fortran Language Specification Version 1.1. November 1994.

[19] Mark Nelson, William Humphrey, Attila Gursoy, Andrew Dalke, Laxmikant Kale, Robert D. Skeel, and Klaus Schulten. NAMD: A Parallel, Object-Oriented Molecular Dynamics Program. *International Journal of Supercomputer Applications and High Performance Computing*, 10(4):251–268, 1996.

[20] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations. *J. Comput. Chem.*, 4(2):187–217, 1983.

[21] A. McKenny and R. Pachter. Implementation Issues for Fast Multipole Implementations for Molecular Dynamics Simulations. *SIAM Annual Meeting*, July 1996.

[22] L. Greengard and V. Rokhlin. A Fast Algorithm for Particle Simulations. *J. Comp. Phys.*, 73:325–348, 1987.

[23] L. Greengard and V. Rokhlin. Rapid evaluation of potential fields in three dimensions. Technical Report YALEU/DCS/RR-515, Dept. of Computer Science. Yale University, February 1988.

[24] Christopher R. Anderson. An implementation of the fast multipole method without multipoles. *SIAM J. Sci. Stat. Comp.*, 13(4):923–947, 1992.

[25] Y. Charlie Hu, Lennart Johnsson, and Shang-Hua Teng. High Performance Fortran for Highly Irregular Problems. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, Nevada, June 1997.

[26] Yu Hu and S. Lennart Johnsson. A Data-Parallel Implementation of Hierarchical N-body Methods. *International Journal of Supercomputer Applications and High Performance Computing*, 10(1):3–40, 1996.

[27] The Portland Group Inc. pghpf. *Reference Manual, Version 2.4*, 1998.

[28] D.J. Shippy, T.W. Griffith, and Geordie Braceras. POWER2 Fixed-Point, Data Cache, and Storage Control Units. *http://www.rs6000.ibm.com/resource/technology/fxu.html*.

[29] C.B. Stunkel, D.G. Shea, B. Abali, M.G. Atkins, C.A. Bender, D.G. Grice, P. Hochschild, D. J. Joseph, B. J. Nathanson, R. A. Swetz, R. F. Stucke, M. Tsao, and P. R. Varker. The SP2 High-Performance Switch. *IBM Systems Journal*, 34(2):185–204, 1995.

[30] HP Corporation. HP Message Passing Interface. *http://www.hp.com/rsn/mpi/product_info.html*.