An Implementation of MPI 1.2 for Linux Clusters

by

SADIK GOKHAN CAGLAR

B.S. (Bogazici University) 2000

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in the

DEPARTMENT OF COMPUTER SCIENCE

of the

UNIVERSITY OF SAN FRANCISCO

Approved:

Gregory D. Benson, Assistant Professor (Chair)

Peter S. Pacheco, Professor

Jeff T. Buckwalter, Associate Professor

Jennifer E. Turpin, Dean of College of Arts and Sciences

2004

An Implementation of MPI 1.2 for Linux Clusters

Abstract

USFMPI is an implementation of MPI 1.2, targeted at Linux clusters of symmetric multiprocessors. USFMPI was written from scratch for a very specific, but widely used, type of cluster. Therefore, compared to existing open source or vendor supplied implementations, USFMPI is compact and easy to understand.

Central to the design of USFMPI is the use of a separate communication thread to facilitate the overlap of computation and communication and to take advantage of multiple processors. The communication engine is abstracted out of the rest of the MPI implementation using this separate thread. This separation makes the design easy to understand and helps an implementor to integrate new communication mechanisms into the back-end of USFMPI. Furthermore, the simplicity of the design allows for the analysis of the critical paths and can expose opportunities for specialization if only a subset of the MPI interface is required by an application.

This thesis describes the design and implementation of USFMPI and presents performance results from experiments executed on a 64 node cluster. We show that a compact and specialized implementation can produce better results than MPICH and LAM/MPI for many types of point to point and collective comunication. Finally we identify opportunities for optimization.

To mom and dad for their love and guidance

Contents

List of Figures

List of Tables

Acknowledgements

First I would like to thank Professor Benson for convincing me that we could implement MPI from scratch and for all his support and help throughout the design, implementation and editing processes. I would also like to specially thank Qing Huang for the many bugs he resolved and the enhancements he did to USFMPI, and other members of the USFMPI team Minchuan Tsai and Cho-Wai Chu for their contributions. Also thanks to Alex Fedosov and Cody R. Nivens for maintaining the Keck cluster, and to other members of the Keck cluster group for their input. Finally I would like to thank my other thesis committee members, Professor Peter S. Pacheco and Jeff T. Buckwalter, for their insightful reviews.

Chapter 1

Introduction

Programmers are always striving for an ever increasing amount of computational power. It is crucial to obtain results as soon as possible in many areas driven by real-time constraints. Even if the results are not needed in realtime, the development cycle of any project is positively affected by faster results. A straightforward way to have more computational power is to obtain faster individual computers. However, a more cost-efficient way to achieve this goal is to use multiple computers that work on the same problem simultaneously. Having more than one computer to operate on one problem at the same time is called parallel computing.

Parallel computing can be achieved simply by network programming. However, there are several routines that are helpful to have in a parallel programming library that regular network libraries do not have, such as distributing a C array to the processes that participate in the computation. Several groups and researchers have attempted to standardize interfaces and languages for parallel programming. A popular interface used for parallel programming is the Message Passing Interface (MPI). USFMPI is an implementation of the MPI 1.2 standard [29].

Several publicly available and commercial implementations of MPI exist, MPICH [3] and LAM/MPI [22] being the most popular ones. The internal structure of these implementations is rather complicated due to their development history. Both of these systems are based on existing systems prior to MPI. Our primary goal in designing USFMPI was to come up with a comprehensible implementation that can support various network protocols for communication. USFMPI can be extended to support new protocols and can be manipulated to use different algorithms in different parts of the implementation e.g., collective communication. Another advantage of USFMPI over many existing implementations is its multi-threaded design. Having a separate thread to handle communication not only achieves better performance on SMP systems, but also increases the understandability of the design by separating the communication from the rest of the MPI system.

The thesis is organized as follows. Chapter 2 explains the history of parallel computing, the hardware and software systems that are used for parallel programming, and public and vendor implementations of MPI. Chapter 3 explains the syntax of common MPI functions and how to write applications using MPI. Chapter 4 presents the design of USFMPI and Chapter 5 presents the implementation of USFMPI. Chapter 6 analyzes the experimental results. Chapter 7 offers conclusions about USFMPI.

The MPI standard was formed to satisfy all kinds of requests a parallel programmer might make of a parallel library. This property of the standard has played a major role in it being widely accepted. However, many MPI applications only use a subset of MPI functions. USFMPI implements only these functions so that it can be delivered in a timely manner. The supported MPI functions are listed in Appendix A.

Chapter 2

Background

This chapter introduces fundamental concepts in parallel computing. Various parallel programming paradigms, parallel hardware architectures, and available MPI implementations are examined.

2.1 Parallel Computing

Parallel computing in its most basic form means having more than one operation done simultaneously on one system. With this definition any personal computer with a central processing unit (CPU) and a graphics card employs parallel computing, since there are two processors working on the same problem that are connected via a system bus. However, this definition is very broad. In order to be considered a parallel computing system in the field of scientific computing, a system needs to have two or more CPUs that can communicate via a network or a bus.

2.1.1 The Need for Parallelism

Parallel computing is used for many reasons. First, many computing problems simply require an extremely large number of operations. With the computational power of a single workstation, it may take days in order to reach a solution or a desired result. This is frustrating but acceptable in some areas of research. However, applications such as weather prediction or stock market analysis have a time bound for the usefulness of the results.

Another motivation for parallel computing is the memory problem. Often the data of a problem that needs to be stored in memory is bigger than the memory size of a single system. This problem can be partially solved with the help of virtual memory, in which part of the disk is used for program memory. While virtual memory will accommodate a program whose memory footprint is larger than physical memory, its performance will suffer due to slow disk accesses. Even lower levels of memory, like the hard disks may be incapable of storing all the data since some applications produce terabytes of information. Another issue to keep in mind when dealing with any memory system is the price. As a rule of thumb price/size ratios increase as the size of the memory increases. Therefore, having multiple memory and storage units is not just more efficient but also more economical.

Theoretically, a parallel system that consists of n computers should perform n times better than each of the computers that it is built from. Unfortunately, communication between processors takes a significant amount of time compared to execution time. Even for the type of problems which are considered embarrassingly parallel [10][1] the communication time — for example, to distribute the problem and collect the results— can take a significant proportion of the total execution time.

2.2 Parallel Architectures

Computer systems from the perspective of how they operate on data, can be generalized to four groups: single instruction single data (SISD), multiple instructions single data (MISD), single instruction multiple data (SIMD) and multiple instructions multiple data (MIMD)[2]. The first two types of computations are not considered parallel computing.

In SIMD systems, a single instruction is applied to a vector of data by the use of multiple arithmetic logic units and a CPU. Due to the structure of the system, these type of machines work synchronously, they all do the same operation at the same time. If the nature of the problem requires the same operation to be done on an array of data then these types of systems scale well. The CM-1 [16, 23] and CM-2 [30, 23] produced by Thinking machines and the MP-2 [23, 24] produced by Maspar are good examples of these kind of machines. However, in more complicated programs that require different types of

[1] Embarrassingly parallel problems can be easily divided into completely independent parts that can be executed simultaneously [32].

[2] MIMD applies many operations to each piece of data in turn, is included for completeness, although no machines of this type have been built.

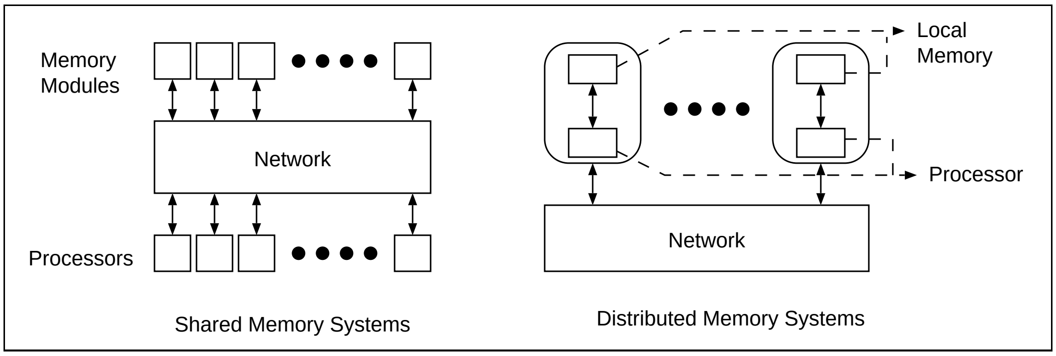

Figure 2.1: Shared Memory and Distributed Memory Architectures

operations on different parts of the array, it is not possible to achieve good parallelism with SIMD machines.

A more interesting and useful way to achieve parallelism is to apply different instructions to different data. The systems that use this technique are called Multiple Instruction Multiple Data (MIMD) systems. A multiple number of processing units perform asynchronously and independently. Depending on the memory organization, MIMD systems are divided into two types: shared memory and distributed memory.

2.2.1 Shared Memory Systems

The classical stored program architecture has a single CPU and a single memory unit that are connected via a system bus. In order to achieve parallelism, this architecture can be extended with multiple CPUs that form a processor group and multiple memory units that form a memory group as shown in the left side of Figure 2.1. The memory group is perceived as a single memory space by each processor, therefore each memory location has a unique address and can be accessed by any processor.

The bus in a single memory system can be replaced by a more complex interconnection network. Shared memory systems are distinguished by the type of this network. Two popular types are *bus based* and *switched* as seen in Figure 2.2. In the first type, a single bus is shared between multiple processors. Although this system is the easiest one to implement, the bus is likely to get congested, introducing long delays serving memory requests from the processors. Another disadvantage of these systems is the limited bandwidth of the bus. These systems simply do not scale well to large numbers of processors. As such, bus based systems tend to have a small number of processors (32 to 64).

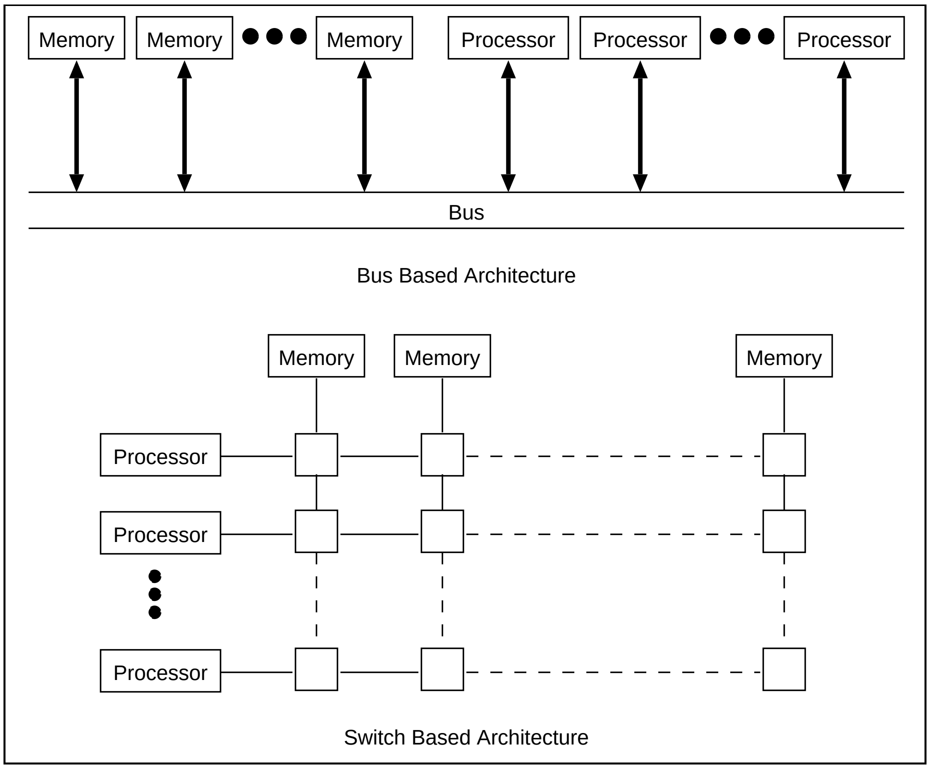

Figure 2.2: Bus Based and Switched Shared Memory Architectures

One or more switches are used instead of a single bus in switch based networks. In these networks a rectangular mesh of buses or a multistage network connect the processors to the memory units, having processors on one side of the rectangle and the memory units on a perpendicular side. A switch is present at every intersection of the mesh, so that every memory unit can be accessed by a processor without restricting other processors from accessing different memory units. Apart from being more complicated to build, the main disadvantage of these networks is economical. A system with p processors and m memory units would require $p \times m$ switches, which increases the price of the system considerably.

The main disadvantage on shared memory systems is the cache coherence problem. It is possible that one process can invalidate some data that is present in some other process's cache. Systems that use the single bus approach can use the snoopy protocol that monitors the bus all the time for memory write operations. However, this technique is not suitable for switched based networks in which more costly algorithms have to be used.

2.2.2 Distributed Memory Systems

Distributed memory systems consist of nodes, each of which has its own processor[3] and memory unit, and a network that connects these nodes. In the perfect case every node should be connected to every other node, however this approach results in expensive systems. Distributed memory systems can be classified by the type of network they use: dynamic, static, and bus.

Networks that use switches are called dynamic interconnection networks. These networks are likely to have the same form as the crossbar of switches found in shared memory systems. The only difference is that these systems have the same type of nodes on both sides of the rectangle. Since they have the same form, these systems are also very costly. Special networks like multistage networks [25] may be useful in bringing the cost down.

The networks that have dedicated links between their nodes are called static interconnection networks. If the network is not fully connected, messages have to be forwarded to the appropriate processor by intermediate processors. A linear array, ring, hypercube, two dimensional mesh, three dimensional mesh [23] are the topologies that are commonly used. Figure 2.3 illustrates some of the statically linked topologies.

[3]Symmetric Multiprocessor (SMP) nodes have more than one processor on the same node sharing the same memory unit.

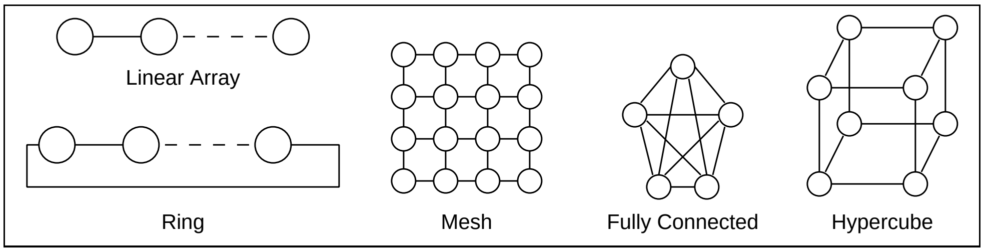

Figure 2.3: Common Statically Linked Network Topologies

The last way to connect the nodes of the system is to use a single bus. A cluster of computers that are connected via Ethernet is a popular example for such networks. This approach suffers from congestion in the single bus as it does in shared memory systems, but it is the cheapest and easiest type of system to build.

2.2.3 Clusters as Multicomputers

The computer systems discussed in sections 2.2.1 and 2.2.2 can generally be classified as Massively Parallel Processors (MPPs)[4]. The users reach these systems via a proxy machine and the rest of the system is seen as a box full of dedicated processors with a fast interconnect. Clusters are a collection of independent computers that are connected by a local area network. The main difference between clusters and MPP systems is the way they are administrated and sold. MPPs are built by large companies like HP or SGI and are sold as a single system. With clusters, the processors and the network can be purchased separately. Therefore, it is possible that a cluster may consist of heterogeneous hardware. With the help of low cost, high performance SMP systems like dual Intel Pentiums, clusters provide parallelism with a better price/performance ratio than MPPs [27]. As such, the market is shifting from expensive MPPs to inexpensive clusters.

Apart from the economic reasons, clusters have two other important advantages over MPPs. First, the hardware, especially the processors, can easily be upgraded as the new, faster products become available. Second, the system users are not bound to the software that is provided by the vendor. Choosing software independent of the system's vendor is likely to decrease the cost of the system since there are high performance, free operating systems such as Linux [2, 31] or FreeBSD [1] and message passing systems such as MPICH [3]. These types of clusters, which are built up of commodity hardware and run free

[4]The MPPs that are mentioned are commercially produced systems.

operating systems are generally called Beowulf clusters [26, 27]. Beowulf clusters usually use Fast Ethernet, which provides a data rate of 100 Megabits per second, and Gigabit Ethernet, which provides a data rate of 1 Gigabit per second, for their communication layer. Myrinet [5] is another cost effective network alternative that is used in Beowulf clusters. Myrinet provides full duplex transmission at 2 gigabits per second.

2.3 Parallel Programming Paradigms

Parallel programming languages can be classified in multiple ways. The most general way of classifying them is to consider the underlying memory model: shared or distributed. It is important to explicitly point out the difference between shared and distributed memory programming and shared and distributed memory architectures. Naturally, it makes sense to setup a shared memory programming environment on a shared memory system and a distributed programming environment on a distributed memory system. However, the programming environments are at an abstract level. That is, a distributed memory architecture can support a shared memory programming environment and vice versa. The programming environment will take care of internal message passing to present the distributed memory as a single address space.

2.3.1 Shared Memory Programming

One of the many ways to program a parallel application on a shared memory system is to use Unix processes. Most modern operating systems are capable of multitasking, which means more than one process (task) can run at the same time. If the system has multiple symmetric processors, multitasking can be used to implement parallel programming. The assignment of tasks to processors is done automatically by the operating system. The data communication operations are done explicitly via system pipes or Unix domain sockets.[5]

POSIX threads (Pthreads) [21] is another mechanism for shared memory programming. Threads exist within a Unix process and use the process's resources, but they have their own stack, scheduling properties, set of pending and blocked signals and thread specific data. These properties enable the thread to be scheduled and run independently of the process to which it belongs. Since they can be scheduled separately, threads can be used

[5]It is possible to share memory using shared memory regions, however it is cumbersome to use that technique.

for parallel programming just like Unix processes. Using threads instead of separate Unix processes has significant advantages. First creating a new process takes more time due to the extra system resources that need to be initialized. Threads can use system pipes and domain sockets for communication like processes, but since they share the same memory address space, they can have pointers to shared memory locations. Being able to write to and read from the same location results in significant performance gains. The only drawback is that these shared memory locations must be explicitly synchronized by the programmer. Other system resources like files can be shared by the threads, but they have to be explicitly synchronized as well.

OpenMP [17] is a standardized API for parallelizing Fortran, C, and C++ programs on shared memory architectures. OpenMP's most important capability is that it can be used to incrementally parallelize a serial program. OpenMP employs fork-join parallelism in which a master thread runs the serial part of the code as a usual program. When a region needs to be executed in parallel, like a for loop, the master thread automatically forks child threads, distributes the data appropriately among itself and the children, and gathers the data at the end of the for loop. This type of programming is easier to use and more fool-proof than any explicit parallelism like manually managing the threads or explicit message passing. However, automatic parallelization leaves the programmer with fewer capabilities to fine-tune her program or use the algorithms she devised for distributing the data.

High Performance Fortran (HPF) [15] can also be used in a distributed memory system. HPF is listed under this section due to the data management being automated by the system rather than by explicit programmer effort. HPF implements a data parallel approach. Unlike OpenMP the data is distributed at the beginning of execution; therefore all of the threads are alive from the beginning to the end of the program. HPF automatically manages data consistency, which results in implicit synchronization. Since there is no explicit message passing, the HPF model is similar to that of OpenMP. The programs are very likely to run and produce correct results; however, if the programmer does not choose a proper distribution for the data, an HPF program may turn out to be very inefficient due to the implicit synchronization every time processes need to communicate. Also data parallel programming of highly irregular problems is not trivial [18].

2.3.2 Distributed Memory Programming

In distributed memory programming, each process has its own private data. In order to share data between processes, explicit message passing must be done by the programmer. Since each process has its own data, these languages are of the MIMD classification. Two approaches exist to structure the executable code in these languages. The most obvious way is the multiple program multiple data (MPMD) structure, in which each process has its own executable code. This approach turns out to be inconvenient in practice, since the same program could be run on a changing number of processes to accommodate the varying number of available processors in a system. It is desirable to structure a program so that it scales with the number of processors available. The single program multiple data (SPMD) structure does not suffer from this problem. As the name implies, there is only one program source for all processors. In order to have different behavior for different processes, the source code can be constructed to behave differently depending on the identity of the computer; e.g., all of the processors that do not have process rank 0 sends a message to process 0 and process 0 receives a message from every other process.

Explicit message passing systems can lead to more efficient programs than the implicit message passing systems. However, it is more complicated and error-prone. Deadlocks can easily occur if the programmer is not experienced enough. In many ways it is analogous to assembly language programming. The program can execute a lot faster than a program built in a high level programming language, but if the programmer does a poor job, a higher level language with a good compiler may perform better.

The first widely adopted programming environment to use a workstation cluster as a multicomputer was PVM (Parallel Virtual Machine) [20]. A daemon that takes care of PVM operations like spawning processes and message passing between processes runs on every workstation that participates in the system. Depending on the programmer's choice, the PVM system can take care of process creation and the initial setup. PVM can be run in three modes: transparent mode, architecture-dependent mode, and low-level mode. The transparent mode is the highest level; tasks are automatically assigned to the most appropriate computer. In the architecture-dependent mode, the user is able to specify the type of the processor on which each process is going to be executed. In the low-level mode, the user can specify a particular processor to execute a task.

The programs that PVM executes are written in C or Fortran, and compiled for the architectures on which the programs are going to be run. PVM provides routines for common message passing operations. For portability, these operations are decoupled from a specific transport protocol like TCP or UDP. It also handles process to processor assignments, which is normally done by the operating system.

MPI is another widely adopted programming environment for distributed memory systems. MPI provides much of the same functionality that PVM does, including the introduction of an abstract level for message passing and automatic process assignment to processors. MPI's history evolved with MPICH as explained in section 2.4.1.1. The rest of this thesis is about MPI since we decided to implement the MPI standard for our parallel programming environment. The main reason for this is that MPI is a distributed memory programming language that has proved to be more popular than PVM for parallel programming. In comparison, MPI has many advantages over PVM. Here are the ten most popular advantages that have been surveyed by the LAM/MPI research group:

1. MPI has more than one freely available, quality implementation. There are at least LAM, MPICH, and CHIMP. The choice of development tools is not coupled to the programming interface.

2. MPI defines a 3rd party profiling mechanism. A tool builder can extract profile information from MPI applications by supplying the MPI standard profile interface in a separate library, without ever having access to the source code of the main implementation.

3. MPI has full asynchronous communication. Immediate send and receive operations can fully overlap computation.

4. MPI groups are solid, efficient, and deterministic. Group membership is static. There are no race conditions caused by processes independently entering and leaving a group. New group formation is collective and group membership information is distributed, not centralized.

5. MPI efficiently manages message buffers. Messages are sent and received from user data structures, not from staging buffers within the communication library. Buffering may, in some cases, be totally avoided.

6. MPI synchronization protects the user from 3rd party software. All communication within a particular group of processes is marked with an extra synchronization variable, allocated by the system. Independent software products within the same process do not have to worry about allocating message tags.

7. MPI can efficiently program MPP and clusters. A virtual topology reflecting the communication pattern of the application can be associated with a group of processes. An MPP implementation of MPI could use that information to match processes to processors in a way that optimizes communication paths.

8. MPI is totally portable. One can recompile and run on any implementation. With virtual topologies and efficient buffer management, for example, an application moving from a cluster to an MPP could even expect good performance.

9. MPI is formally specified. Implementations have to live up to a published document of precise semantics.

10. MPI is a standard. Its features and behavior were arrived at by consensus in an open forum. It can change only by the same process.

2.4 MPI Implementations

MPI is a standard developed by the MPI Forum [29], similar to the POSIX standard and the ANSI C standard. To be useful, MPI must be implemented and several MPI implementations exist, including publicly available versions, vendor supplied versions and commercial versions. This section focuses on the two publicly available implementations of MPI: MPICH and LAM/MPI. In addition, commercially available and vendor supplied implementations are briefly examined.

2.4.1 Publicly Available Implementations of MPI

Many MPI implementations are non-commercial and available for public use. The two most popular implementations, MPICH and LAM, are described in this section.

2.4.1.1 MPICH

MPICH [3] is the first and the most popular implementation of MPI. The chameleon inspired the "CH" in the name, as the creature is adaptable to different environments; it symbolizes the portability of MPI. Also a chameleon is a fast animal, which represents the requirement for performance. Gropp and Lusk implemented MPI as the MPI Forum started shaping the specification. This effort helped to expose possible implementation difficulties that could arise. The very first version was built on top of existing stable code, so a portable and efficient implementation was ready shortly after the MPI forum began meeting. This project evolved as the specification changed and the first version was ready as soon as the MPI standard was released in May 1994 [9].

The fact that MPICH was developed at the same time the specification was formed greatly contributed to both MPI's and MPICH's popularity. A fully implemented library, which performed quite well, was available on multiple platforms as soon as the specification was released. In contrast; the implementation of High Performance Fortran (HPF) did not start until the specification was fully released, thus delaying the adoption of HPF until an implementation was complete in 1996. Thus, most of the scientific community had actively used MPI for over a year by the time HPF was ready.

MPICH's main challenge was having an implementation that could be portable to many different hardware platforms while taking advantage of their specific features. To tackle this challenge, the internal structure of MPICH was designed as a number of layers. The upper layers implemented the high-level functionalities, like broadcasting a message, using the lower layers. The higher levels do not need to be reimplemented for new platforms, hence code sharing across platforms is maximized.

The highest level of MPICH consists of the MPI routines such as collective and point to point operations. This level is built on top of the Abstract Device Interface (ADI) [12]. The ADI is intended to achieve a portable implementation with good performance. It is seen as a virtual machine so the high-level portion of the implementation stays unchanged for all the different processor and communication architectures. The design of the ADI layer and the layers beneath it do not assume that they will be used to implement MPI, hence the ADI can be used to build any other high-level specification for message passing. The ADI can be implemented for any low-level system using that level's specific functions, this approach leads to very efficient implementations but it takes a significant amount of time. The channel interface [13] can be used to port the ADI easily to any low-level system. Five functions need to be implemented for the channel interface to work with any hardware transport layer. The functions `MPID_SendControl()`, `MPID_RecvAnyControl()`, and `MPID_ControlMsgAvail()` are used for control messages; `MPID_SendChannel()` and `MPID_RecvFromChannel()` send and receive data respectively. After the new port is quickly implemented using the channel interface, performance can be enhanced by providing a new hardware-specific back end to the ADI.

Chameleon, which is a thin layer of C functions and mostly macros, was used as a lower level to the channel interface. Chameleon relies on P4 [7], which is a parallel programming library, to handle its communication requests. A number of platforms including the RS6000, Sun, Dec, HP, SGI were supported within a couple days for the first implementation of MPICH using these two, already stable systems. Later, TCP was supported as a lower layer to Chameleon so that heterogeneous environments could be handled easily.

2.4.1.2 LAM-MPI

LAM [22] is a parallel software environment that provides a messaging layer for MPI. LAM's native message passing library is designed to fulfill the requirements of applications that run on massively parallel processors (MPPs) [4].

LAM is a subset of the Trollius system[6] [6, 8], which runs only on UNIX clusters. If LAM is run on Symmetric Multiprocessor (SMP) machines, each of these machines is seen as a node. The programs that run on these machines are called processes and the data transmitted through the network are called messages. Machines are assigned node identifiers and processes use UNIX process identifiers. Node identifiers are used to route the messages to the correct node. One or more processes may run on the same node at the same time, this prompts the need for another mechanism to match the messages to correct processes. An event mechanism is implemented to take care of this problem. Processes have to specify an event in order to receive messages that are bound to that event.

A LAM daemon runs on every node that participates in communication. It consists of a micro-kernel and about a dozen system processes. These processes are internally scheduled by the micro-kernel, so that there is no external context switching. The micro-kernel also interacts with external processes, which are typically application processes that

[6]Trollius is a uniform foundation for message passing that can run natively on multicomputers.

use LAM for communication. The external processes interact through UNIX system calls. The communication is handled by the internal processes, not the micro-kernel, therefore the communication preferences can be changed, recompiled, and executed without affecting the rest of the system. These internal processes can also be executed as separate processes at the cost of degraded performance.

On a virtual level, LAM processes[7] see a link to every other process involved in the computation. These links are managed via the route daemon, which is one of the internal processes in the micro-kernel. LAM multiplexes the messages through one pair of datalink processes. The UDP/IP protocol, with a special error correction and flow control level, is used by these datalink processes in order to achieve a reliable communication channel.

Critical to LAM's design is the internal buffering of all incoming messages. The buffer daemon gathers the messages that are received and stores them. The processes treat this daemon as a database, sending queries to probe or receive the requested messages. An asynchronous flow-control mechanism is triggered when the buffer reaches a preset level and control messages are sent to other processes notifying them to stop sending messages. It is expected that there might be some delays propagating these messages to other processes, so the incoming messages continue to be buffered up to a hard limit. From this point only control messages are received.

MPI can easily be implemented on top of LAM. MPI processes can behave as LAM processes with a few modifications to the point to point communication functions. The rest of the communication is handled through these functions and the environment modifier functions that are related to group and communicator information, process ranks, etc. are stored locally and do not need any LAM interaction. The standard and ready mode MPI send functions use LAM network send; buffered mode MPI functions use nonblocking LAM network send; and synchronous mode MPI send functions use LAM network send. There is only one type of MPI receive which maps to LAM network receive and the MPI probe function maps to LAM network probe.

2.4.2 Vendor Supplied Implementations of MPI

Hardware vendors that manufacture parallel architectures usually supply their own proprietary operating systems. They might also choose to implement their own low-level

[7]UNIX applications that use LAM to communicate.

network messaging system. In order to take advantage of the specific operating system and low-level messaging features, vendors usually provide their own MPI implementation as well.

IBM is one of the strongest supporters of MPI. IBM's proprietary library MPL has been replaced by MPI as the preferred library on IBM SP systems. IBM's first implementation of MPI, MPI-F was based on MPICH, but the currently available version is written from scratch. IBM MPI works on IBM SP systems and AIX workstation clusters. The latest version of MPI released by IBM runs on PE for AIX 5L [19].

The MPI implementation for Hewlett Packard systems is called HP MPI [14]. It is derived from MPICH, but significantly influenced by LAM/MPI. HP MPI takes advantage of the shared-memory architecture of HP servers to provide low-latency, high-bandwidth message-passing performance. An application executed on multiple servers passes messages using shared memory within each host, and TCP/IP between hosts. If appropriate hardware is installed, HP MPI can perform intra-host communication using HP's low-latency networking protocol HMP.

Sun also provides its own MPI implementation, it is called Sun MPI [28] (the current version is 6.0). Sun MPI is optimized for running with Sun HPC ClusterTools 5 software. It has multi-protocol support so that the fastest available medium is selected for each type of connection e.g., shared memory, RSM, and ATM.

2.4.3 Commercial Implementations of MPI

Commercial implementations of MPI are available from several companies. Here are the most popular currently available commercial implementations:

- MPI/Pro by MPI Software Technology, Inc., available for Windows, Linux, Mercury, and Mac OS X systems.
- WMPI II by Critical Software S.A., available for Windows platforms (NT/2000/XP).
- ScaMPI by Scali, available for Solaris, Linux and Windows NT.

Unfortunately, there is no publicly available technical information for these systems.

Chapter 3

MPI Programming

MPI is available as a library for C, C++, and Fortran. This chapter describes how to write parallel programs using MPI for C. In addition, this chapter summarizes the most commonly used MPI functions.

3.1 Compiling and Running MPI Programs

An MPI library is linked into a program differently than the conventional `-l<Library Name>` approach. Most implementations, including USFMPI, use a special script called `mpicc` that wraps the standard C compiler of the system. Since most of today's Linux clusters are homogeneous systems, the code only needs to be compiled once. Otherwise, the source code has to be compiled for every architecture that is going to participate in the execution of the program.

MPI executables have to be run with a special program called `mpirun`. Processes can not be dynamically spawned or exited in MPI-1, hence a static number of processes are involved in each computation. The MPI user must provide the number of processes that she wants to use in a particular execution. For example, `mpirun -np 4 test` is used to run the executable named `test` with four processes.

3.2 A Simple Program and Required Calls

Figure 3.1 lists a trivial program that does a ping-pong operation between processes 0 and 1.

```
#include <stdio.h>
#include <mpi.h>
static int SIZE = 1024;  /* The size of the buffer. */
int main(int argc, char** argv)
{
  int nump;           /* number of processes in the system.       */
  int rank;           /* rank of the current process.             */
  int tag;            /* tag that is used for messaging.          */
  int peer;           /* other party in communication.            */
  MPI_Status status;  /* status of the received message           */
  int buffer[SIZE];   /* buffer that is used for message transfer */

  /* MPI is initialized. */
  MPI_Init(&argc, &argv);
  tag = 1;
  /* Number of process's in the system. */
  MPI_Comm_size(MPI_COMM_WORLD, &nump);
  /* Current process's rank. */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /* Processor 0 sends the message, processor 1 receives,
     then vice-versa. */
  if (rank == 0) {
    peer = 1;
    MPI_Send(buffer, SIZE, MPI_INT, peer, tag, MPI_COMM_WORLD);
    MPI_Recv(buffer, SIZE, MPI_INT, peer, tag, MPI_COMM_WORLD, &status);
  } else if (rank == 1) {
    peer = 0;
    MPI_Recv(buffer, SIZE, MPI_INT, peer, tag, MPI_COMM_WORLD, &status);
    MPI_Send(buffer, SIZE, MPI_INT, peer, tag, MPI_COMM_WORLD);
  }
  /* MPI is finalized. */
  MPI_Finalize();
  return 0;
}
```

Figure 3.1: A Simple MPI Program

MPI programs require the `mpi.h` header file to be included in the source code. When an MPI program starts, before doing anything else, it should call the `MPI_Init()` function. This is critical, most of the environment variables and even the command line parameters (`argc` and `argv`) are meaningless before the execution of this function. The details of this function are system dependent and transparent to the user. For example, a system like `MPICH` might initialize TCP sockets and internal data structures. Also, the original command line parameters might be manipulated by `mpirun`. For example, the default TCP port may be passed as a system argument that is removed by `MPI_Init()`.

Parallel programs are engineered to be capable of running with a variable number of processors. In order to figure out the number of processors in the system, `MPI_Comm_size()` is used. However, a communicator must be passed to this function. If the user's concern is to find out all the processors that participate in the computation, `MPI_COMM_WORLD` can be used. Communicators will be explained in more detail in section 3.6.

`MPI_Comm_rank()` is used to determine the current process's rank, which is simply the process ID. Similar to `MPI_Comm_size()`, a communicator must be provided. The reason is that a processor may have different ranks in different communicators.

Before the processes complete execution, the MPI system must be notified. This is done by calling the `MPI_Finalize()` function. Using this is simpler than initialization as it does not require any parameters. Like `MPI_Init()`, the function is implementation dependent and transparent to the programmer. During finalization, the system is likely to close the communication links and deallocate internal data structures.

3.3 Communication Routines

This section explains the core MPI communication routines. Different types of point to point and collective communication functions are explained.

3.3.1 Point to Point Communication

Point to point communication routines can be categorized in two ways. First, a routine either sends or receives. Second, a routine is either blocking or nonblocking. The second approach is taken in this document due to the existence of some helper routines that do not send or receive messages, but are related to communication by their blocking or nonblocking property.

3.3.1.1 Blocking Routines

Blocking routines do not return until the operation is completed. This property is useful when the buffer that is used in communication is used or changed right after the communication. Blocking routines are often the easiest to use because they have simple semantics. However, it also means that communication and computation can not be overlapped. In most cases the nonblocking counterparts should be used even though they are more complicated.

```
int MPI_Send (
  void*         mes      /* message to be sent                     */,
  int           count    /* number of elements in the message      */,
  MPI_Datatype  type     /* type of the elements in the message */,
  int           dest     /* destination of the message             */,
  int           tag      /* tag value                              */,
  MPI_Comm      comm     /* communicator                           */)
```

Figure 3.2: MPI_Send Syntax

MPI defines four different types of send operations. The types of arguments that are passed to the functions are exactly the same for all of them, hence only the standard send is shown in Figure 3.2. `MPI_Send()` is used for the standard send, `MPI_Bsend()` is used for buffered send, and `MPI_Ssend()` is used for synchronous send, `MPI_Rsend()` is used for ready send. The different modes work as follows:

- **Standard mode:** This mode is implementation dependent, the system chooses between internally buffering the data and waiting until the operation is complete.
- **Buffered mode:** The user must provide a buffer for this mode. If there is no matching receive post, the message will be stored in the buffer and the send will return immediately.
- **Synchronous mode:** The send operation does not start until a matching receive is posted. Synchronous mode send returns when the receive is started.
- **Ready mode:** If a matching receive has already been posted in the system, ready mode can be used. If a matching receive has not been posted, the program will be erroneous. This mode returns immediately.

In contrast to having multiple send routines, MPI provides only one routine to receive a message (see Figure 3.3) and it returns only when the message is completely received.

```
int MPI_Recv (
  void*         mes      /* message to be received             */,
  int           count    /* number of elements in the message  */,
  MPI_Datatype  type     /* type of the elements in the message */,
  int           source   /* source of the message              */,
  int           tag      /* tag value                          */,
  MPI_Comm      comm     /* communicator                       */,
  MPI_Status*   status   /* status pointer                     */)
```

Figure 3.3: MPI_Recv Syntax

One extra argument is used in receive operations: the status object. This object stores valuable information about the received message. For example, if any wildcards[1] are used for the tag or source values, the actual value can be obtained. It is unacceptable to receive a message that is longer than the specified size in `MPI_Recv()` due to a possible buffer overflow. However it is acceptable to receive a shorter buffer. In this case, `MPI_Get_Count()` (see Figure 3.4) is used to determine the actual message size.

```
int MPI_Get_count(
  MPI_Status*  status   /* status pointer                  */,
  MPI_Datatype datatype /* type of the data received        */,
  int*         count    /* number of elements received      */)
```

Figure 3.4: MPI_Get_count Syntax

Probing is often useful if the programmer wishes not to allocate a buffer until there is something to receive. `MPI_Probe()` (see Figure 3.5) will block until a message that matches the probe arguments is received.

[1] MPI_ANY_SOURCE is used to receive from any process and MPI_ANY_TAG is used to receive a message with any tag.

```
int MPI_Probe(
  int         source  /* source that posted the send       */,
  int         tag     /* tag of the posted message         */,
  MPI_Comm    comm    /* communicator of the posted message */,
  MPI_Status* status  /* status pointer                    */)
```

Figure 3.5: MPI_Probe Syntax

3.3.1.2 Nonblocking Routines

Nonblocking routines return immediately, hence the operation is usually not complete when a nonblocking routine returns. All of the blocking point to point functions have nonblocking counterparts, all of the arguments are used the same way. `MPI_Isend()` (see Figure 3.6) is used for the standard send, `MPI_Ibsend()` is used for the buffered send, `MPI_Issend()` is used for the synchronous send, and `MPI_Irsend()` is used for ready send. As in blocking communication, there is only one type of nonblocking receive (see Figure 3.7).

```
int MPI_Isend (
  void*         mes      /* message to be sent                    */,
  int           count    /* number of elements in the message     */,
  MPI_Datatype  type     /* type of the elements in the message */,
  int           dest     /* destination of the message            */,
  int           tag      /* tag value                             */,
  MPI_Comm      comm     /* communicator                          */,
  MPI_Request*  request  /* request pointer                       */)
```

Figure 3.6: MPI_Isend Syntax

```
int MPI_Irecv (
  void*         mes      /* message to be received             */,
  int           count    /* number of elements in the message   */,
  MPI_Datatype  type     /* type of the elements in the message */,
  int           source   /* source of the message               */,
  int           tag      /* tag value                           */,
  MPI_Comm      comm     /* communicator                        */,
  MPI_Status*   status   /* status pointer                      */,
  MPI_Request*  request  /* request pointer                     */)
```

Figure 3.7: MPI_Irecv Syntax

The nonblocking calls have an extra argument that allows the programmer to refer to the operation later on during execution. The `MPI_Request` structure is used for this purpose. Two functions use this structure to determine if the operation is complete. First, `MPI_Wait()` (see Figure 3.8) can be used to wait until the operation is complete. That is, when `MPI_Wait()` is used the nonblocking communication turns into a blocking communication. Second, if the programmer wishes only to test but not block on the communication using the request object, she can use the `MPI_Test()` function (see Figure 3.9). This function returns a boolean value indicating the completion of the communication.

`MPI_Iprobe()` (see Figure 3.10) is the nonblocking version of `MPI_Probe()`. It is used to determine if a message can be received with the specified arguments as described in the previous section.

```
int MPI_Wait(
  MPI_Request*  request  /* request pointer */,
  MPI_Status*   status   /* status pointer  */)
```

Figure 3.8: MPI_Wait Syntax

```
int MPI_Test(
  MPI_Request*  request  /* request pointer                  */,
  int*          flag     /* true if operation is complete  */,
  MPI_Status*   status   /* status pointer                   */)
```

Figure 3.9: MPI_Test Syntax

```
int MPI_Iprobe(
  int         source  /* source that posted the send              */,
  int         tag     /* tag of the posted message                */,
  MPI_Comm    comm    /* communicator of the posted message       */,
  int*        flag    /* true if there us a message to receive    */,
  MPI_Status* status  /* status pointer                           */)
```

Figure 3.10: MPI_Iprobe Syntax

3.3.2 Collective Communication

Some message passing patterns are done very frequently in distributed computing, such as distributing an array, gathering an array, and broadcasting a value. It is convenient to have these routines built into the message passing library. Another advantage of having pre-built routines is that most of these operations can be optimized if the network topology is known. Since MPI programs are designed to be portable, the optimization can be handled at the MPI implementation level.

Collective routines are usually built on top of the MPI point to point routines. Various algorithms exist for each and the MPI standard does not enforce any specific one to be used. The algorithms that fit the underlying network topology best are selected for a particular implementation of MPI.

Collective operations do not use a tag argument, so the only way to differentiate the operations is to create new groups and communicators. All of the processes in the group used when calling a collective routine must participate in the operation. If one of the processes fails to issue a call to the collective routine, or calls a different one than every other process, the program is erroneous.

In terms of message passing behavior, there are three different types of collective operations: one to many, many to one, and many to many. Broadcast and scatter are one to many; gather and reduce are many to one; and barrier, allgather, alltoall, allreduce, reduce_scatter, and scan are many to many. Apart from being a multiple number of point to point communications, collective communications may be built as variations of four routines: broadcast, gather, scatter and reduce. As an example, a barrier call can be built using a broadcast followed by a reduce.

The routines that are of the type one to many or many to one require a master node that is called the root node, which is the center of operation. The root node holds the data to be broadcast or scattered, or needs to gather data. All of the processes in the group must specify the same rank as the master node.

Four of the operations can be called with variable buffer sizes in each process, these are: gather, scatter, allgather, and alltoall. A v character is added to the name of the original function, e.g., `MPI_Scatter()` and `MPI_Scatterv()`. The difference between the two operations can be observed in Figures 3.17 and 3.18. In `MPI_Scatter`, only a single integer for `send_count` is specified, because every process will be sent the same amount of data. However the `MPI_Scatterv` operation needs two arrays, `send_counts` and `displs`, to specify how many data items each process needs to receive. Each entry in the `send_counts` array shows the number of items the corresponding process is going to receive. The entries in the `displs` array indicate where the data for the corresponding process starts in the send buffer.

3.3.2.1 Barrier

All of the processes are blocked until every process in the group described by the communicator calls the barrier routine (see Figure 3.11).

```
int MPI_Barrier(
  MPI_Comm      comm       /* communicator */)
```

Figure 3.11: MPI_Barrier Syntax

3.3.2.2 Broadcast

The contents of the buffer in the root process are sent to every other process that participates in the communication (see Figure 3.12).

```
int MPI_Bcast(
  void*         buf       /* buffer to be sent or received         */,
  int           count     /* number of elements in the buffer      */,
  MPI_Datatype  type      /* type of the elements in the buffer    */,
  int           root      /* master process that will has the data */,
  MPI_Comm      comm      /* communicator                          */)
```

Figure 3.12: MPI_Bcast Syntax

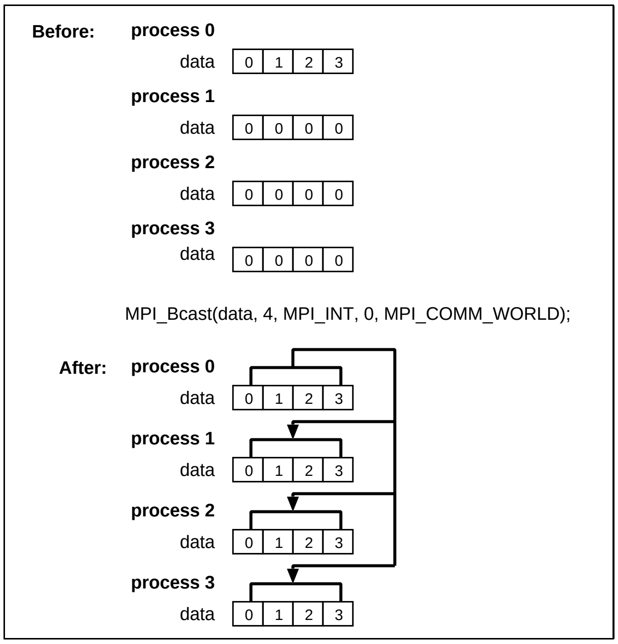

Figure 3.13: MPI_Bcast Illustrated

3.3.2.3 Gather

The contents of each process's send buffer are gathered at the receive buffer of the root process (see Figures 3.14 and 3.15).

```
int MPI_Gather(
  void*         send_buf   /* buffer to be sent                     */,
  int           send_count /* number of elements in the send_buf   */,
  MPI_Datatype  send_type  /* type of the elements in the send_buf */,
  void*         recv_buf   /* buffer for the received data         */,
  int           recv_count /* number of elements in the recv_buf   */,
  MPI_Datatype  recv_type  /* type of the elements in the recv_buf */,
  int           root       /* master process, will have final data */,
  MPI_Comm      comm       /* communicator                         */)
```

Figure 3.14: MPI_Gather Syntax

```
int MPI_Gatherv(
  void*         send_buf    /* buffer to be sent                      */,
  int           send_count  /* number of elements in the send_buf     */,
  MPI_Datatype  send_type   /* type of the elements in the send_buf   */,
  void*         recv_buf    /* buffer for the received data           */,
  int*          recv_counts /* number of elements in the recv buffers */,
  int*          displs      /* displacements array                    */,
  MPI_Datatype  recv_type   /* type of the elements in the recv_buf   */,
  int           root        /* master process, will have final data   */,
  MPI_Comm      comm        /* communicator                           */)
```

Figure 3.15: MPI_Gatherv Syntax

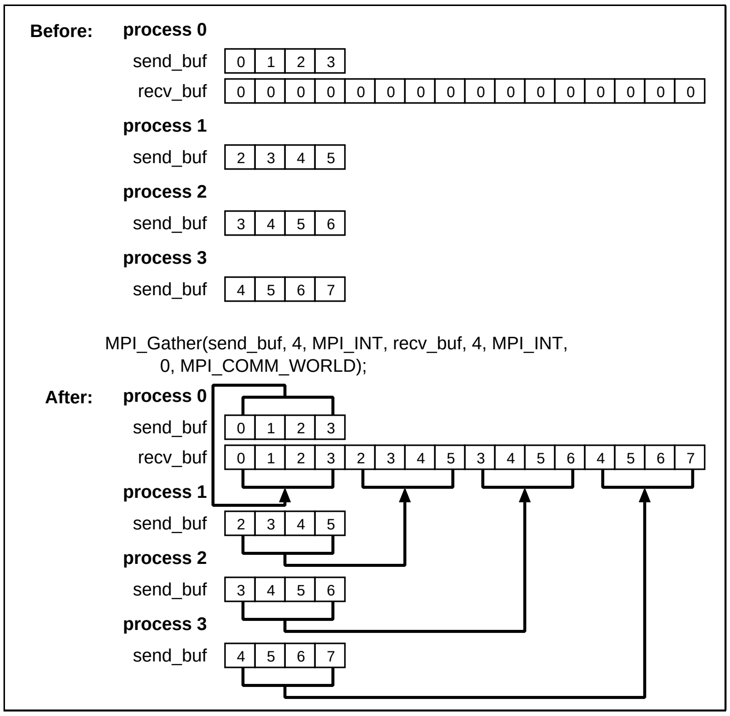

Figure 3.16: MPI_Gather Illustrated

3.3.2.4 Scatter

The contents of the send buffer of the root process is scattered among all the other processes that are involved in the communication (see Figures 3.17 and 3.18).

```
int MPI_Scatter(
  void*         send_buf   /* buffer to be sent                      */,
  int           send_count /* number of elements in the send_buf    */,
  MPI_Datatype  send_type  /* type of the elements in the send_buf */,
  void*         recv_buf   /* buffer for the received data          */,
  int           recv_count /* number of elements in the recv_buf    */,
  MPI_Datatype  recv_type  /* type of the elements in the recv_buf */,
  int           root       /* master process that has final data    */,
  MPI_Comm      comm       /* communicator */ )
```

Figure 3.17: MPI_Scatter Syntax

```
int MPI_Scatterv(
  void*         send_buf    /* buffer to be sent                       */,
  int*          send_counts /* num of elements in the send buffers  */,
  int*          displs      /* displacements array                     */,
  MPI_Datatype  send_type   /* type of the elements in the send_buf */,
  void*         recv_buf    /* buffer for the received data          */,
  int           recv_count  /* number of elements in the recv_buf   */,
  MPI_Datatype  recv_type   /* type of the elements in the recv_buf */,
  int           root        /* master process that has final data   */,
  MPI_Comm      comm        /* communicator                            */)
```

Figure 3.18: MPI_Scatterv Syntax

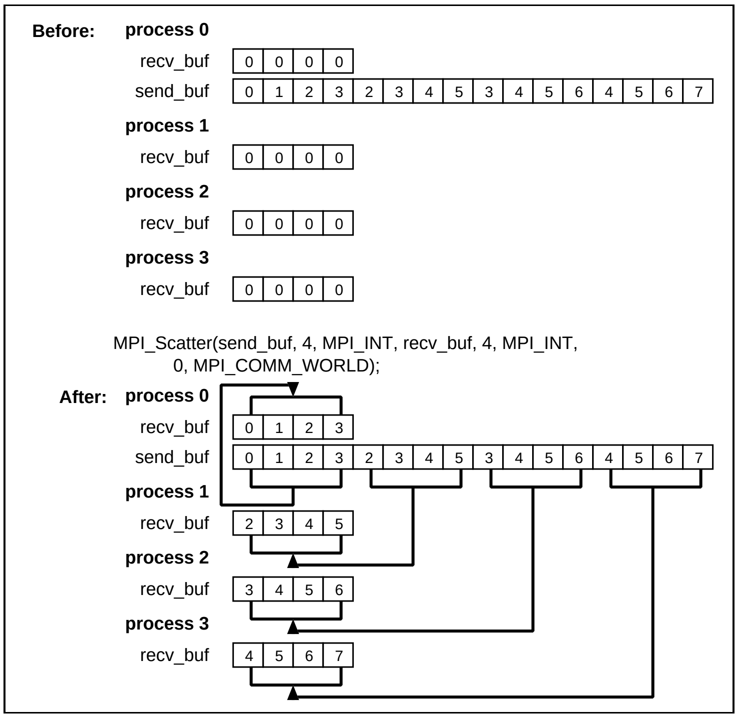

Figure 3.19: MPI_Scatter Illustrated

3.3.2.5 Reduce

The operation described by the operator is applied to each process's operand data and stored in the root process's result buffer (see Figure 3.20). MPI standard operations and user defined operators are explained in Section 3.5.

```
int MPI_Reduce(
  void*         operand  /* buffer to be operated on                */,
  void*         result   /* buffer for the results                  */,
  int           count    /* number of elements in the operand buffer */,
  MPI_Datatype  datatype /* type of the elements in the buffers      */,
  MPI_Op        operator /* operation to be performed                */,
  int           root     /* master process that will have final data */,
  MPI_Comm      comm     /* communicator                             */)
```

Figure 3.20: MPI_Reduce Syntax

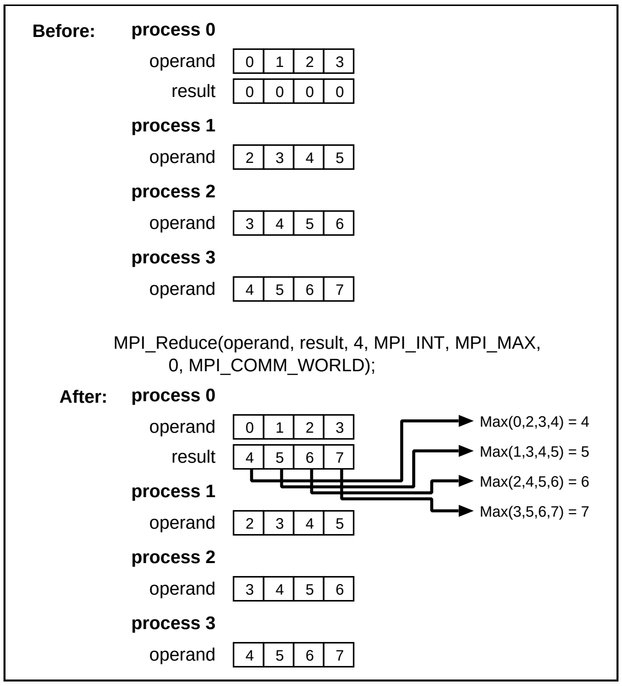

Figure 3.21: MPI_Reduce Illustrated

3.3.2.6 Reduce_Scatter

The operation described by the operator is applied to each process's operand data and scattered to all the processes defined by the communicator (see Figure 3.22). MPI standard operators and user defined operators are explained in Section 3.5.

```
int MPI_Reduce_scatter(
  void*         operand     /* buffer to be operated on            */,
  void*         recv_buf    /* buffer for the results              */,
  int*          recv_counts /* number of elements in the recv_buf  */,
  MPI_Datatype  datatype    /* type of the elements in the buffers */,
  MPI_Op        operator    /* operation to be performed           */,
  MPI_Comm      comm        /* communicator                        */)
```

Figure 3.22: MPI_Reduce_scatter Syntax

3.3.2.7 Scan

Each process stores the result of the operation involving itself and all the other processes that have ranks smaller than itself (see Figure 3.23).

```
int MPI_Scan(
  void*         operand    /* buffer to be operated on             */,
  void*         result     /* buffer for the results               */,
  int           count      /* number of elements in the operand buf */,
  MPI_Datatype  datatype   /* type of the elements in the buffers  */,
  MPI_Op        operator   /* operation to be performed            */,
  MPI_Comm      comm       /* communicator                         */)
```

Figure 3.23: MPI_Scan Syntax

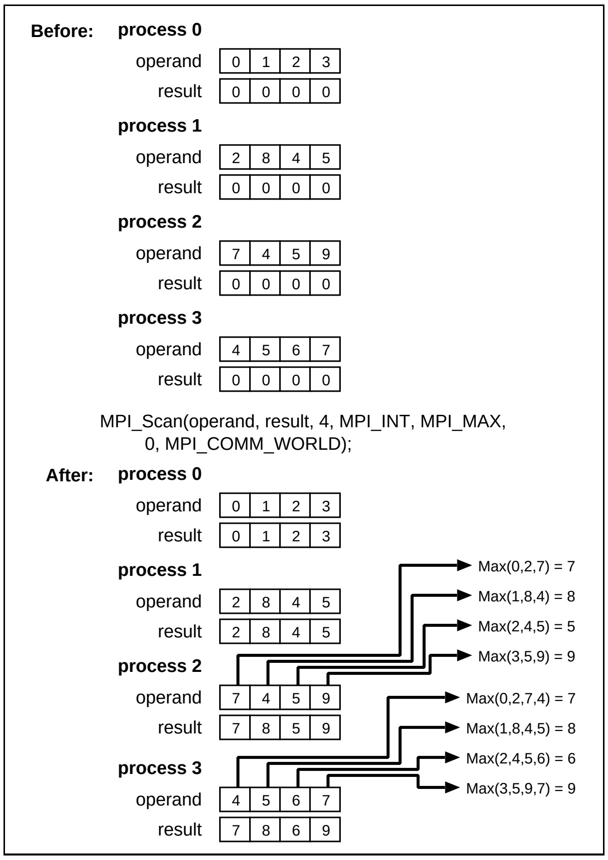

Figure 3.24: MPI_Scan Illustrated

3.3.2.8 Allgather

Each process's send buffer is gathered in the receive buffer of all the processes in the group defined by the communicator `comm` (see Figures 3.25 and 3.26).

```
int MPI_Allgather(
  void*         send_buf   /* buffer to be sent                  */,
  int           send_count /* number of elements in the send_buf   */,
  MPI_Datatype  send_type  /* type of the elements in the send_buf */,
  void*         recv_buf   /* buffer for the received data         */,
  int           recv_count /* number of elements in the recv_buf   */,
  MPI_Datatype  recv_type  /* type of the elements in the recv_buf */,
  MPI_Comm      comm       /* communicator                         */)
```

Figure 3.25: MPI_Allgather Syntax

```
int MPI_Allgatherv(
  void*         send_buf    /* buffer to be sent                      */,
  int           send_count  /* number of elements in the send_buf     */,
  MPI_Datatype  send_type   /* type of the elements in the send_buf   */,
  void*         recv_buf    /* buffer for the received data           */,
  int*          recv_counts /* number of elements in the recv buffers */,
  int*          displs      /* displacements array                    */,
  MPI_Datatype  recv_type   /* type of the elements in the recv_buf   */,
  MPI_Comm      comm        /* communicator                           */)
```

Figure 3.26: MPI_Allgatherv Syntax

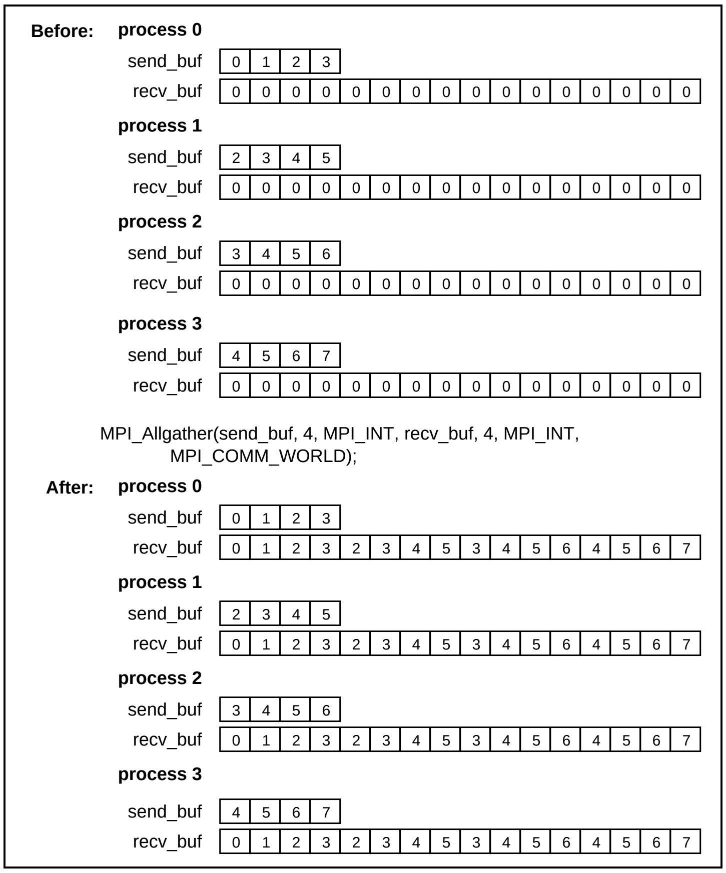

Figure 3.27: MPI_Allgather Illustrated

3.3.2.9 Allreduce

The operation described by the operator is applied to each process's operand data and stored in all of the processes result buffers (see Figure 3.28). MPI standard operations and user defined operators are explained in Section 3.5.

```
int MPI_Allreduce(
  void*         operand    /* buffer to be operated on             */,
  void*         result     /* buffer for the results               */,
  int           count      /* number of elements in the operand buf */,
  MPI_Datatype  datatype   /* type of the elements in the buffers   */,
  MPI_Op        operator   /* operation to be performed is defined  */,
  MPI_Comm      comm       /* communicator                          */)
```

Figure 3.28: MPI_Allreduce Syntax

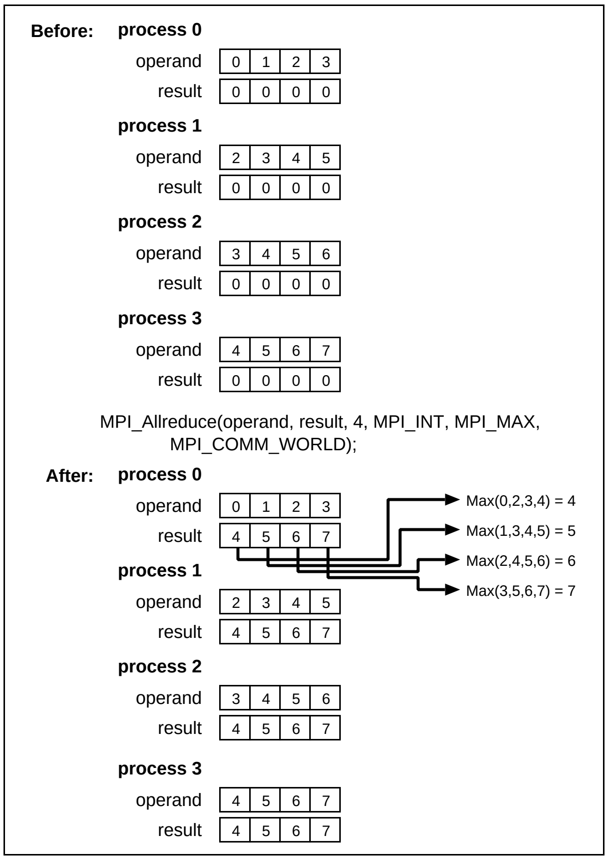

Figure 3.29: MPI_Allreduce Illustrated

3.3.2.10 Alltoall

Each process involved in the communication scatters the contents of its send buffer among all the processes in the system (see Figures 3.30 and 3.31). That is, each process gathers its portion of data from every other process.

```
int MPI_Alltoall(
  void*         send_buf   /* buffer to be sent                    */,
  int           send_count /* number of elements in the send_buf   */,
  MPI_Datatype  send_type  /* type of the elements in the send_buf */,
  void*         recv_buf   /* buffer for the received data         */,
  int           recv_count /* number of elements in the recv_buf   */,
  MPI_Datatype  recv_type  /* type of the elements in the recv_buf */,
  MPI_Comm      comm       /* communicator                         */)
```

Figure 3.30: MPI_Alltoall Syntax

```
int MPI_Alltoallv(
  void*         send_buf    /* buffer to be sent                     */,
  int*          send_counts /* number of elements in the send buffers */,
  int*          send_displs /* displacements array for send buffers   */,
  MPI_Datatype  send_type   /* type of the elements in the send_buf   */,
  void*         recv_buf    /* buffer for the received data          */,
  int*          recv_counts /* number of elements in the recv buffers */,
  int*          recv_displs /* displacements array for recv buffers   */,
  MPI_Datatype  recv_type   /* type of the elements in the recv_buf   */,
  MPI_Comm      comm        /* communicator                          */)
```

Figure 3.31: MPI_Alltoallv Syntax

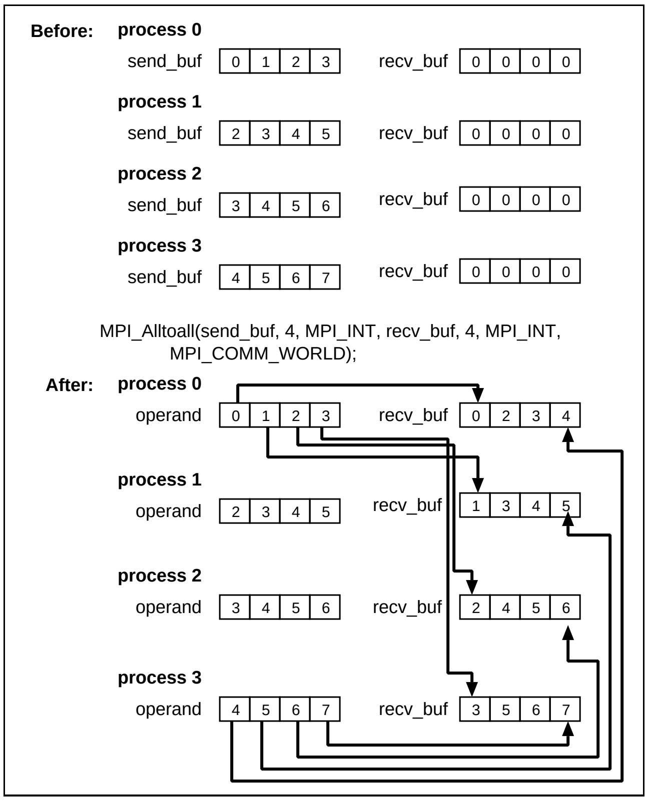

Figure 3.32: MPI_Alltoall Illustrated

3.3.3 Collective Communication Example

Figure 3.33 is an example program that uses collective communication. An array is first scattered among the processes. Then all the processes do some arithmetic operation on their part of the array, in this case they are calculating the square of the integers. After the operation, the data is gathered back to the root process.

```
#include <stdio.h>
#include <mpi.h>
#define COUNT 5
#define ROOT 0

int main (int argc, char **argv)
{
        int nump, rank, i;
        MPI_Status status;
        int *data, *local_buf;

        MPI_Init (&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &nump);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        local_buf = (int *) malloc(COUNT*sizeof(int));
        if (rank == ROOT) {
                data = (int *) malloc(nump*COUNT*sizeof(int));
                for (i=0; i < nump * COUNT; i++) {
                        data[i] = i;
                }
        }
        MPI_Scatter(data, COUNT, MPI_INT, local_buf, COUNT, MPI_INT,
                   ROOT, MPI_COMM_WORLD);
        for (i = 0; i < COUNT; i++) {
                local_buf[i] *= local_buf[i];
        }
        MPI_Gather(local_buf, COUNT, MPI_INT, data, COUNT, MPI_INT,
                    ROOT, MPI_COMM_WORLD);
        free(local_buf);
        if (rank == ROOT) {
                free(data);
        }
        MPI_Finalize();
        return 0;
}
```

Figure 3.33: Collective Communication Example

Datatype Name	C Representation
MPI_CHAR	signed char
MPI_SHORT	signed short
MPI_INT	signed int
MPI_LONG	signed long
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	byte
MPI_PACKED	packed datatype

Table 3.1: Basic Datatypes

3.4 Basic and Derived Datatypes

An important aspect of any message to be transmitted is the type of elements that make up the message. The efficiency of the message passing system relies on how the messages are handled while they are prepared for transmission and after they have been received. Due to transmission latency, it is a good practice for the programmer to try to send all the data she needs to be transmitted in a single call. The system will try to break up the data into appropriate chunks or send the data in a single chunk depending on the network transport. Derived datatypes are used to compose different types of data into a single message, leaving the efficiency concerns to the MPI implementation.

3.4.1 Basic Datatypes

The MPI standard allows for heterogeneous communications as well as communications for homogeneous systems. Therefore, basic datatypes have to be used for communications or to build up more complicated datatypes. These datatypes are listed in Table 3.1; they are appropriately converted to systems local datatypes by the MPI implementation.

3.4.2 Derived Datatypes

MPI programs often need to transmit information that has a more complex format than an array of the basic datatypes. As described in the beginning of this section, instead of sending multiple messages, multiplexing them into one big message and sending in one transmission will be less costly, especially in high-latency networks. This approach can be taken directly by the MPI programmer; however it may require extra memory copying, or extra messages. It is best to leave this decision to the underlying MPI implementation for both portability and efficiency. This also removes the burden of packing and unpacking data from the programmer.

The programmer constructs a derived datatypes at runtime. A datatype must be *committed* (see Figure 3.34) before it can be used in communication. Basic datatypes and previously derived datatypes can both be used in deriving new datatypes. Some datatypes are created as intermediate ones in order to create more complicated datatypes. Such intermediate datatypes do not have to be committed. Derived datatypes can be *freed* (see Figure 3.35) if there is no use for them at a later stage in the program.

```
int MPI_Type_commit(
  MPI_Datatype* datatype /* datatype that is committed */)
```

Figure 3.34: MPI_Type_commit Syntax

```
int MPI_Type_free(
  MPI_Datatype* datatype /* datatype that is freed */)
```

Figure 3.35: MPI_Type_free Syntax

There are four different ways to build new datatypes: contiguous, vector, indexed, and structure.

3.4.2.1 Contiguous Datatypes

The most straightforward derived datatype is the contiguous type (see Figure 3.36). The new datatype is formed by concatenating the old datatype one after another in a contiguous manner.

```
int MPI_Type_contiguous(
  int           count    /* number of elements to replicate */,
  MPI_Datatype  old_type /* old datatype */,
  MPI_Datatype* new_type /* new derived datatype */)
```

Figure 3.36: MPI_Type_contiguous Syntax

3.4.2.2 Vector Datatypes

The vector datatype constructor (see Figure 3.37) is used to replicate blocks of a datatype into locations that are equally spaced. The stride argument is used to indicate the number of elements between the start of each block. This function can be used to easily convert row major C matrixes to column major matrixes using a block of one and a stride of row length. There is an alternative version of this function called `MPI_Type_hvector`, which has the same functionality, but the stride is specified in bytes instead of elements. The stride type is of `MPI_Aint`, which is used to represent memory addresses.

```
int MPI_Type_vector(
  int           count        /* number of elements to replicate      */,
  int           block_length /* length of each block                 */,
  int           stride       /* number of elements between each block */,
  MPI_Datatype  old_type     /* old datatype                         */,
  MPI_Datatype* new_type     /* new derived datatype                 */)
```

Figure 3.37: MPI_Type_vector Syntax

3.4.2.3 Indexed Datatypes

A more complex way to construct a new datatype is to use the indexed routine (see Figure 3.38). This routine works the same way the vector routine does with the addition of variable sized blocks. Also the stride values can be different for each element, therefore the array that keeps these values is referred to as a displacements array. There is an alternative routine for the indexed constructor: `MPI_Type_hindexed`. This routine is used the same way `MPI_Type_indexed` is used. The only difference is that the displacements are in terms of bytes.

```
int MPI_Type_indexed(
  int           count        /* number of element to replicate    */,
  int*          block_lengths /* lengths of blocks                 */,
  int*          displs       /* displacement values of the blocks */,
  MPI_Datatype  old_type     /* old datatype                      */,
  MPI_Datatype* new_type     /* new derived datatype              */)
```

Figure 3.38: MPI_Type_indexed Syntax

3.4.2.4 Defining a New Structure

The most flexible way to construct a new datatype is to define a new MPI structure (see Figure 3.39). This can be seen as an indexed constructor that has the ability to have different types for each block of elements. With this level of flexibility any kind of C structure, except the ones that involve pointers, can be built.

```
int MPI_Type_struct(
  int           count        /* number of elements to replicate   */,
  int*          block_lengths /* lengths of blocks                 */,
  MPI_Aint*     displs       /* displacement values of the blocks */,
  MPI_Datatype* old_types    /* datatypes of each blocks          */,
  MPI_Datatype* new_type     /* new derived datatype              */)
```

Figure 3.39: MPI_Type_struct Syntax

3.4.2.5 Packing and Unpacking

There are a couple of disadvantages to derived datatypes. First, they are fixed. In order to manipulate them, they have to be committed and freed. When the packing and unpacking technique is used, new datatypes do not have to be formed, so the programmer does not bother with committing and freeing fixed datatypes. Another major disadvantage is that the receiver side may not know what to expect beforehand. This problem can be solved with a simple technique: a table of contents can be sent as the first block of data. The rest of the message can be unpacked using this information.

The way that the packing routines are specified forces a specific way to implement them which includes memory copying. This might be seen as a drawback if the block sizes are fairly large.[2]

Three functions are used to pack and unpack data. `MPI_Pack_size()` (see Figure 3.40) is used to figure out the buffer size that is required for a block of data. `MPI_Pack()` (see Figure 3.41) is used to perform the packing and `MPI_Unpack()` (see Figure 3.42) is used for the reverse operation.

The use of communicators in packing and unpacking data may seem useless at first glance. If the system is heterogeneous, the machine byte orders may not match each other. As such, communicators store necessary information to adjust the byte orders.

```
int MPI_Pack_size(
  int          in_count /* number of elements that will be packed */,
  MPI_Datatype datatype /* type of elements                       */,
  MPI_Comm     comm     /* communicator that will use this buffer */,
  int*         size     /* upper bound of the packed message      */)
```

Figure 3.40: MPI_Pack_size Syntax

```
int MPI_Pack(
  void*        in_buffer  /* data to pack                           */,
  int          in_count   /* number of elements to pack             */,
  MPI_Datatype datatype   /* type of elements to pack               */,
  void*        out_buffer /* packed buffer                          */,
  int          out_size   /* size of packed buffer, in bytes        */,
  int*         position   /* current position in buffer, in bytes */,
  MPI_Comm     comm       /* communicator for the message           */)
```

Figure 3.41: MPI_Pack Syntax

[2]It usually is a good practice to send the information in separate messages if the block sizes are large.

```
int MPI_Unpack(
  void*        in_buffer  /* buffer that has all the information  */,
  int          in_count   /* size of packed buffer, in bytes      */,
  int*         position   /* current position in buffer, in bytes */,
  void*        out_buffer /* buffer to unpack into                */,
  int          out_size   /* number of elements to unpack         */,
  MPI_Datatype datatype   /* type of elements to unpack           */,
  MPI_Comm     comm       /* communicator for the message         */)
```

Figure 3.42: MPI_Unpack Syntax

3.4.3 Derived Datatypes Example

In Figure 3.43, derived datatypes are used to send the upper triangle of a matrix [23].

```c
#include<stdio.h>
#include"mpi.h"

#define SIZE 5
int main (int argc, char **argv)
{
        float A[SIZE][SIZE];            /* Complete Matrix */
        float T[SIZE][SIZE];            /* Upper Triangle */
        int i, nump, rank;
        int displacements[SIZE], block_lengths[SIZE];
        MPI_Datatype index_mpi_t;
        MPI_Status status;

        /* Initialize MPI and get the number of processes and rank */
        MPI_Init (&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &nump);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        for (i=0; i < SIZE; i++) {
                block_lengths[i] = SIZE - i;
                displacements[i] = (SIZE + 1) * i;
        }
        MPI_Type_indexed(SIZE, block_lengths, displacements, MPI_FLOAT,
                         &index_mpi_t);
        MPI_Type_commit(&index_mpi_t);

        if (rank == 0)
                MPI_Send(A, 1, index_mpi_t, 1, 0, MPI_COMM_WORLD);
        else /* rank == 1 */
                MPI_Recv(T, 1, index_mpi_t, 0, 0, MPI_COMM_WORLD,
                         &status);

        MPI_Finalize();
        return 0;
}
```

Figure 3.43: Derived Datatypes Example

Operator Name	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Tuple with maximum value and location
MPI_MINLOC	Tuple with maximum value and location

Table 3.2: Predefined Operators

3.5 MPI Standard Reduction Operators and User Defined Operators

The MPI standard provides twelve predefined operators (see Table 3.2) to be used with reduction routines. All of these operators are associative and they may or may not be commutative. They can only be used by the predefined MPI datatypes that are compatible with the operation, i.e., a float can not be used in a bitwise operation.

The last two entries `MPI_MAXLOC` and `MPI_MINLOC` require special tuple datatypes on which to operate. Six basic datatypes are defined in the MPI standard for this purpose: `MPI_FLOAT_INT`, `MPI_DOUBLE_INT`, `MPI_LONG_INT`, `MPI_2INT`, `MPI_SHORT_INT` and `MPI_LONG_DOUBLE_INT`.

In order to create an operation at run time, `MPI_Op_create()` can be used as in Figure 3.44.

```
int MPI_Op_create(
  MPI_User_function* function /* user defined function                   */,
  int                commute  /* true if commutative, false otherwise */,
  MPI_Op*            operator /* handle for the operator                 */)
```

Figure 3.44: MPI_Op_create Syntax

The signature of the user defined function has to conform to the description in Figure 3.45.

```
typedef void MPI_User_Function(
  void*          in_vec    /* first operand vector                   */,
  void*          inout_vec /* second operand vector and the results */,
  int*           length    /* number of elements in the vectors     */,
  MPI_Datatype* datatype  /* type of the elements                   */)
```

Figure 3.45: MPI_User_Function Syntax

An operator can be destroyed using the `MPI_Op_free()` function as in Figure 3.46.

```
int MPI_Op_free(
  MPI_Op* operator /* operator that will be freed */)
```

Figure 3.46: MPI_Op_free Syntax

3.6 Groups and Communicators

A group in MPI is a collection of ordered processes. All of the processes in the group are assigned a unique rank from 0 to $p-1$, where p is the number of processes in the group. With this property, groups define a scope for process names in point to point communication and they also define the scope of the collective operations. Groups have to be associated with communicators in order to be used in communications. A communicator is formed by a group and a context. All the necessary information about the communicator, including the communicator's unique identifier, is stored in the context. The context's internals are implementation specific and MPI programmers do not have to worry about them.

Defining custom communicators can be very handy. As we have seen in the previous sections, all of the communication routines use a communicator. Companies and organizations can take advantage of this property and use communicators for their own private communication space when designing their libraries. Collective communication is another area that extensively takes advantage of communicators. Since all of the processes in a communicator have to participate in the communication, new communicators have to be defined for custom cases.

```
int MPI_Comm_group(
  MPI_Comm     comm     /* communicator that is being inquired    */,
  MPI_Group*   group    /* group of the inquired communicator     */)
```

Figure 3.47: MPI_Comm_group Syntax

```
int MPI_Group_incl(
  MPI_Group    group    /* original group                         */,
  int*         n        /* number of elements in the new group    */,
  int*         ranks    /* ranks in the old group                 */,
  MPI_Group    newgroup /* new group that is created              */)
```

Figure 3.48: MPI_Group_incl Syntax

```
int MPI_Group_free(
  MPI_Group*   group    /* group to be dis-allocated              */)
```

Figure 3.49: MPI_Group_free Syntax

In Figure 3.1, `MPI_COMM_WORLD` is used as the communicator. This communicator's group involves all the processes available at program startup. The first custom communicator has to be derived from this group. `MPI_Comm_group()` (see Figure 3.47) is used to get a handle to a communicator's group. `MPI_Group_incl()` (see Figure 3.48) is used to create a new group from an existing group and an array of ranks. For example if the array is $[2, 4, 5]$, process 2 in the old group becomes the 0th process in the new group, 4 will be the 1st, and 5 will be the 2nd. `MPI_Group_free()` (see Figure 3.49) is used to deallocate groups.

All of the group operations are done locally, no synchronization between processes is required. On the contrary, communicator creation and freeing operations are collective operations and require synchronization. They should be handled with the same care as any other collective operation to prevent deadlocks. `MPI_Comm_create()` (see Figure 3.50) can be used to create a new communicator, the group is derived from communicator `comm` and the ranks are specified by the group `newgroup`.

Another and a more convenient way of creating a new communicator is to use `MPI_Comm_split()` (see Figure 3.51). This routine splits the original communicator into multiple, new communicators. The `color` variable is used to determine the new groups; the processes that specify the same color are assigned to the same group. The `key` variable is used to assign ranks in the new groups. The process with the smallest key value in the new group is assigned the 0th rank, and the other ranks are determined in the same way. If two processes have the same key value, a tie occurs. Ties are broken using the process ranks from the original group. `MPI_UNDEFINED` is a special value that can be used instead of an integer value for color. In this case, the new communicator will be a null communicator (`MPI_COMM_NULL`). `MPI_Comm_split()` can be used as `MPI_Comm_create()` if all the processes that are going to be in the new group call this function with color 0 and all the others call it with `MPI_UNDEFINED`. Communicators can be freed using `MPI_Comm_free()` (see Figure 3.52).

```
int MPI_Comm_create(
  MPI_Comm     comm     /* original communicator                   */,
  MPI_Group    newgroup /* new group for the new communicator      */,
  MPI_Comm     newcomm  /* new communicator that is created        */)
```

Figure 3.50: MPI_Comm_create Syntax

```
int MPI_Comm_split(
  MPI_Comm     comm       /* original communicator             */,
  int          color      /* processes with same color will    */
                          /* belong to the same group          */,
  int          key        /* orders processes in the new group */,
  MPI_Comm*    newcomm    /* new communicator that is created  */)
```

Figure 3.51: MPI_Comm_split Syntax

```
int MPI_Comm_free(
  MPI_Comm*    comm        /* communicator to be destructed    */)
```

Figure 3.52: MPI_Comm_free Syntax

3.6.1 Groups and Communicators Example

In Figure 3.53, two new communicators are created using the existing `MPI_COMM_WORLD`. The new communicator will have the first half of the processes, that is, if there are 5 processes in the system, the new communicator will consist of processes 0 and 1. Two different techniques will be used to achieve this result.

In the first technique, a new group that is going to form the base of the new communicator must be created. `MPI_Comm_group()` is used to extract the group associated with `MPI_COMM_WORLD`. This group and the `new_ranks` array which holds the rank information of the processes are used in the function `MPI_Group_incl()` in order to create the new group. Finally `MPI_Comm_create()` is used to create the new communicator.

In the alternative technique, every process picks a color for itself. Processes 0 and 1, which are going to be included in the new group, select the same color 1. All the other processes select `MPI_UNDEFINED` so that they are not incorporated in a new communicator. `MPI_Comm_split()` is used to create the new communicator.

```
#include <stdio.h>
#include <mpi.h>

int main (int argc, char **argv)
{
        MPI_Group world;
        MPI_Group new_group;
        MPI_Comm  new_comm, split_comm;
        int* new_ranks;
        int i, nump, rank, new_size, color;

        /* Initialize MPI and get the number of processes and rank */
        MPI_Init (&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &nump);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        new_size = nump / 2;
        /* Creating a new communicator with the first method */
        new_ranks = (int*) malloc (new_size*sizeof(int));
        for (i = 0; i < new_size; i++) {
                new_ranks[i] = i;
        }
        MPI_Comm_group(MPI_COMM_WORLD, &world);
        MPI_Group_incl(world, new_size, new_ranks, &new_group);
        MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
        /* Creating a new communicator with the second method */
        if (rank < new_size) {
                color = 1;
        } else {
                color = MPI_UNDEFINED;
        }
        MPI_Comm_split(MPI_COMM_WORLD, color, rank, &split_comm);
        /* Free the created communicators and finalize MPI */
        MPI_Comm_free(&new_comm);
        MPI_Comm_free(&split_comm);
        MPI_Finalize();
        return 0;
}
```

Figure 3.53: Groups and Communicators Example

Chapter 4

Design

Fundamental to the design of USFMPI is the separation of computation and communication into two threads. MPI programmers strive to overlap computation and communication in order to minimize overall execution time. If this goal is achieved by the programmer, an MPI implementation that uses a different thread for communication can greatly benefit the application. This assumes that the operating system has a robust thread implementation. This section describes the multi-threaded design of USFMPI. We specify the responsibilities of the communication and the computation threads as well as their interactions.

4.1 Separation of Computation and Communication

Communication can easily be separated from computation with a multi-threaded approach. The computation thread, which is spawned by the operating system, handles various computation operations and the top-level MPI operations. The top-level MPI operations should not be confused with communication; these operations mostly involve preparing requests for the communication thread and synchronizing with the communication thread if necessary.

As illustrated in Figure 4.1, the two threads communicate using MPI internal requests, signals, and a system pipe. The vertical bars represent interfaces to each component of the system. The small boxes that point to these bars are examples of the messages that can be sent to these interfaces. MPI function calls prepare the requests, signal the communication thread, and take additional actions, e.g., sleep if it is a blocking call. Similarly,

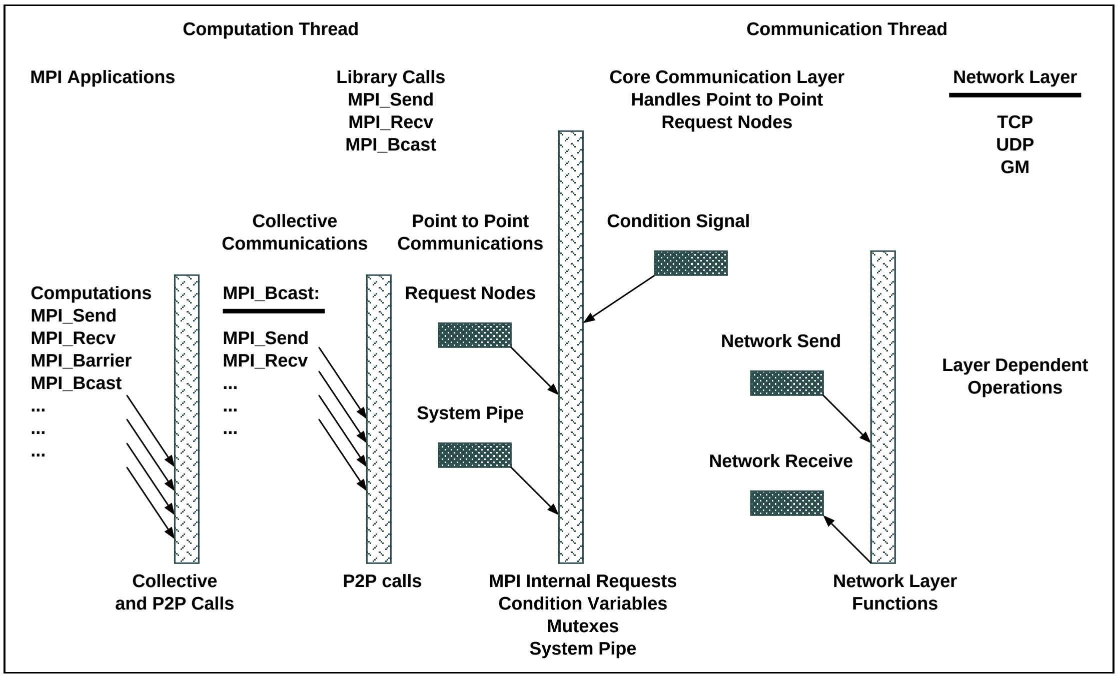

Figure 4.1: USFMPI Thread Design

the communication thread signals the computation thread if necessary, e.g., if a blocking communication completes.

4.2 The Communication Thread

The communication thread employs a multi-layered design in order to support different underlying network protocols and communication mechanisms. The upper layer interacts with the computation thread and the network layer. The lower layer is the network layer and it is usually an implementation of a reliable transport protocol such as TCP. Unreliable protocols like UDP can also be supported by using an intermediate layer that provides reliability.

4.2.1 The Core Communication Layer

The main duty of this layer is to interact with the computation thread in order to handle point to point communication requests. This layer is aware of the network topology, the number of nodes in the system, and the type of system transport protocol. Everything needed for communication is gathered in the core layer.

If the system is not transmitting or receiving a message, the communication thread monitors incoming sockets for messages and the computation thread for requests. The following types of messages and actions are supported by the communication thread.

- **Send Post:** Assume that process A and process B are involved in a point to point communication where process A sends data to process B. Process B is supposed to receive a send post from process A. Receiving this message indicates that process A is ready to send a message. Transmission is started after a receive post is sent by process B to process A. The send post is used by a few different operations. First, it is matched to a receive post. Second, probes use send posts to figure out which posted messages can be received. Finally, if any wild-cards are used by the receive call, such as `MPI_ANY_SOURCE` , then the receive post is sent to the process with the first matching send post.

- **Receive Post:** As with the send post, assume that process A is sending a message to process B. Process A receives this post from process B. Receiving this post indicates that process B is ready to receive a message from the process A. If process A has a matching message to send, transmission is started right away. Otherwise, the receive post is stored as a request from another process.

- **First Chunk:** This message is an indicator of the start of a long message. The system matches it to the first matching request and sends an acknowledgment.

- **Middle Chunk:** This message is an indicator of an ongoing long message transmission. Since there is only one ongoing long communication between two peers at a time, the buffer is easily found and the new chunk is added to the rest of the message. The sending process answers with an acknowledgment.

- **Last Chunk:** This message is an indicator of the end of an ongoing long message transmission. The buffer is found and the last chunk is added to the rest of the buffer. The request is marked as complete and necessary action is taken e.g., if it was a blocking request, the computation thread is signaled. An acknowledgment is not required.

- **Acknowledgment:** This message can only be received by the sender side of a long ongoing transmission. The buffer to be sent is found and the next appropriate chunk of the message is sent to the receiver.
- **Short Message:** When this message is received, one of the receive requests is fulfilled, the matching request is found, marked as complete and the necessary action is taken.

Two modes of transmission are supported depending on the size of the messages to be transported :

- **Short Messages:** The short message size is system dependent. In the current implementation of MPI a short message is less than or equal to 64 kilobytes. Short messages can be transferred in a single transmission.
- **Long Messages:** Any message that is not a short message is considered a long message. These messages are broken down into chunks, and transferred one by one. This mode uses a rendezvous protocol, the following chunk is only sent after an acknowledgment is received. This implies there can be only one ongoing communication at the same time between two peers.

4.2.2 Interaction with the Computation Thread

The communication thread is driven by requests from the computation thread. These requests serve the MPI functions that are issued by the MPI programmer. As mentioned before, in the core communication layer the system is only aware of sends and receives. The requests are only for sending, receiving, and finalizing.

- **Send Request:** If this request is received, a message is ready to be sent. If this process received a receive post that matches this request, the transmission is started right away. If not, a send post is sent to the receiving process.
- **Receive Request:** If this request is received, a message is ready to be received. If the `MPI_ANY_SOURCE` wild-card is not used, the source to receive from is known and a receive post is sent to that process. Otherwise, the process waits until there is a matching send post (See the actions taken when a send post is received.).

- **Finalize Request:** The communication thread is terminated and the termination sequence is started. The termination is handled by the computation thread. Initialization and termination are the only times that the computation thread sends and receives messages directly. This does not pose a deadlock threat to the shared resource, which is the communication media, due to the communication thread not being active at these times. The communication thread is started at the end of initialization and terminated at the beginning of finalization.

4.3 Computation Thread

The computation thread is the main thread, in other words this thread is the one that is forked by the system when the `mpirun` command is issued by the user. Depending on the nature of the application, this thread may spend most of its time on computation. The chosen name reflects this property.

The only operations that manipulate the internal MPI structures are done in MPI library calls in this thread. Interrupting the work in order to check for messages that come from other processes or perform other communication related operations is not necessary due to the existence of the communication thread. Single threaded systems, like MPICH, must interrupt the computation explicitly to poll for messages or to transmit a message.

Initialization and finalization are the only two operations that are not handled by the communication thread. The communication thread is created during initialization and terminated before finalization, leaving the communication to the computation thread for these two operations. Some MPI implementations run daemons that avoid the need for initialization and finalization operations or make their use very minimal. The connections are established during the boot-up of these daemons and they are closed when the daemons halt. All the other communication operations can be categorized as point to point or collective. Finally, some environment operations are handled by the computation thread, such as `MPI_Comm_rank` and `MPI_Comm_size`.

4.3.1 Initialization

The initialization of the system depends entirely on the chosen network layer. Since USFMPI is capable of supporting multiple network layers, a different initialization is needed for every supported network.

TCP is the default network layer, therefore we use it to explain the design. First, connections between the processes must be established. Since the system is TCP there should be a connection between every process. This design changes dramatically for different network layers. For example, if the layer is UDP, a single incoming socket per process would be enough. The processes with higher ranks attempt to connect to the processes with lower ranks in order to avoid deadlock. At the end, each process has a list of sockets that it uses to communicate with the other processes.

4.3.2 Finalization Operation

At a first glance, the finalization operation may seem unnecessary. However, it is needed as a final barrier before the program exits in most network layers. Different network layers will have different reasons for this final barrier, and some network layers, due to their internal structure, may not need a final barrier at all.

TCP is a layer that needs the final barrier operation. Finalization is handled as follows. Process 0 works as the master in this operation. Each process sends a message to process 0 when `MPI_Finalize()` method is called at that process. Therefore, process 0 should be capable of handling these messages in the communication thread during the execution of the program. When process 0 reaches the finalization state, it waits for finalization messages from the processes that did not already send a message indicating it is ready to finalize. When every process successfully sends their respective messages, process 0 sends back a message to every process indicating that it is time to terminate.

The reason that the TCP system does not work without this barrier is as follows. When a socket is closed, a message is sent over the socket indicating that the connection is closed. This message is not generated by USFMPI, therefore it is not handled by the communication thread. In other words, the communication thread treats a closed socket as a network or a processor failure. Having a system as described above guarantees that this can only happen if one of the processes dies unexpectedly or if one of the processors gets disconnected from the network.

4.3.3 Point to Point Operations

The communication part of point to point operations is handled by the communication thread as explained in Section 4.2.1. The MPI routines that are called for point to point operations, such as `MPI_Send()` and `MPI_Recv()`, setup an internal request for the communication thread. This internal structure and the way each routine sets it up are explained in Chapter 5.

In Chapter 3, point to point communications are categorized into two major subsets, blocking and nonblocking. This technique of categorization is very suitable for explaining how MPI is used for distributed programming. However, from a design perspective blocking and nonblocking communications are very much alike. Both of them use the same communication thread in order to transmit and receive messages. Their first difference is in the MPI calls, which are done in the computation thread. The blocking calls sleep on a signal after they setup the communication request and notify the communication thread. The other difference is in the communication thread. When the communication successfully finishes, the computation thread is signaled if the operation is blocking. Note that if `MPI_Wait()` is called for a nonblocking communication, and if the communication is not completed yet, it is treated as a blocking communication from that point.

4.3.4 Collective Operations

The communication operations that require the involvement of all the processes in a group of processes are considered collective. The collective communication operations are built using the point to point communication routines. Therefore, the multi-threaded structure and the other MPI internals are not used directly by these operations. In other words, the collective communication part of the library is almost like a separate MPI application.

The robustness and efficiency of collective operations depend mostly on the network topology. Butterfly algorithms [23] are chosen to implement these operations due to their logarithmic behavior. The nature of the algorithms and how some odd cases are handled are explained in the following paragraphs:

- **Barrier:** The barrier operation is satisfied when all the processes in a group execute this operation. From another perspective, every process has to know that every one of them issued a barrier call. In order to achieve this, USFMPI carries out a gather operation with the master process being 0; this master process decision does not affect the result of the operation. When the gather operation ends, process 0 knows that every process issued a barrier call. At the second step a broadcast operation is issued to every process with the master process 0. If these two steps are executed successfully, then all the processes have issued a barrier call.

- **Broadcast:** The broadcast operation executes in $\lceil \log_2 n \rceil$ steps where n represents the number of processes. In each step the processes have a partner that is indicated by that step. Each process waits until it receives the message, and starts to send it to its partners after the message is received.

- **Scatter:** The scatter operation's communication scheme is very similar to broadcast's communication scheme. The only difference is that only portions of the message is transferred to the partners. Only the parts that the receiver and his future partners need are sent.

- **Gather:** The gather operation works exactly like the scatter operation with the order of transmission reversed and the sending and receiving roles exchanged.

- **Reduce:** The reduce operation uses the same communication scheme as the gather operation. The chosen reduce operator is applied to the receiver side's original data and the received data.

- **Scan:** The scan operation does not operate in logarithmic time due to its specification, please refer to Section 3.3.2.7 for details. The scan operation executes linearly, every process with rank p waits for a message from process $(p-1)$, executes the operation as in reduce, and passes the result to process $(p+1)$.

- **Allgather:** The allgather operation is optimized for operations that involve a power of two processes. If this is the case, the allgather operation executes in $\log_2 n$ steps. The odd cases add a constant factor to the number of steps depending on the situation. In each of the steps the processes are paired with another one depending on their rank; the paired processes exchange their information. The size and the location of the data are dynamic in this operation; that is, at the later steps there will be more data to exchange. In the odd cases, some processes may not have their partners present in the system. In these cases, the system choses an appropriate process with the desired data to replace the missing process.

- **Allreduce:** The allreduce operation works exactly the same way as allgather. It is a little more straightforward due to the data size and location being constant throughout the operation.

- **Alltoall:** The alltoall operation uses the scatter operation to fulfill its requirement. Both the gather and scatter operations can be used to accomplish this task. All the processes scatter their data to other processes that are involved in the communication.

Chapter 5

Implementation

This chapter explains the implementation of the internal structures and operating system specific code used in USFMPI.

5.1 Running MPI with Mpirun

`Mpirun` is the daemon that is responsible for starting up the individual MPI processes and checking if they exited with correct return values. MPI users can configure `mpirun` using standard configuration files such as the machines file[1] or by command line parameters, e.g., `mpirun -np 2` would start up the program with two processes.

`Mpirun` starts up the system by creating a new process on each node specified by the machines file. This start up operation is achieved by using either `ssh` or `rsh` depending on the cluster security settings. The process's rank, the number of processes in the system, and the machines file are passed as command line arguments to the newly created process by the `Mpirun` daemon. After all the processes start, the daemon waits until all of these processes terminate. If any one of them terminate with a nonzero value, the error is reported to the user.

[1]The machines file is implementation specific. For the TCP implementation, it includes the IP addresses and the number of processes that should be started per address. It is configured externally to mpirun, e.g., by a batch system such as PBS.

5.2 Data Structures

This section explains the various data structures that are used in the implementation of USFMPI.

5.2.1 MPI Internal Request

`MPI_Request_Int` is the main data structure of the library. Every communication event is handled through an internal MPI request. Consequently, it is a fairly large data structure. The size of this data structure is directly proportional to the amount and types of functionality supported by the implementation.

A doubly-linked list is used to queue communication requests. The elements of this list are called request nodes. They have a back pointer, a next pointer, and a pointer to the `MPI_Request_Int`. The global variables `request_hd` and `request_tl` are the head and tail of this list. The MPI internal request structure is explained in Table 5.1.

Field Name	Meaning
active	This field exists in order to achieve persistent communication functionality. Persistent communications are inactive when they are created, `MPI_Start()` is used to activate them. Possible values: 0 - inactive, 1 - active non-persistent, 2 - active persistent. Inactive requests are always persistent. In order to save space and code, another field named persistent is not used.
done	Indicates that the requested operation (data transfer) is complete. `MPI_Wait()` and `MPI_Test()` and their variants use this field to find out the state of a nonblocking message transfer. In addition, the functions in the `comm_thread` module that serve the internal requests ignore requests with the done field equal to 1. Possible values: 0 - not done, 1 - done.
type	Type of communication: 0 - receive and 1 - send.
blocking	Possible values: 0 - nonblocking, 1 - blocking.
posted	Both types of communication (send and receive) first send a post when they are processed by the `serve_mpi_requests` function if they do not have a matching send or receive post in their system request (`sysreq`) list. The system request list, which is explained in Section 5.2.2, is used to store send and receive posts that are not matched to an internal MPI request. The function `serve_mpi_requests` ignores the already posted messages. Possible values: 0 - not posted, 1 - posted.
buf	A pointer to the message buffer.
size	The size of the buffer.
tag	The tag of the message. Possible values: The range 0..32767 is used by the MPI programmer, negative values can be used by the implementation, e.g., collective communication.
peer	The other peer of the communication, e.g., the receiver side in a send operation.
comm	The communicator of the message. `MPI_COMM_WORLD` is the most common value.
multiple	Depending on the size of the message, a number of chunks may be used to transmit the message. This is platform and network dependent. This field determines if the message can be transmitted in one chunk or not. Possible values: 0 - one chunk, 1 - multiple chunks.
nchunk	This field records the number of the chunks (packets) that have been sent or received.
node	The back pointer to the node that holds this information in the doubly-linked list, used for efficient removal.

Table 5.1: MPI Internal Request Structure

5.2.2 System Requests

The send and receive posts that do not match a corresponding MPI request have to be stored for future reference. These posts are stored in the Systems Requests hash table. The system request structure is explained in Table 5.2.

Field Name	Meaning
next	Required for the linked list; it points to the next node.
tag	The tag of the message. Possible values: 0..32767 is used by the MPI programmer, negative values can be used by the implementation. e.g., collective communication.
source	The other peer of the communication, e.g., the receiver side in a send operation.
type	Type of communication: 0 - receive and 1 - send.
comm	The communicator of the message. (`MPI_COMM_WORLD` is used most commonly.)
size	The size of the buffer.

Table 5.2: System Request Structure

5.2.3 Connection List

A list of connections, where each connection has a type and a platform specific descriptor, e.g., TCP or UDP sockets. Currently, only TCP sockets are used for communication, GM for Myrinet will be added in the future. The connection list structure is explained in Table 5.3.

Field Name	Meaning
sd	The socket descriptor used for communication.
hostname	The hostname of the peer, used by TCP sockets.
port	The port number of the peer, used by TCP sockets.
path	The path of the peer, may be used by GM.
type	Which type of connection it is, TCP or GM.

Table 5.3: Connection List Structure

5.2.4 Rank List

Every process has a rank in every communicator, and every communicator needs to know how many processes there are in that communicator. The rank list structure is explained in Table 5.4.

Field Name	**Meaning**
rank	The rank of the process.
comm	The value of the communicator.
nump	The number of processes in the communicator.

Table 5.4: Rank List Structure

5.2.5 Finalize List

All the processes have to coordinate before terminating. Otherwise, the closed TCP sockets can cause errors. The closed sockets send an event indicating that they are closed. This event is not handled by the communication thread. This list is managed by the 0th process and helps it make sure that all the processes are ready to terminate; this also serves as a hard-coded barrier for finalization of the application.

5.2.6 Synchronization List

Only one multiple-chunked send transmission can exist between two processes at a time, or the acknowledgments will be ambiguous. The synchronization list makes sure this property is maintained. An internal request pointer is needed for each connection in order to keep track of ongoing transmissions. These individual pointers form the synchronization list. If there is an ongoing communication with one of the other processes, the pointer will point to that request so that it can be updated quickly and to prevent another communication. If there is no communication, the pointer is NULL.

5.2.7 System Pipe and Mutexes

The Pthreads interface is used to create a communication thread and to synchronize the computation thread with the communication thread. Both mutex variables and a UNIX pipe are used for synchronization. The system pipe is used by the computation thread to signal the communication thread when a new internal MPI request arises. If there are no ongoing communication events, the communication thread sleeps on a `select()` call that monitors the system pipe and the network file descriptors.[2] The system pipe is also used by `MPI_Finalize()` for signaling the termination of the MPI program. Mutexes are used to protect the system requests hash table and the MPI internal requests list. A Pthreads condition variable is used for waking up the computation thread that waits for blocking communication to finish. The condition variable is needed to signal between the threads so that the main program can stop execution (e.g., for blocking receives or `MPI_Wait()`) and be signaled when the communication is done.

5.3 Point To Point Communication

The communication part of the system is handled by a communication thread and the execution of the user code is handled by the computation thread. In the communication thread, mutexes are used to protect the shared data variables and a system pipe is used to receive data from the computation thread. The mutexes and pipes are created in the `MPI_Init()` function. The main thread uses the `MPI_Request_Int` mechanism to define the communication requests and uses the pipe to notify the communication thread. It can go to sleep or continue execution depending on the type of the request.

The `select()` system call used in the communication thread monitors TCP sockets and the system pipe for events. If any data are in the system pipe, there is a new transmission request or an execution termination request. If it is a new transmission request, the `serve_mpi_requests()` function is called. This function traverses the request list and serves all the requests that are not complete and not posted. A send or receive post is sent if there is not a matching post in the system request hash table, otherwise it starts the transmission.

If there are data in one of the TCP sockets, a handle message function is called to process the incoming message. The message header is analyzed to find out the type of the incoming message. The message header format is described in Table 5.5.

If the incoming message is a post, the `MPI_Request_Int` list is searched for an incomplete, matching request. If one is found the transmission is started, otherwise the post is stored in the system request table. If `MPI_ANY_SOURCE` is used instead of specifying

[2]The network file descriptors are implementation specific. The current implementation sleeps on a number of TCP sockets.

a real peer, a receive post can not be sent. The sending of the receive post is delayed until there is a matching[3] send post from any other peer. This check is done in handling the send post function, therefore the delayed receive post is sent by this function.

Maintaining only one ongoing multiple-chunk send operation is handled on the sender side; `synchr_list` (Synchronization List) is used for this purpose as explained in Section 5.2.6. The sending side will receive acknowledgments throughout the transmission and will send the next chunk as soon as the acknowledgment is received. The size of the chunk is adjusted so that the sender does not overflow the receiver's buffer. If the size of the message is smaller than one chunk, the message is sent as a short message. The receiver side has to handle three types of messages in an ongoing communication; if the incoming packet is the first chunk, it matches the packet to the first matching request in the user buffer and copies the message buffer. If it is a middle chunk, the message is copied to the proper place in the user buffer. With the arrival of the terminal chunk, the message is copied and the request is marked as done and appropriate action is taken, e.g., signaling the computation thread if it is a blocking receive. Similarly, when the sender side sends the terminal chunk, it signals the computation thread if necessary. The concept of having only one ongoing multiple-chunk send makes it necessary for the communication thread to check for the receive posts that may have been received during an ongoing communication. The `serve_sysreq` function is called after sending the last chunk of a communication. This function searches the system request table for a matching `recv_post` and starts a new transmission if it finds one.

The communication termination message can be received only by process 0, as it is the coordinator for program termination. Process termination information is saved into the finalize list and used when `MPI_Finalize()` is called.

[3]The communicator and tag must match.

Field Name	Meaning
0) Source Rank	The sender of the message.
1) Destination Rank	The destination of the message.
2) Tag	The tag of the message.
3) Communicator	The communicator that is used for the message.
4) Size	The size of the message.
5) Type	0) Recv post
	1) Send post
	2) First chunk of synchronous communication
	3) Middle chunk, chunk of synchronous communication
	4) Terminal chunk of synchronous communication
	5) Acknowledgment of chunk
	6) Short message
	7) Communication termination message

* All of the variables are integers.

Table 5.5: Message Header Format

5.4 Collective Communication

The collective communication routines are implemented on top of point to point communication. Messages that belong to collective communication do not interfere with the rest of the communications due to the usage of tags. Each collective routine has a special tag with a negative value that it uses for its messages. MPI programmers are only allowed to use non-negative valued tags in their source code. Negative valued tags are reserved for the implementation.

Functionality that is provided by the collective routines is heavily used in typical distributed programs. Users can also achieve this functionality using the point to point communication functions. However, this will not only introduce extra work for the users, but it will also affect the portability of programs. Selection of the algorithms to implement collective routines should be made by taking the underlying network topology into account. When MPI functions are used, the users can safely assume that the best algorithm for the specific cluster is used.[4] MPI implementations can also take advantage of the network layer functions that provide functionality such as hardware broadcast or barrier.

Collective routines need a communicator in order to operate; all of the processes

[4]The selected MPI implementation must be optimized for the underlying network topology.

in the communicator have to participate in the communication. Communicators other than MPI_COMM_WORLD are usually used to achieve the collective functionality for a variable set of processes. If there is a master process needed for that particular routine, any of the processes can be selected by the MPI programmer to behave as the master.

The algorithms that are used for collective communication are explained in Section 4.3.4. This section on implementation explains how the bitwise operations are handled to get these algorithms to work. Also, the odd cases are explained in more detail.

Most collective operations require several steps depending on the number of processes involved. In most of the operations there are $\lceil \log_2 n \rceil$ steps where n is the number of processes participating in the operation. The logarithmic behavior is due to the nature of the butterfly algorithms [23]. A bitmask is used to figure out the partner process of each process in each step. This bitmask's length is equal to $\lceil \log_2 n \rceil$, e.g., if n is 16 the bitmask will be 0000. In each step the bitmask is XOR'ed with the process rank in order to find its partner. In the first step the bitmask is equal to 0001, so that process 0 (0000)'s partner is process 1 (0001) and vice-versa. In the next step the bitmask is shifted left so that it becomes 0010.

Odd cases may arise in allgather and allreduce algorithms where each process must have a partner for each step. The partners will be missing due to the number of processes in the system. If it is not equal to a power of two, some process will miss a partner in some steps. In all the other operations, if the partner is not present in the system, it can be ignored. In these odd cases, the partner is replaced by a process that is present in the system. This process is chosen by looking at the partner of the missing process in the previous step. If that is also missing, the search process continues with the previous steps. As an example, lets consider an Allgather operation with 5 processes. The operation will be completed in three steps. In the first step, process 0 partners with process 1, so does process 2 and 3. Process 4 does not partner with any other process. In the second step, process 0 and 2, and process 1 and 3 is selected as partners. Process 4 does not partner with any other process as in the previous step. At this point the first four processes all have each other's data. In the third step process 0 exchanges its data with process 4. Process 1, 2, and 3 do not have partners, which would have been processes 5, 6, and 7. Since these processes are not present in the execution, process 4 sends the necessary data to these processes. This example is illustrated in Figure 5.1.

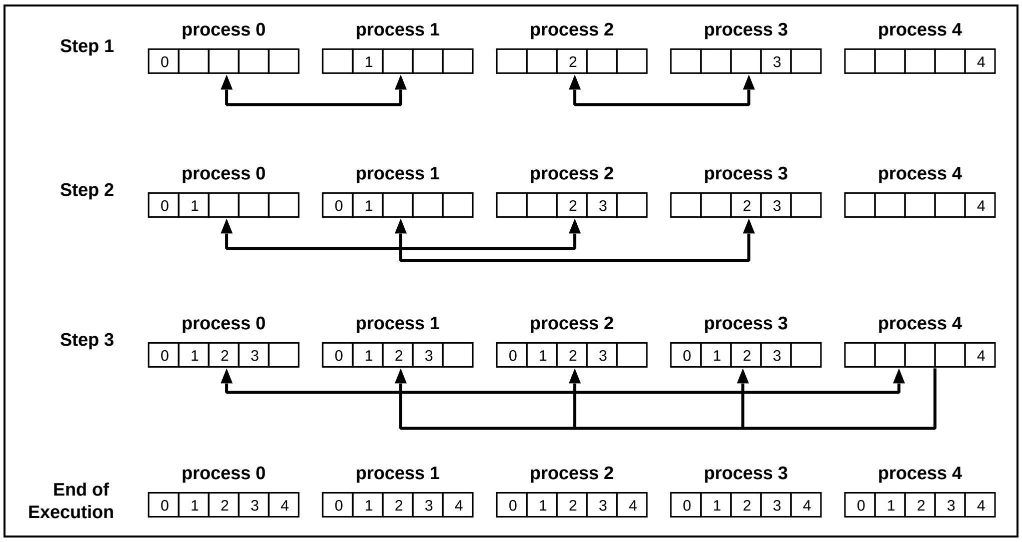

Figure 5.1: Odd Case Allgather Example

Chapter 6

Experimental Results

In this chapter we present experimental results for three implementations of MPI: USFMPI, LAM-MPI [22] version 6.5.8 and MPICH [3] version 1.2.5. The Pallas [11] benchmarks version 2.2 are used to test these systems. The algorithms used for each benchmark are explained in the following sections, please refer to the Pallas manual for more detailed descriptions. The results are only intended to compare the performance of USFMPI versus these two publicly available implementations. The reasons for performance differences are not explained for every case, because it might require extensive analysis of the other two implementations.

Each test is run with 16 dual processor SMP machines (32 processors). Fastethernet is used for the communications between the SMP machines. The local network card is used for communications between the processors on the same machine. The data sizes used for each benchmark start from 0 bytes and are doubled until the size is 4194304 bytes. Each step[1] is run multiple times and an average value is reported. Data sizes from 0 to 32768 are run for 1000 times, size 65536 is run 640 times and this number is halved in each following step.

The benchmarks are also run with a varying number of processes. Pallas runs the tests with powers of 2 processes getting involved for all the tests except PingPong and PingPing, because they can only be run with 2 processes.

USFMPI performs worse than both MPICH and LAM-MPI for the buffer sizes that are less than 1024 bytes, because USFMPI does not optimize short message transfer with extra buffering. This property holds for all of the Pallas tests, so it is not explicitly

[1] A step is an experiment that is run with the current data size.

mentioned in every section. The part of the graph that is discussed is message sizes that are bigger than 1024 bytes. Also, due to the logarithmic property of the graphs, differences in the smaller values are magnified and differences in bigger values are shrunk.

6.1 PingPong

The PingPong benchmark can only be tested with two processes. Process 0 first issues an `MPI_Send()` call, and then an `MPI_Recv()` call, each with the current data size. Process 1 issues the same calls in the opposite order. Figure 6.1 shows the results for the three implementations when the processes are selected from two different nodes. MPICH's and USFMPI's performances are very similar; they both outperform LAM-MPI. Figure 6.2 shows the results for the three implementations when the processes are selected from the same node. This test shows that USFMPI uses Linux more effectively than MPICH and LAM-MPI for processes that run on the same node.

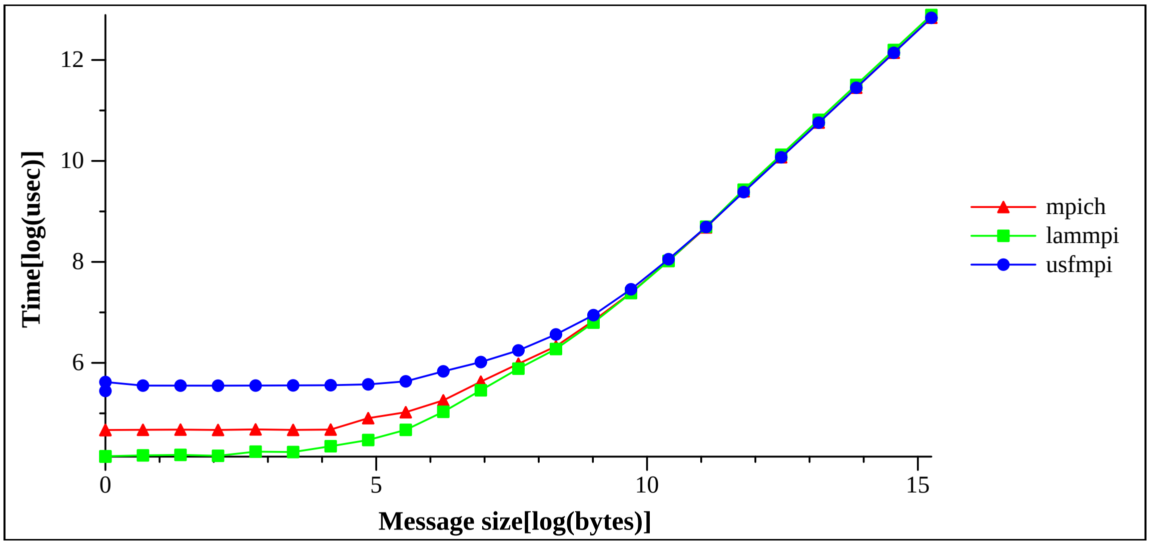

Figure 6.1: PingPong with 2 Processes on Two Nodes.

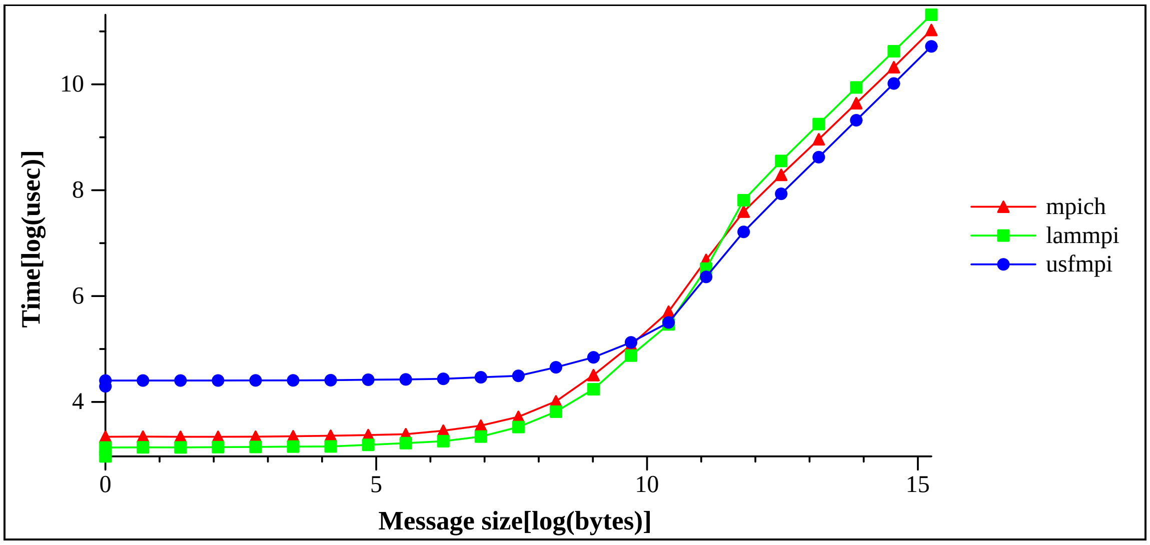

Figure 6.2: PingPong with 2 Processes on One Node.

6.2 PingPing

The PingPing benchmark is similar to the PingPong benchmark. In this case, both of the processes call the same functions in the same order. First an `MPI_Isend()` call, then an `MPI_Recv()` call, finally an `MPI_Wait()` call on the nonblocking send call. Figure 6.3 shows the results for the three implementations when the processes are selected from two different nodes. MPICH is the worst performer in this benchmark. USFMPI is the best performer, and LAM-MPI is the middle performer. Based on the results from the PingPong test we can conclude that both USFMPI and MPICH can exploit the full capacity of the network. However, when a nonblocking call is involved, MPICH has a noticeable drawback due to its single-threaded implementation. Figure 6.4 shows the results for the three implementations when the processes are selected from the same node. As in Figure 6.2, this test also shows that USFMPI uses Linux more effectively than MPICH and LAM-MPI for processes that run on the same node.

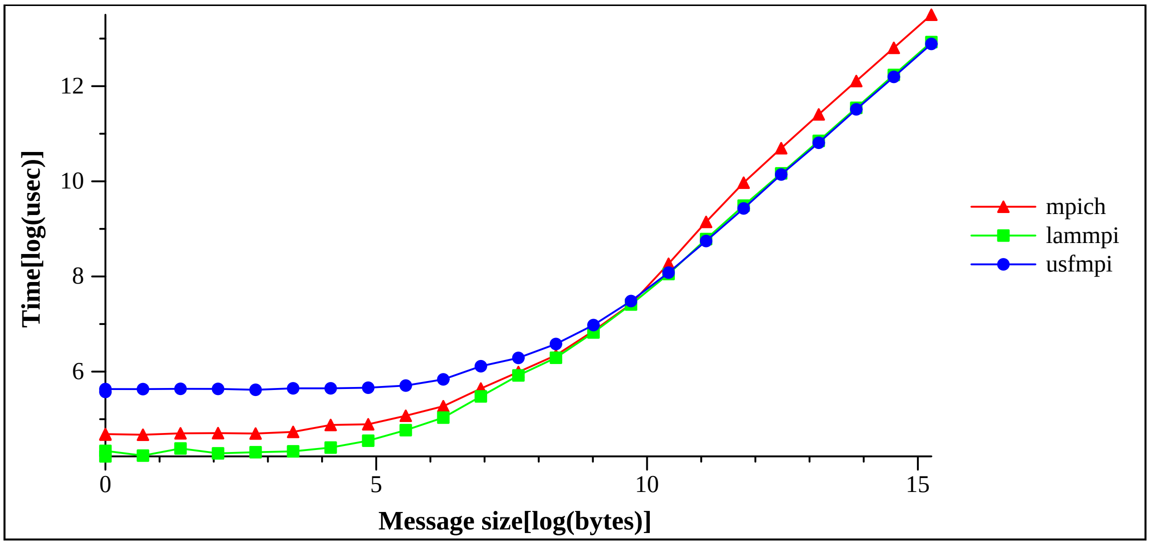

Figure 6.3: PingPing with 2 Processes on Two Nodes.

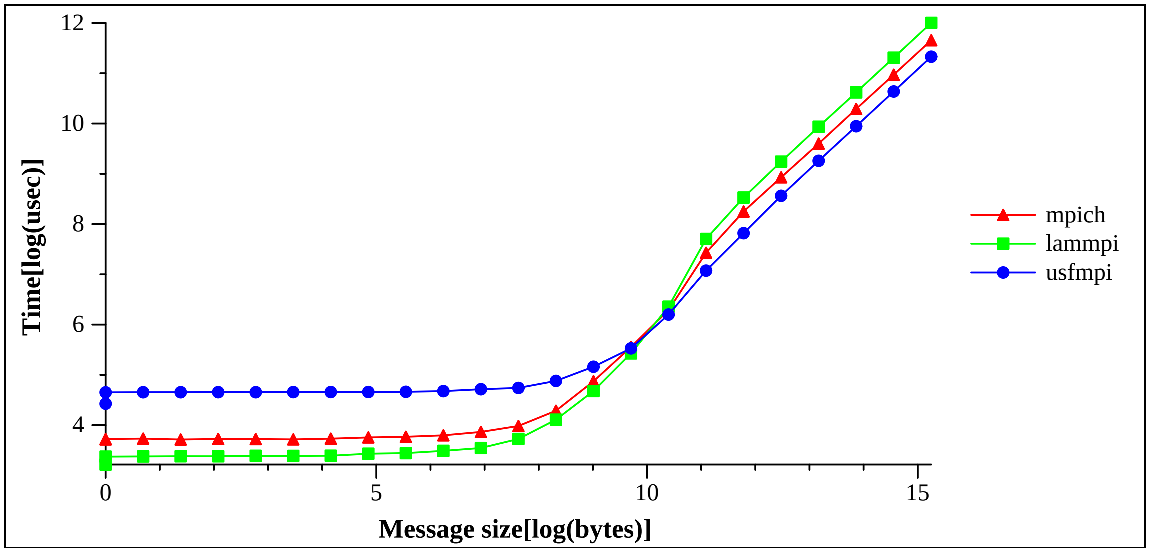

Figure 6.4: PingPing with 2 Processes on One Node.

6.3 Allgather

The allgather benchmark uses the `MPI_Allgather()` operation. Each process inputs the current data size n and gathers $n\times$ `<number of processes>`. Figures 6.5, 6.6, 6.7, 6.8 show the results of this benchmark. Due to a complication in MPICH, the results for 32 processes are not presented for this experiment. In all of the four cases, USFMPI produces the best results, MPICH is the second best implementation. As the number of processes in the system grows, MPICH starts to perform closer to USFMPI, while LAM-MPI is always the same regardless of the number of processes.

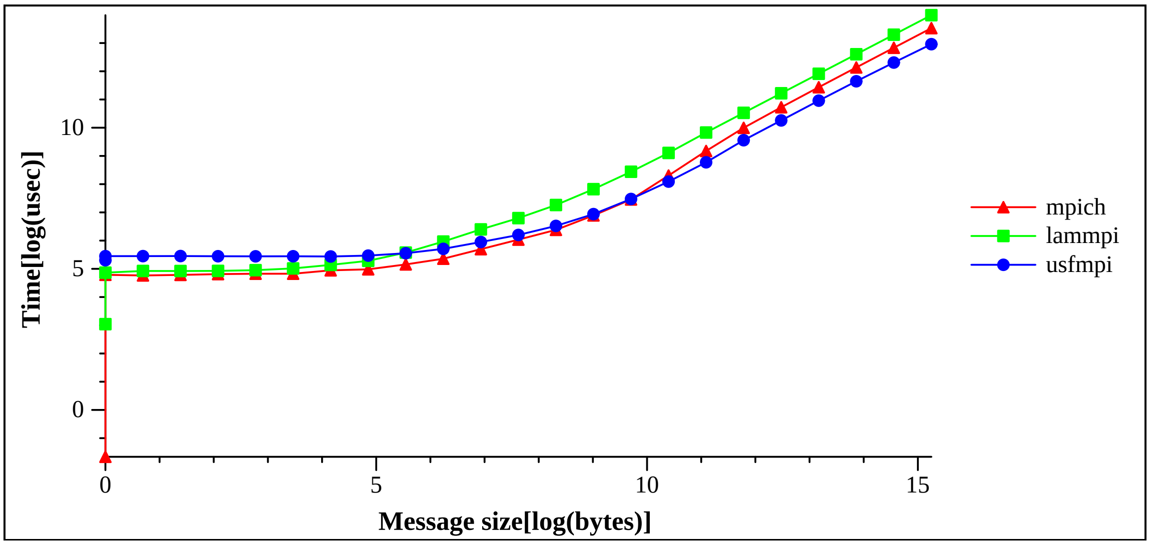

Figure 6.5: Allgather with 2 Processes.

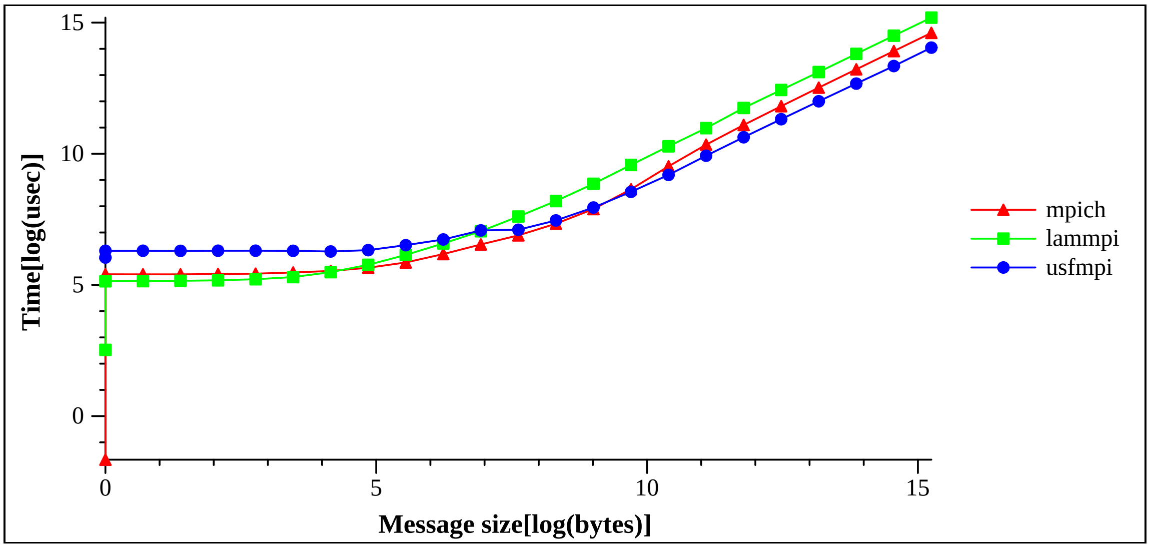

Figure 6.6: Allgather with 4 Processes.

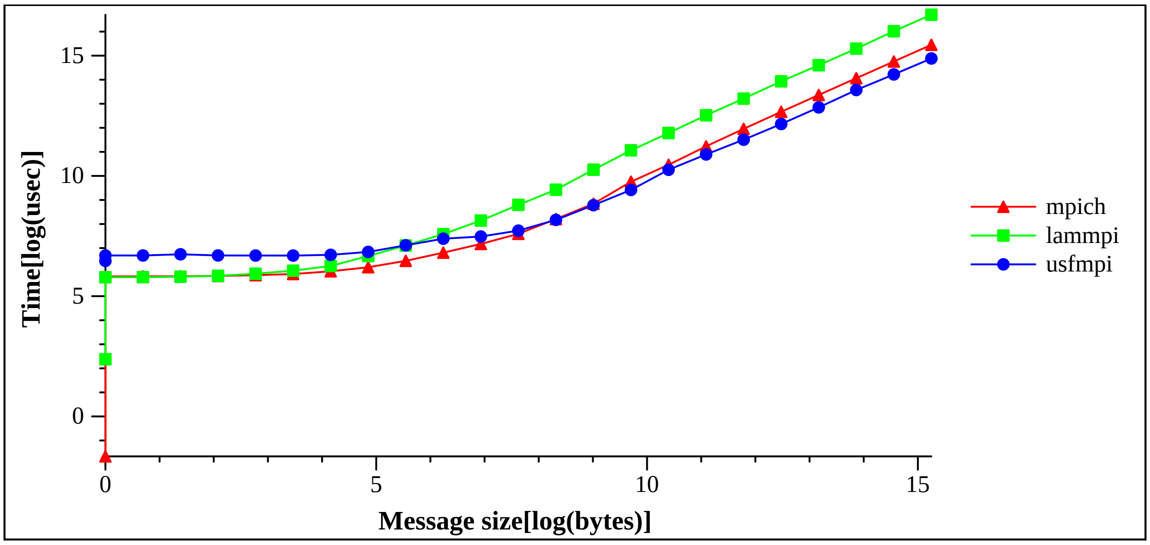

Figure 6.7: Allgather with 8 Processes.

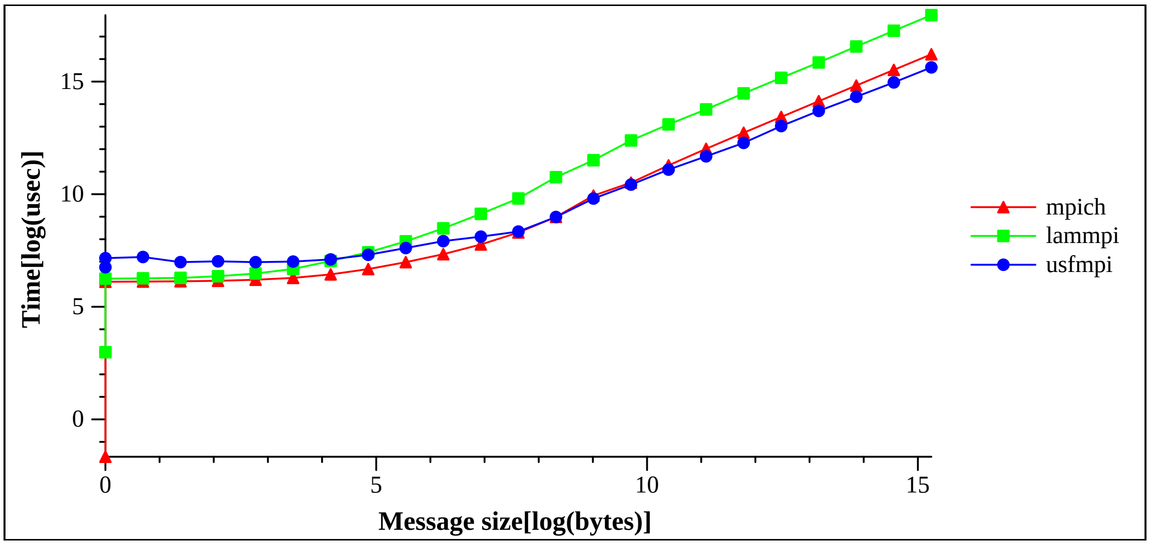

Figure 6.8: Allgather with 16 Processes.

6.4 Allreduce

The allreduce benchmark uses the `MPI_Allreduce()` operation. The `MPI_FLOAT` datatype and the `MPI_SUM` operator are used in the reduce operation. Figures 6.9, 6.10, 6.11, 6.12, 6.13 illustrate the results of this benchmark. As in the allgather benchmarks, USFMPI performs noticeably better than the other two MPI implementations. Again, LAM-MPI performs worst, and MPICH is in the middle.

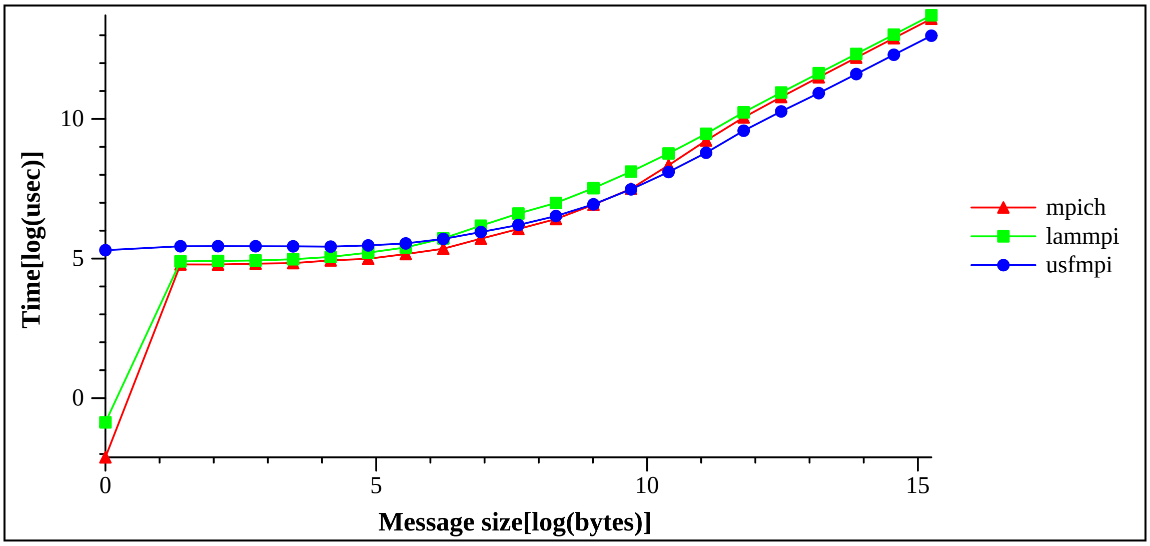

Figure 6.9: Allreduce with 2 Processes.

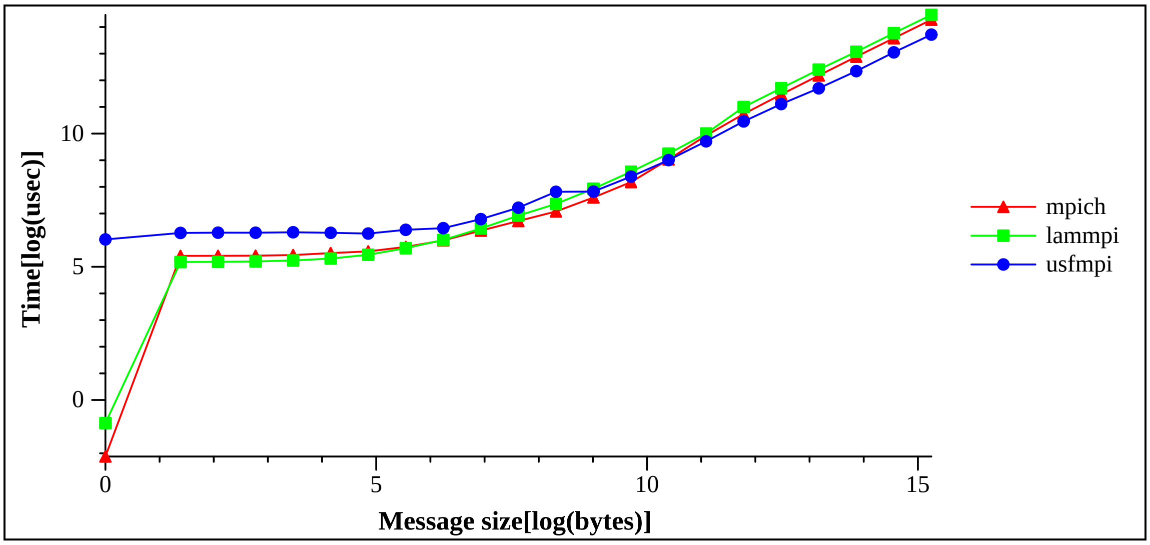

Figure 6.10: Allreduce with 4 Processes.

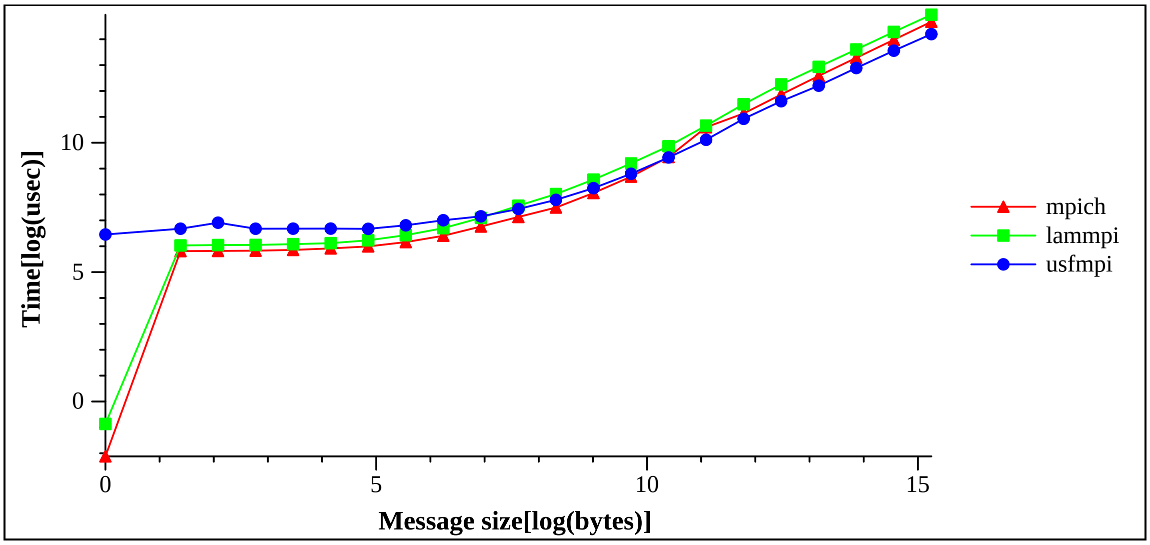

Figure 6.11: Allreduce with 8 Processes.

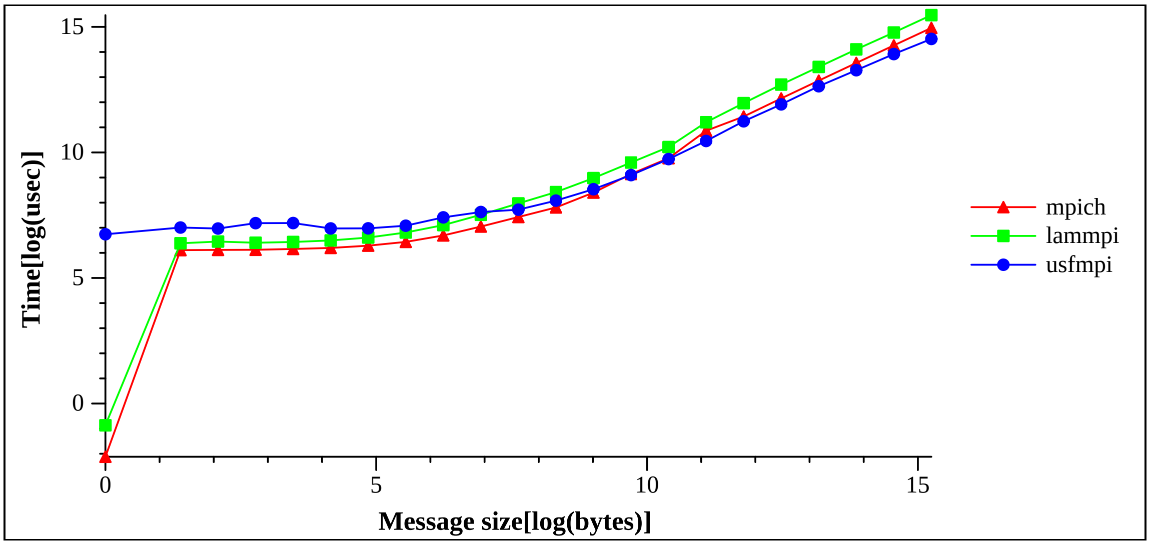

Figure 6.12: Allreduce with 16 Processes.

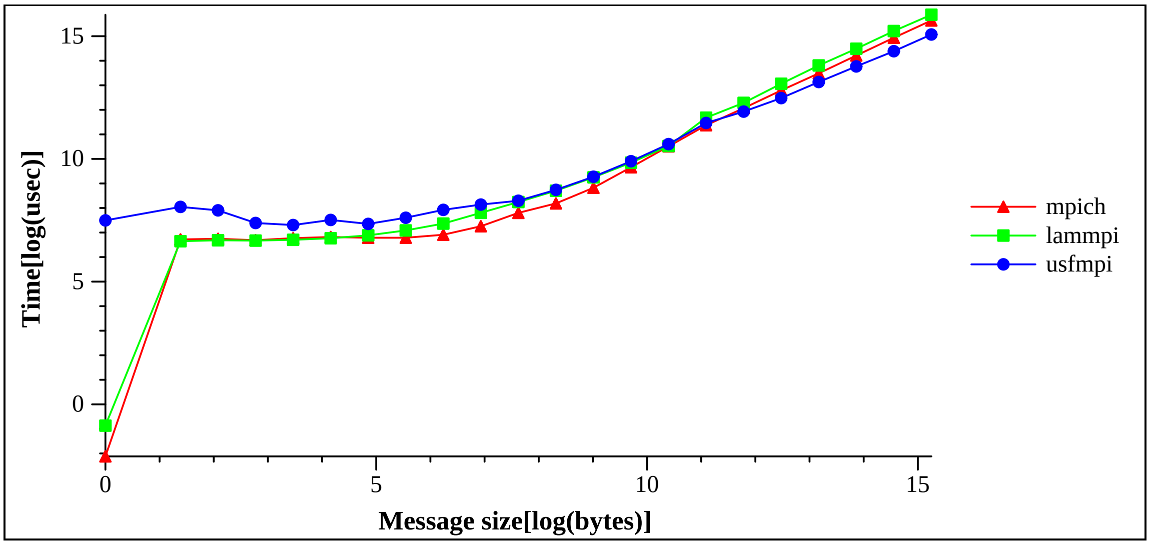

Figure 6.13: Allreduce with 32 Processes.

6.5 Alltoall

The alltoall benchmark uses the `MPI_Alltoall()` operation. Figures 6.14, 6.15, 6.16, 6.17, 6.18 illustrate the results of this benchmark. MPICH performs the worst in all five of the tests. There is no single best implementation for this benchmark. For the cases that involves 2 and 4 processes USFMPI and LAM-MPI are almost the same, USFMPI being a little bit better. With 8 and 16 processes, USFMPI performs best and with 32 processes LAM-MPI performs best.

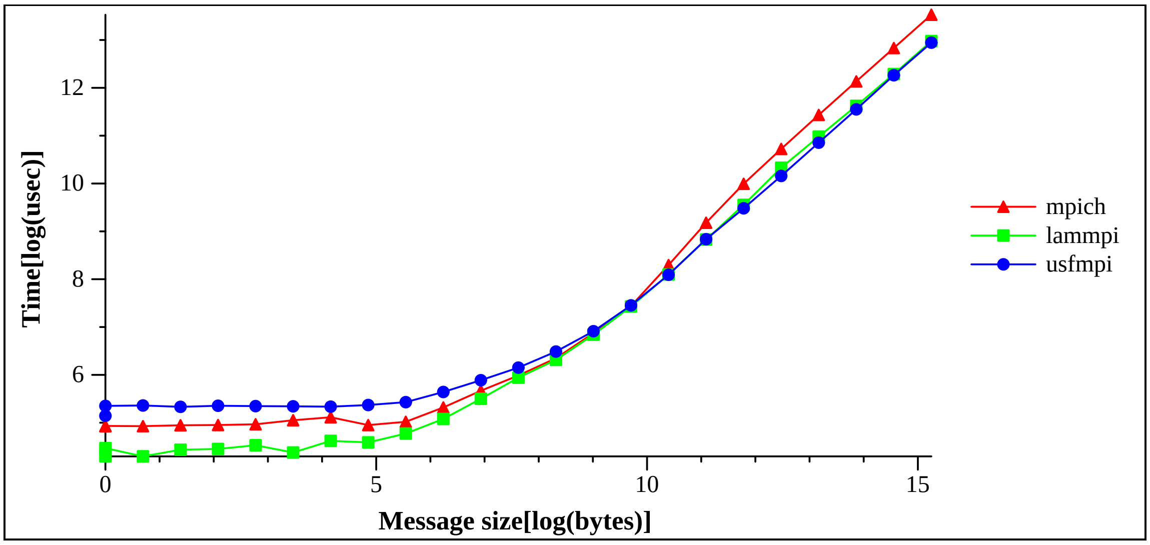

Figure 6.14: Alltoall with 2 Processes.

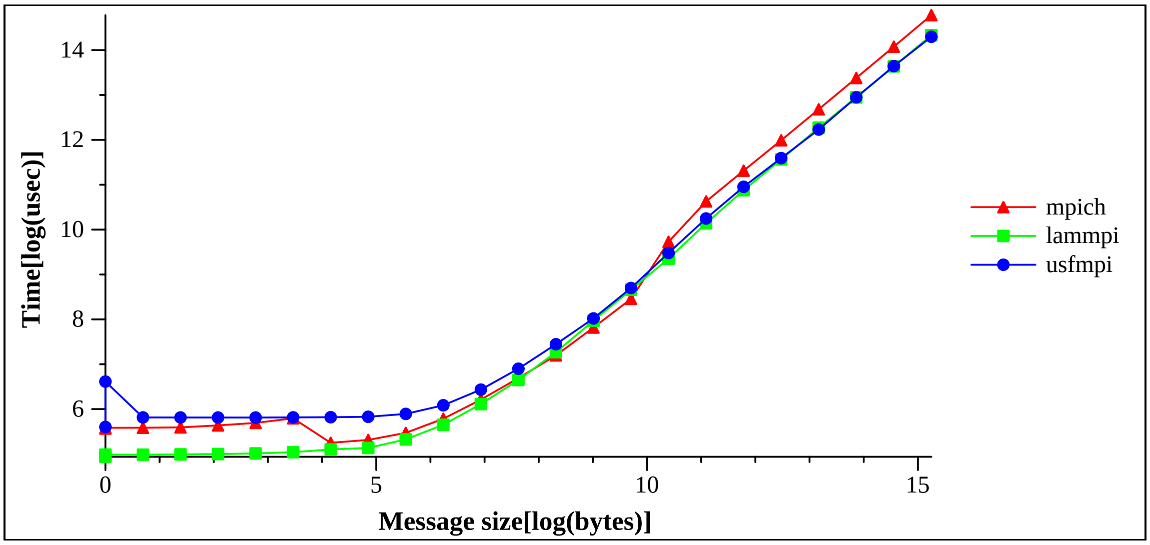

Figure 6.15: Alltoall with 4 Processes.

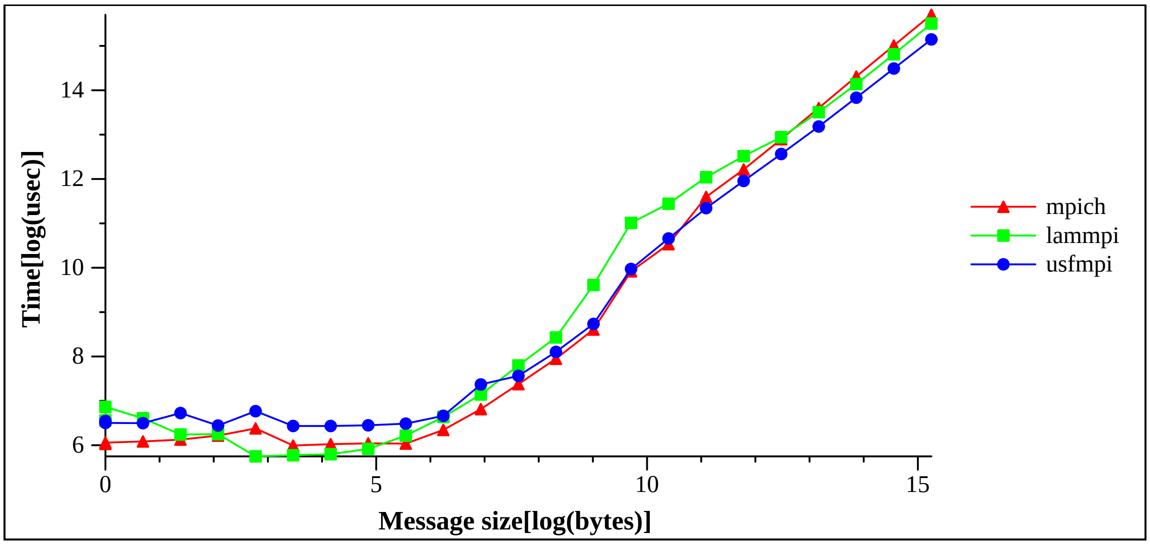

Figure 6.16: Alltoall with 8 Processes.

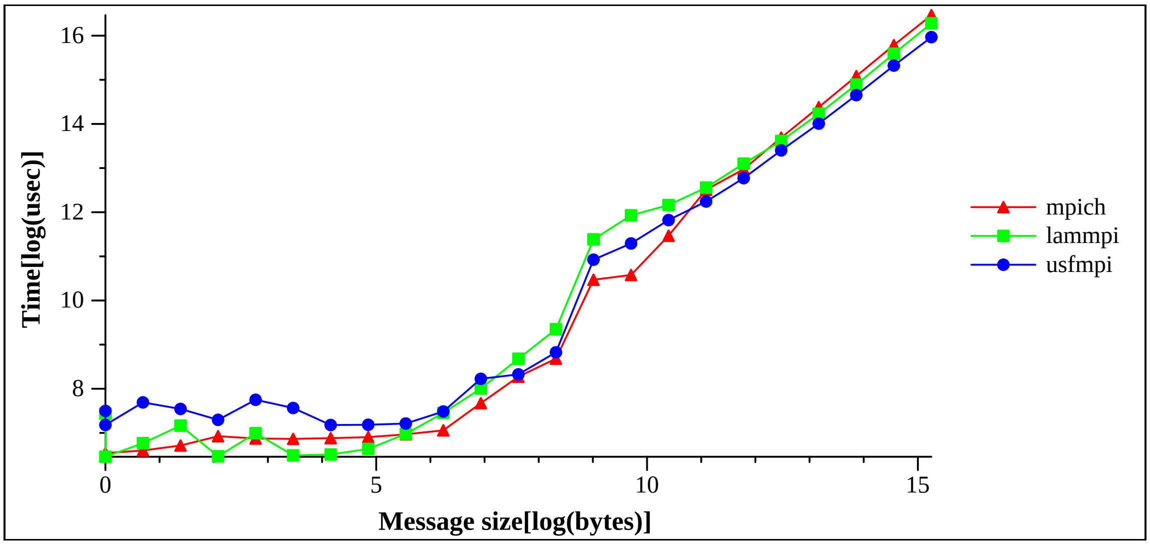

Figure 6.17: Alltoall with 16 Processes.

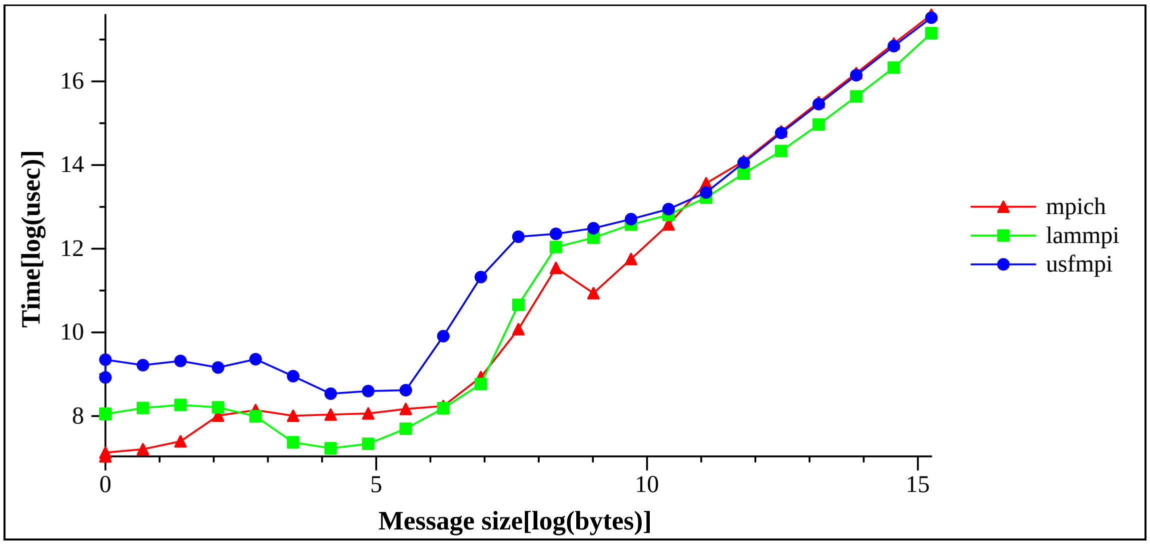

Figure 6.18: Alltoall with 32 Processes.

6.6 Broadcast

The broadcast benchmark uses the `MPI_Bcast()` operation. Figures 6.19, 6.20, 6.21, 6.22, 6.23 illustrate the results of this benchmark. USFMPI starts out as the best performer with a small number of processes, i.e., 2 and 4. As the number of processes increase, MPICH starts performing the best. USFMPI performs the second for these cases.

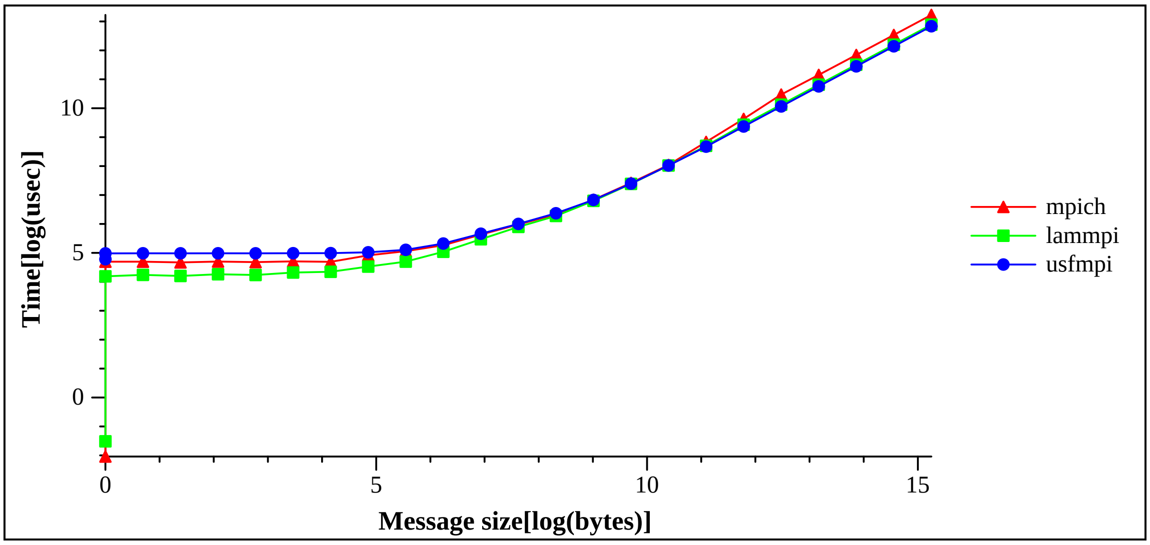

Figure 6.19: Bcast with 2 Processes.

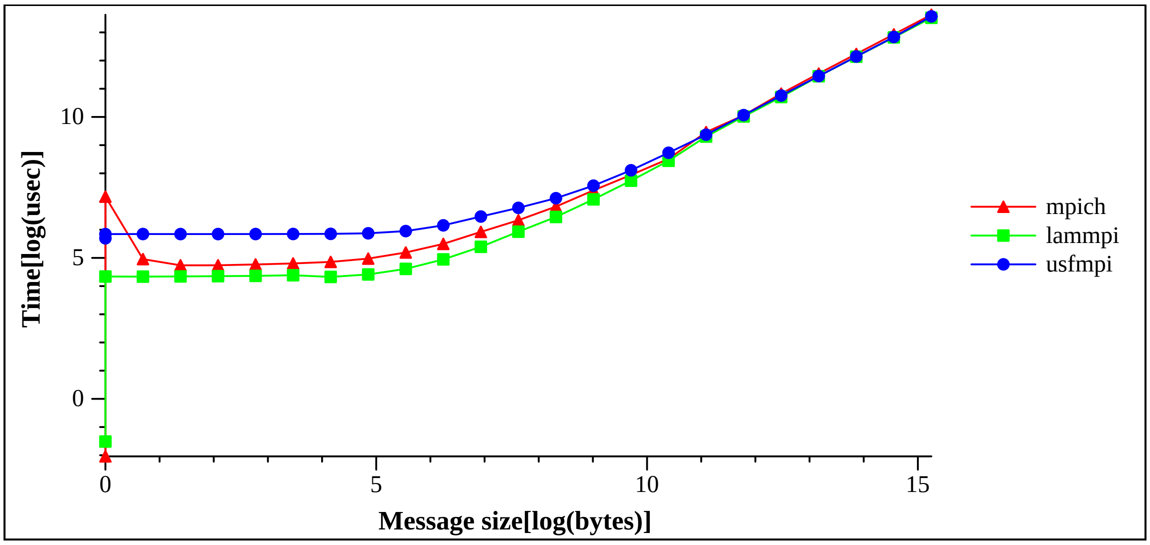

Figure 6.20: Bcast with 4 Processes.

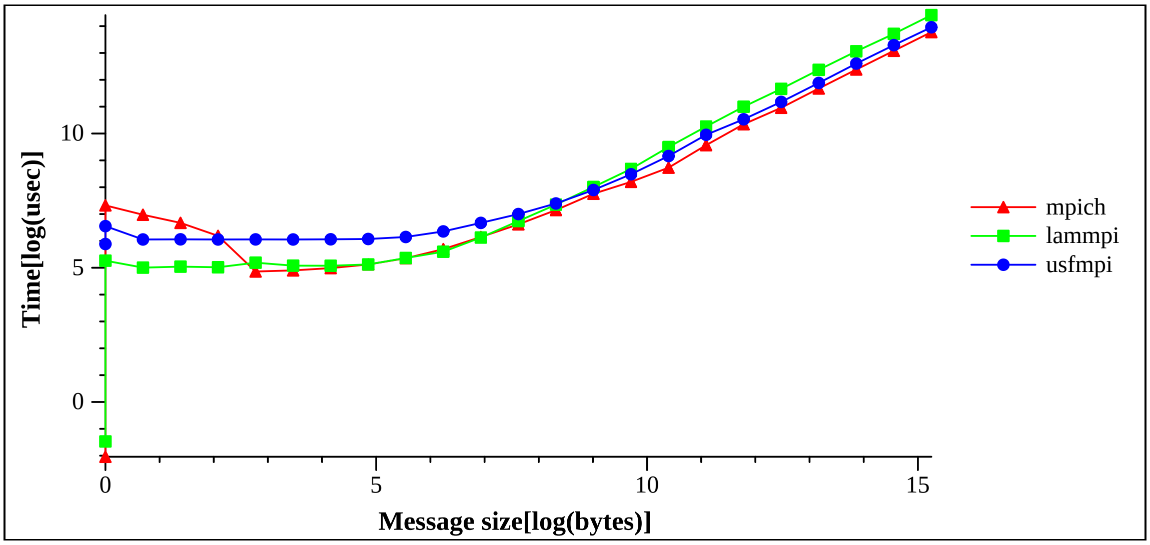

Figure 6.21: Bcast with 8 Processes.

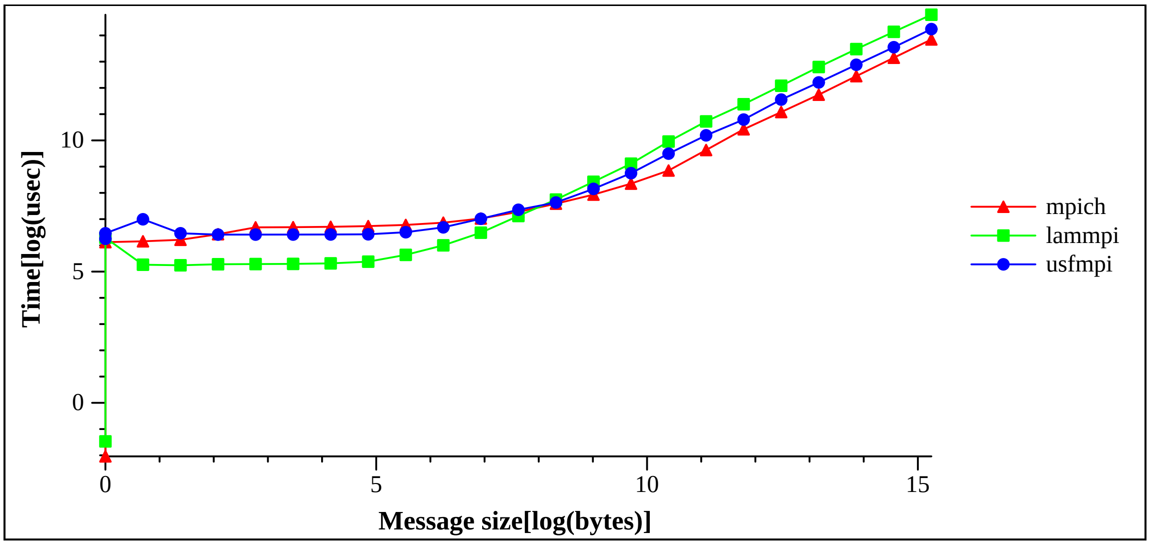

Figure 6.22: Bcast with 16 Processes.

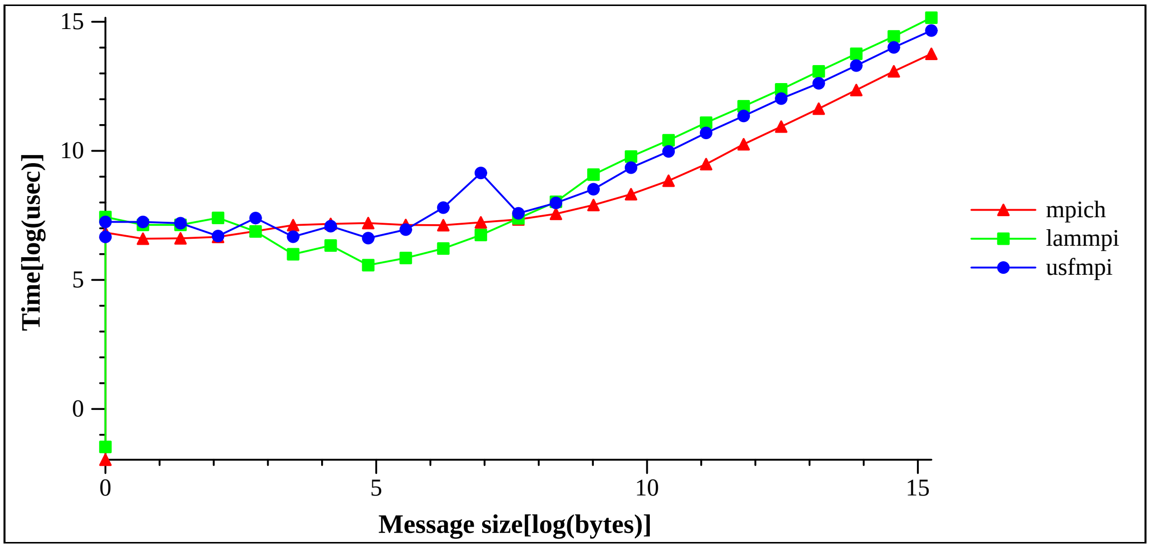

Figure 6.23: Bcast with 32 Processes.

6.7 Exchange

The exchange benchmark uses multiple blocking and nonblocking point to point calls. Exchange is a communication pattern that often occurs in grid splitting algorithms. The processes are seen as a periodic chain; each process exchanges data with both the left and right neighbors in the chain. Each process calls the functions in the same order. First, a message is sent to each neighbor with the `MPI_Isend()` operation, then the neighbors's data is received by `MPI_Recv()` operations. Finally, `MPI_Waitall()` is issued for the nonblocking send calls.

Figures 6.24, 6.25, 6.26, 6.27, 6.28 illustrate the results of this benchmark. MPICH performs worst in this benchmark. The exchange benchmark is conceptually very similar to the PingPing benchmark. So this behavior is expected and is due to the single threaded design. For tests with 2, 8 and 16 processes USFMPI performs best and for the rest of the tests LAM-MPI performs best.

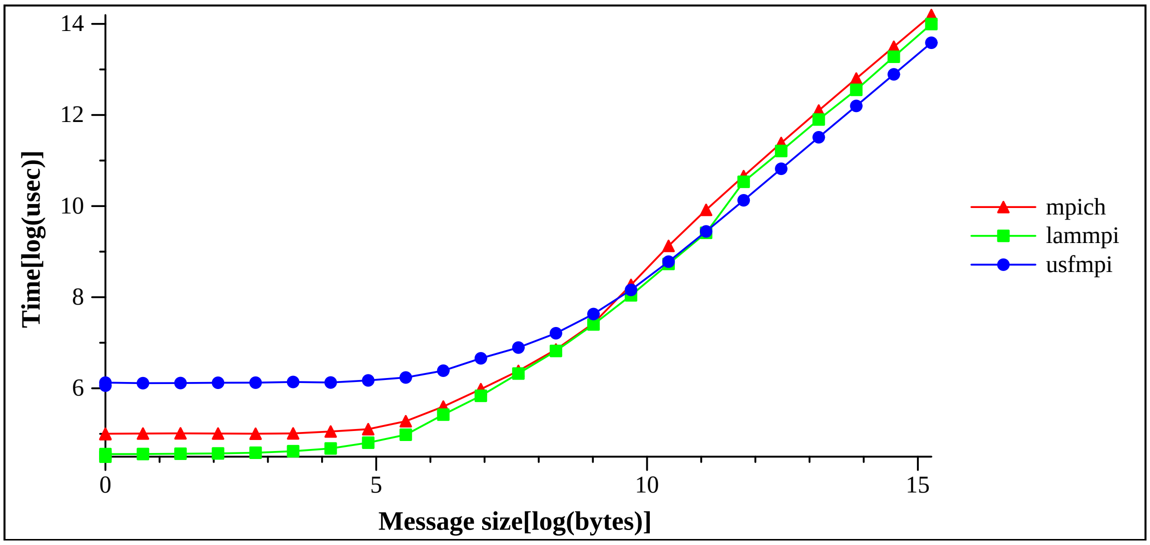

Figure 6.24: Exchange with 2 Processes.

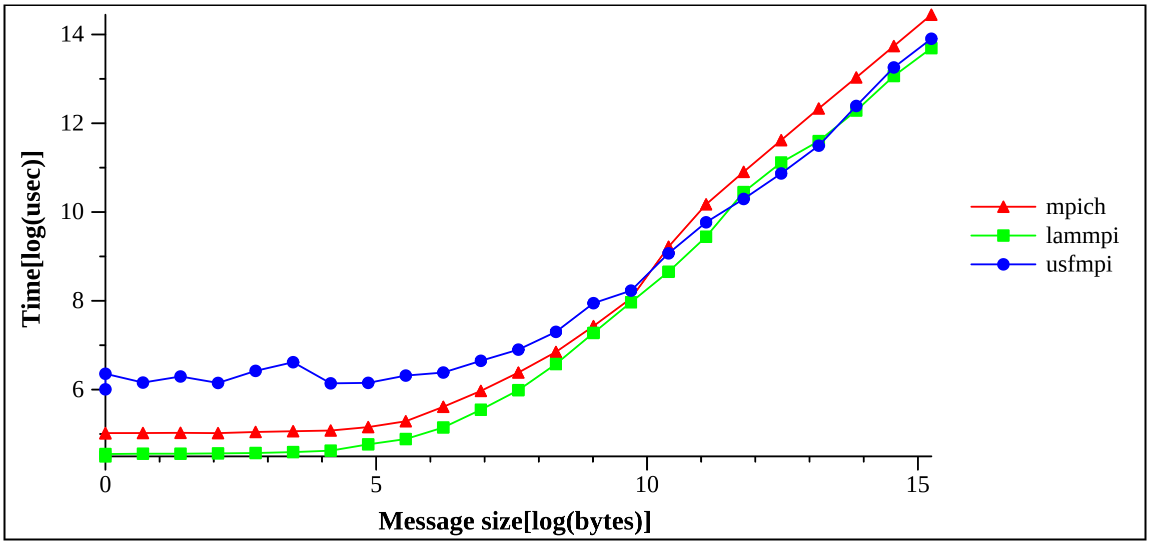

Figure 6.25: Exchange with 4 Processes.

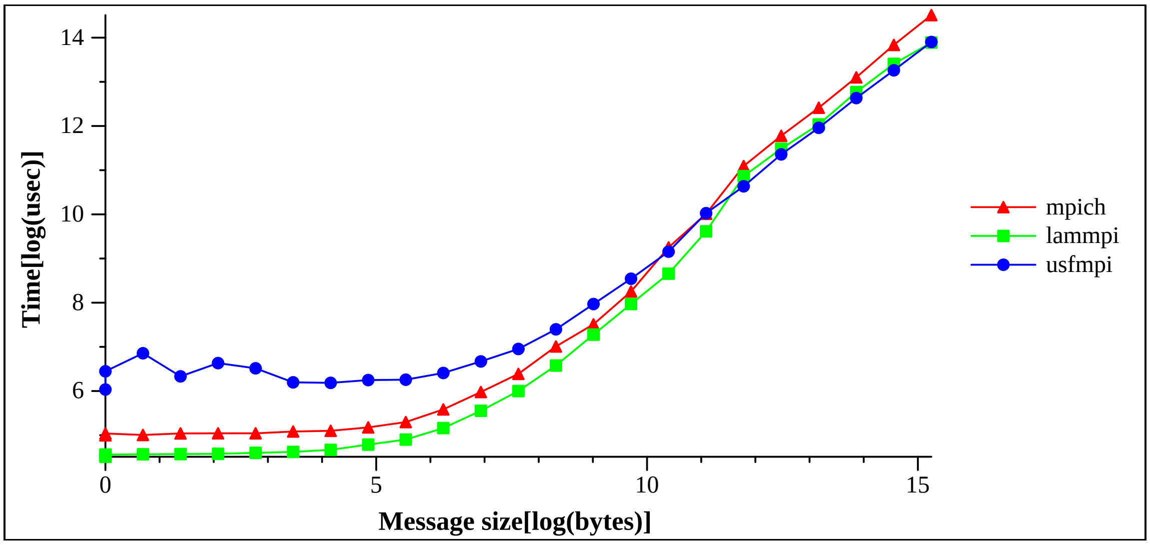

Figure 6.26: Exchange with 8 Processes.

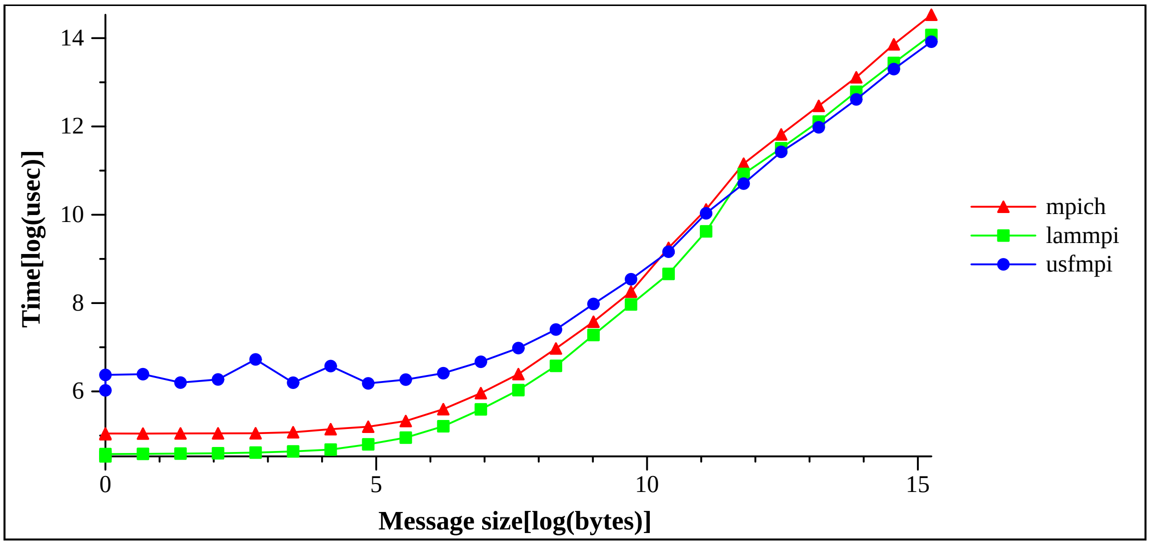

Figure 6.27: Exchange with 16 Processes.

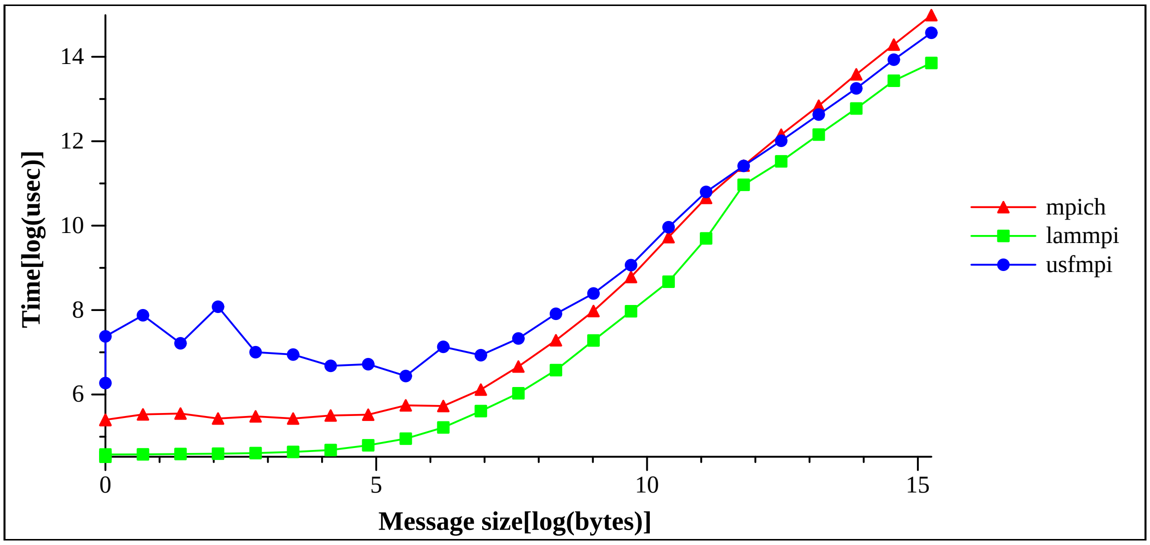

Figure 6.28: Exchange with 32 Processes.

6.8 Reduce

The reduce benchmark uses the `MPI_Reduce()` operation. The `MPI_FLOAT` data type and the `MPI_SUM` operator are used in the reduce operation. Figures 6.29, 6.30, 6.31, 6.32, 6.33 illustrate the results of this benchmark. The tests involving 2, 4, 8, and 16 processes are very similar to each other, USFMPI performs best, MPICH performs second, and LAM-MPI performs third. LAM-MPI changes its behavior in the test with 32 processes and performs the best, while MPICH and USFMPI stay the same.

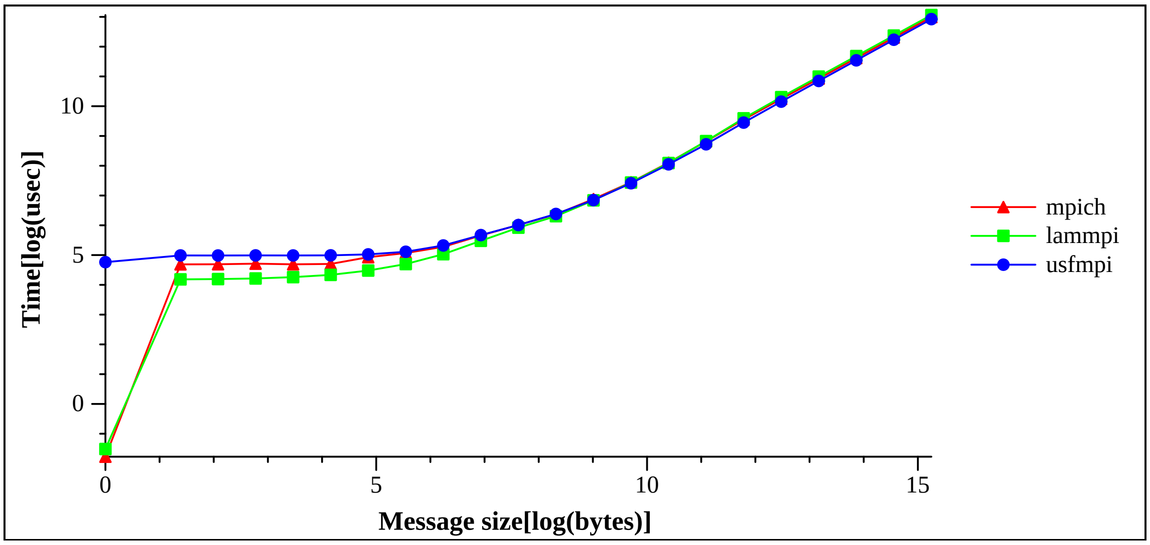

Figure 6.29: Reduce with 2 Processes.

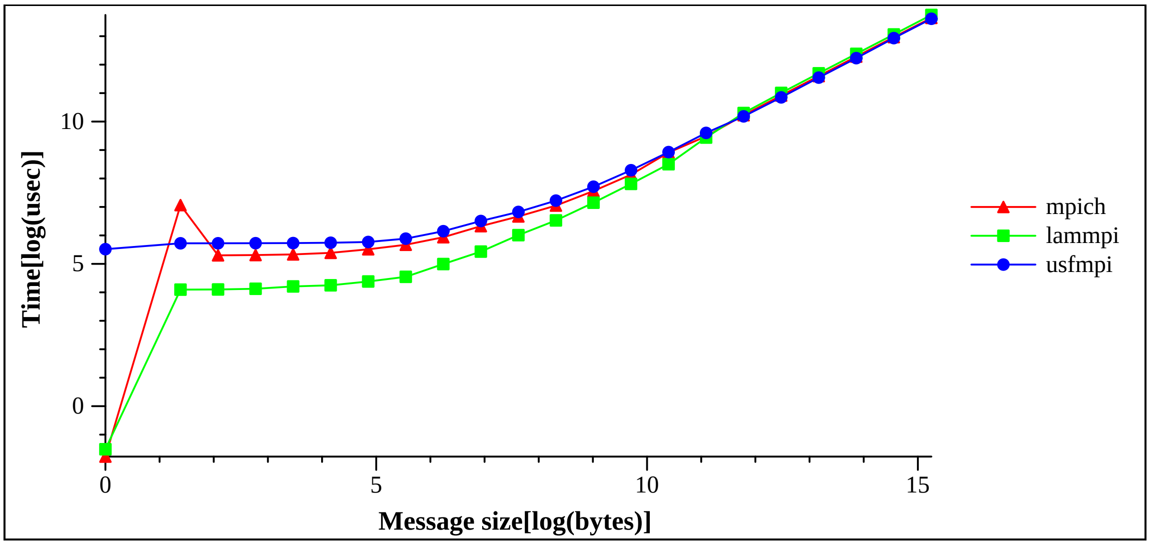

Figure 6.30: Reduce with 4 Processes.

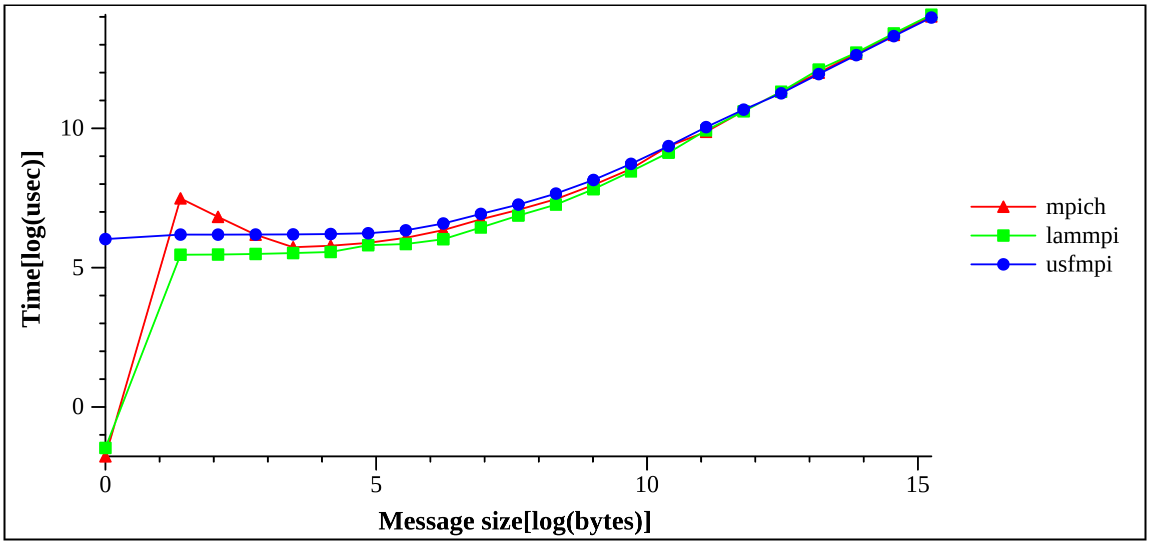

Figure 6.31: Reduce with 8 Processes.

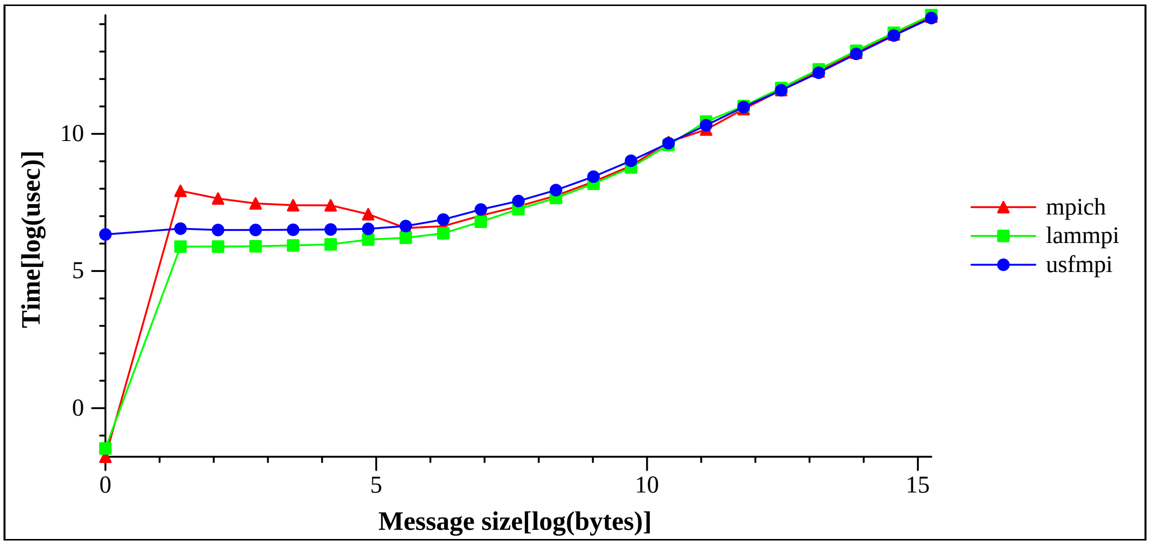

Figure 6.32: Reduce with 16 Processes.

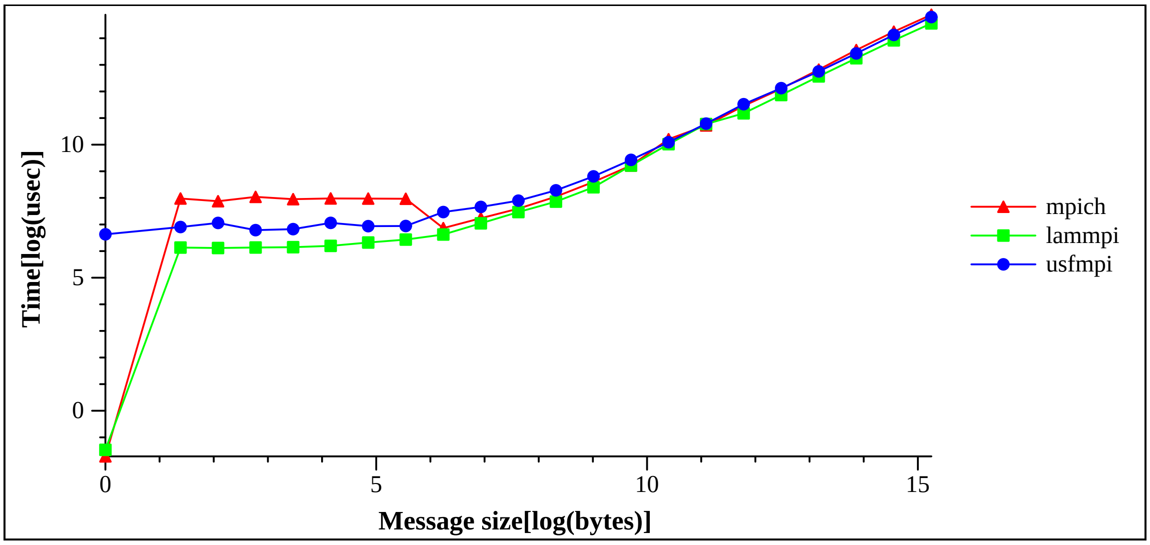

Figure 6.33: Reduce with 32 Processes.

6.9 Send-Receive

The send-receive benchmark uses the `MPI_Sendrecv()` operation. Each process with rank p issues a sendrecv call to process $p+1$. The last process is paired with process 0. Figures 6.34, 6.35, 6.36, 6.37, 6.38 illustrate the results of this benchmark. `MPI_Sendrecv()` is usually implemented with nonblocking send and receive calls, or a blocking and a nonblocking call, which effectively have the same performance.[2] Due to the involvement of nonblocking calls, MPICH performs the last in this benchmark. USFMPI and LAM-MPI have similar results, but as the number of processes in the test increases LAM-MPI's performance gap increases.

[2]USFMPI uses nonblocking calls to implement this operation.

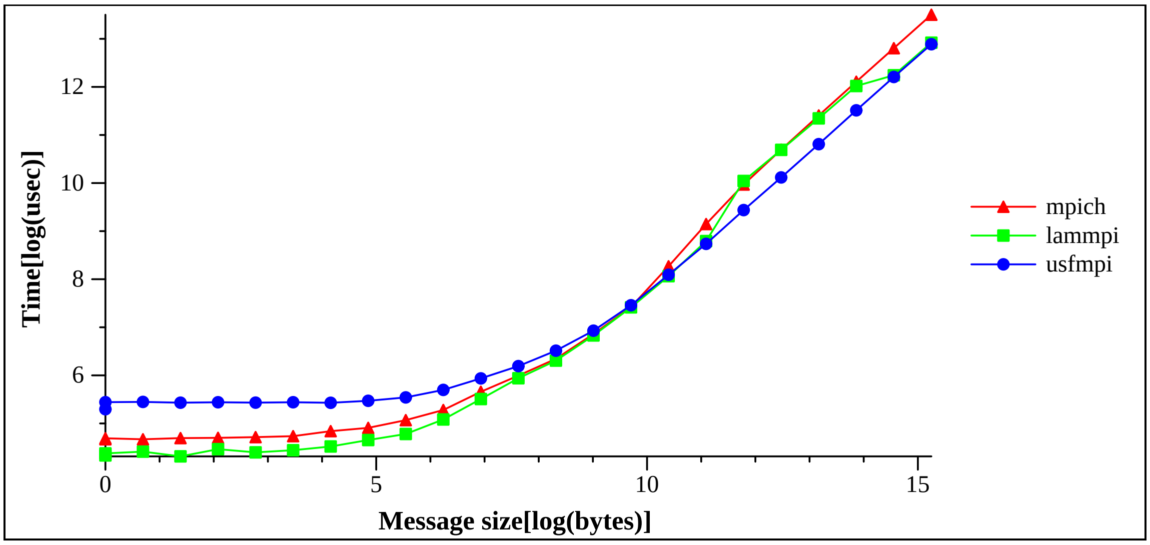

Figure 6.34: Sendrecv with 2 Processes.

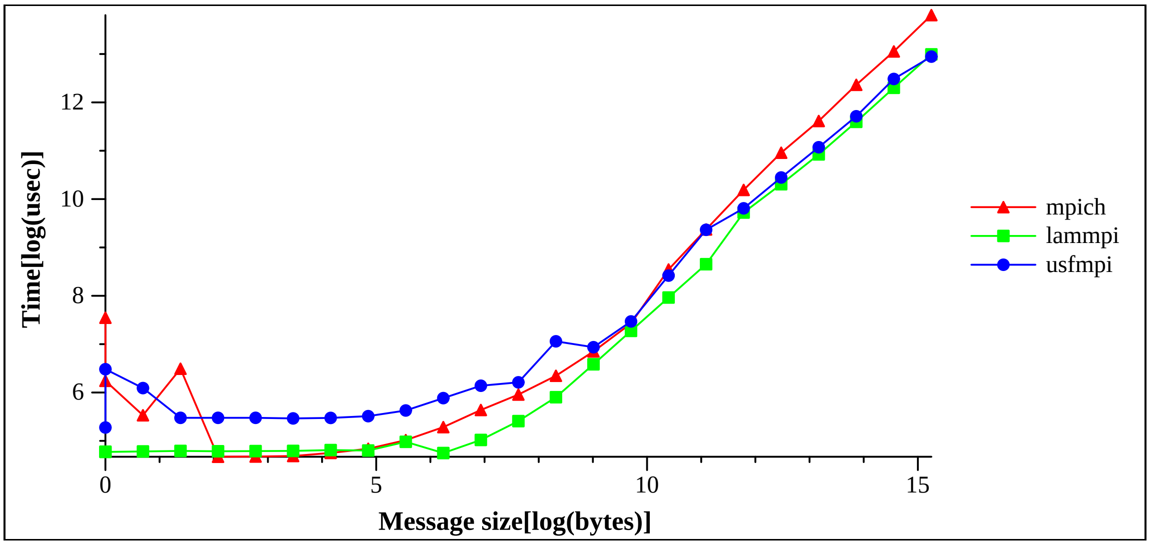

Figure 6.35: Sendrecv with 4 Processes.

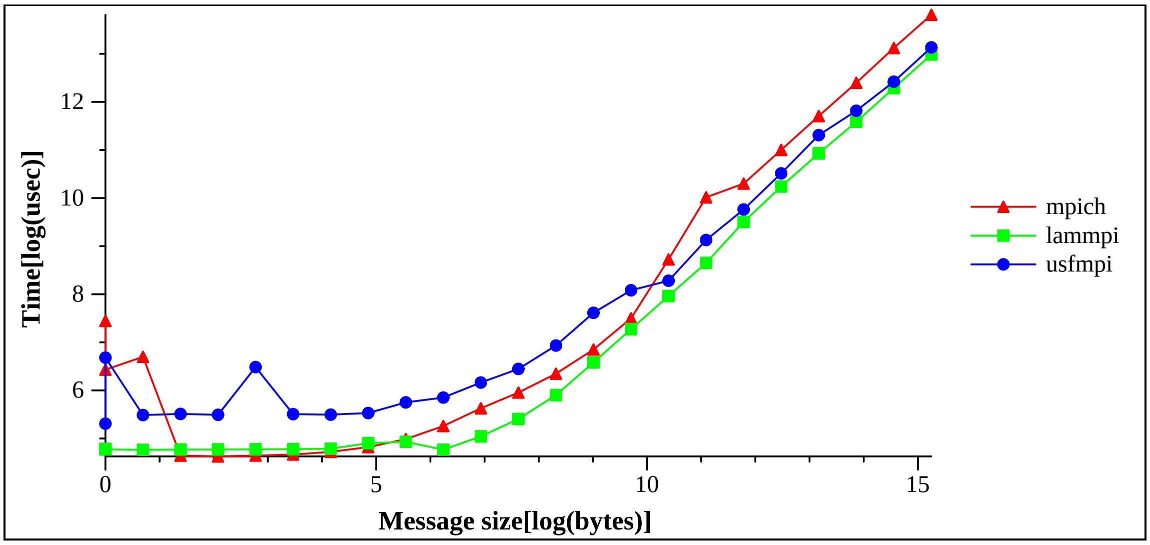

Figure 6.36: Sendrecv with 8 Processes.

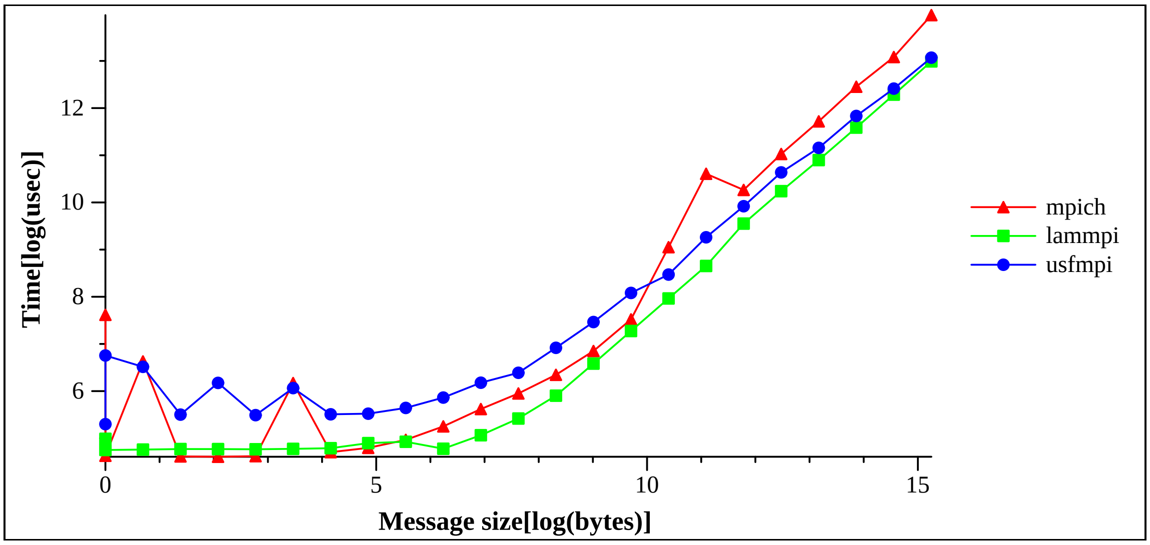

Figure 6.37: Sendrecv with 16 Processes.

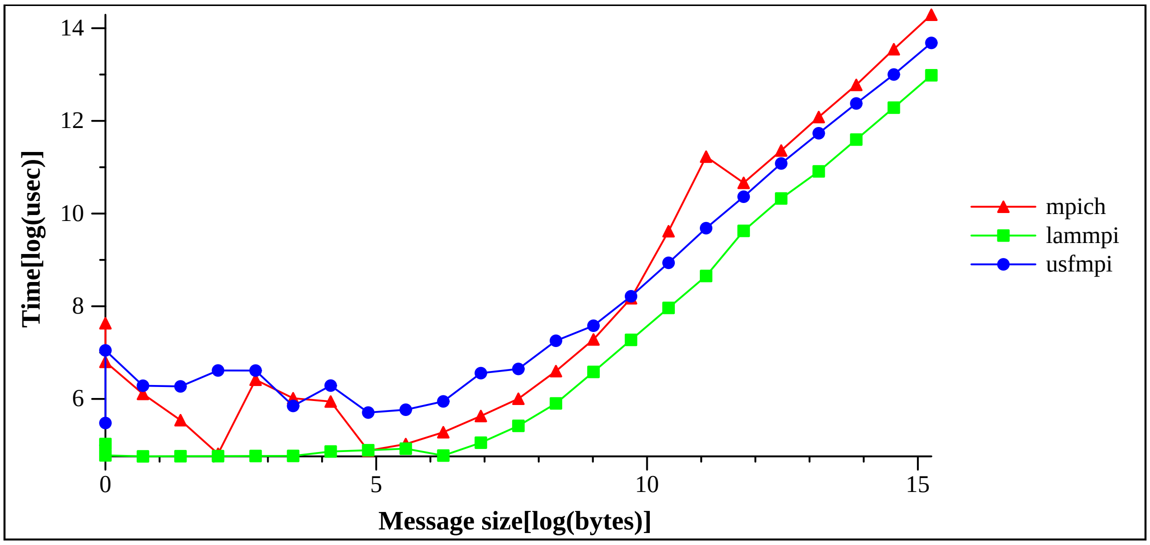

Figure 6.38: Sendrecv with 32 Processes.

Chapter 7

Conclusions and Future Work

This chapter discusses the major results that were observed in the Pallas benchmarks, the design process, major technical difficulties in implementation, debugging issues, cluster issues, and future work for USFMPI.

7.1 Major Results

As mentioned in the introduction, the motivation behind developing USFMPI was to provide a comprehensible and easy to modify implementation of MPI to the distributed programming community. Unlike most other open-source MPI implementations, USFMPI is not based on any other previously existing message-passing system. This reduces the complexity and the number of kludges in the implementation. Some of the complexity in MPICH and LAM-MPI comes from tailoring old systems.

USFMPI also has performed best in the majority of the PALLAS benchmarks, except for the cases where the message size is less than 1024 bytes. The point to point tests show that separating communication and computation with two threads has a major positive impact on nonblocking communications. Both USFMPI and LAM-MPI have this advantage over MPICH. USFMPI performs better than LAM-MPI, proving that the message processing overheads of LAM-MPI are larger. The collective benchmark results show that USFMPI's collective routines perform well if not best for all types of collective operations. With the increasing popularity of Linux systems, having a specialized, compact version of MPI for Linux is very useful, since it performs better than the generic implementations for most of the frequently used operations.

7.2 Major Technical Difficulties

The hardest part of implementing USFMPI was to synchronize and communicate between threads. Timing is very important in this type of programming, especially due to signaling sleeping threads and locking shared variables. It is extremely difficult to create test cases that cover all the possible types of sequences that can occur due to the physical network delays. Debugging also affects the timing; it is often the case that an error will stop occurring with the presence of a `printf()` statement simply because it adds a delay.

Using a debugger is also very hard with a distributed system, especially if the system is not mature. Usually the execution of an MPI program is halted with an `MPI_Barrier()` and a `scanf()` at master node, so that the programmer can attach a debugging tool like `gdb` to the running processes. This approach does not fully apply to implementing MPI, because the collective and point to point calls are being debugged. Many blocking or nonblocking calls are likely to occur in a `MPI_Barrier()` call. Another downside in using a debugger is the presence of two threads per MPI process; a debugger must be attached to each one. Note that this process has to be repeated every time the program is run due to the changing process IDs.

It is very time consuming to set up the debugging system, so printing data is often a better solution. However, this also has its downsides. Even though all of the output is forwarded to one terminal; it is almost impossible to understand in which order the events occurred, due to the network delays. It is a good practice to have a rank printed with each message, and then query the output for these rank tags to see how much each process accomplished. The order of the messages can also be used, but with caution; the output does not always show the real order.

7.3 Future Work

Currently USFMPI only has the most frequently used and most important functions of the MPI specification [29]. It can be made complete by implementing the missing functions. Most of the point to point functions that are introduced in the third chapter of the MPI specification have been implemented. The only missing functions are `MPI_Buffer_attach`, `MPI_Buffer_detach`, and `MPI_Cancel`. The buffer operations enable the user to use her own buffers in the buffered send mode. `MPI_Cancel` is used to cancel nonblocking communication requests.

Collective communication routines are fully implemented. However the vector calls such as `MPI_Allgatherv` are not fully optimized. `MPI_Comm_group`, `MPI_Comm_free`, `MPI_Comm_split`, `MPI_Group_incl`, and `MPI_Comm_create` are the only implemented functions from group and communicator management, so that new groups and communicators can be defined but more complex operations are not supported. None of the functions from process topologies that are described in the sixth chapter of the MPI specification and none of the error handling functions are implemented.

Bibliography

[1] Freebsd organization, `http://www.freebsd.org`.

[2] Linux organization, `http://www.linux.org`.

[3] Argonne National Laboratory. *MPICH 1.2.5, A Portable Implementation of MPI*, 2003. `http://www-unix.mcs.anl.gov/mpi/mpich/`.

[4] K. E. Batcher. The massively parallel processor (MPP). In *Digest of Papers, Compcon85 Thirtieth IEEE Computer society International Conference*, pages 21–24, San Francisco, CA, February 1985. IEEE Computer Society.

[5] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King K. Su. Myrinet — A gigabit-per-second local-area-network. *IEEE Micro*, 15(1):29–36, February 1995.

[6] Moshe Braner. *Trollius User's Manual*. Cornell Theory Center, Ithaca, NY, May 1988.

[7] Ralph Butler and Ewing Lusk. User's guide to the p4 parallel programming system. Technical Report ANL-92/17, Argonne National Laboratory, October 1992. Version 1.4.

[8] David L. Fielding, Moshe Braner, James R. Beers, Jr., and Roslyn Leibensperger. The trollius programming environment for multicomputers. In A. S. Wagner, editor, *Transputer Research and Applications 3*, pages 207–210, Amsterdam, 1990. IOS Press.

[9] MPI Forum. MPI: A message-passing interface. Technical Report CSE-94-013, Oregon Graduate Institute of Science and Technology, April 1994.

[10] Geoffrey C. Fox. What have we learnt from using real parallel machines to solve real problems? In *The 3rd Conference on Hypercube Concurrent Computers and Applications*, volume II, Applications, pages 897–955, Pasadena, CA, January 1988. ACM. cccp-522.

[11] Pallas GmbH. *Pallas Benchmarks*, 2003. `http://http://www.pallas.com/e/products/pmb/`.

[12] William Gropp and Ewing Lusk. *An abstract device definition to support the implementation of a high-level point-to-point message passing interface*. Preprint MCS-P342-1193. Argonne National Laboratory, 1994.

[13] William Gropp and Ewing Lusk. Mpich working note: Creating a new mpich device using the channel interface. Technical Report ANL/MCS-TM-213, Argonne National Laboratory, 1995.

[14] Hewlett Packard. *HP MPI User's Guide Seventh Edition, HP Part No. B6060-96009*, June 2002.

[15] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Houston, Texas, 1993.

[16] W. Daniel Hillis. *The Connection Machine*. Series in Artificial Inteligence. MIT Press, Cambridge, MA, 1985.

[17] Y. Charlie Hu, Honghui Lu, Alan L. Cox, and Willy Zwaenepoel. OpenMP for networks of SMPs. *Journal of Parallel and Distributed Computing*, 60(12):1512–1530, 2000.

[18] Zhihong Hu and Etienne Barnard. Efficient estimation of perceptual features for speech recognition. In *Proc. Eurospeech '97*, pages 493–496, Rhodes, Greece, 1997.

[19] IBM. *IBM Parallel Environment for AIX 5L: MPI Programming and Subroutine Reference, version 2 release 3, Document Number SA22-7422-01*, December 2001. `http://www.rs6000.ibm.com/resource/aix\_resource/sp\_books/pe/index.htm%l`.

[20] Oak Ridge National Laboratories. Parallel virtual machine. `http://www.epm.ornl.gov/pvm/pvm_home.html`.

[21] Bradford Nichols, Bick Buttlar, and Jackie Proulx Farrell. *Pthreads Programming.* O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1996.

[22] Ohio Supercomputer Center, Ohio State University. *MPI Primer / Developing with LAM*, November 1996. `http://www.lam-mpi.org`.

[23] Peter Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann, San Francisco, CA, 1997. http://www.usfca.edu/mpi (source programs available) and http://www.mkp.com.

[24] Reusch. The autonomous SIMD systems MP-1 and MP-2. In *SPAA: Annual ACM Symposium on Parallel Algorithms and Architectures*, 1993.

[25] Shuichi Sakai, Yuetsu Kodama, and Yoshinori Yamaguchi. Design and implementation of a circular omega network in the EM-4. *Parallel Computing*, 19(2):125–142, February 1993.

[26] Kragen Sitaker. Beowulf frequantly asked questions, 1999. `http://www.canonical.org/~kragen/beowulf-faq.txt`.

[27] Thomas Sterling. Beowulf-class clustered computing: Harnessing the power of parallelism in a pile of PCs. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, page 883, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann. Invited talk.

[28] Sun. *Sun MPI 6.0 Software Programming and Reference Manual, Part Number 817-0085-10*, 2002. `http://www.sun.com/products-n-solutions/hardware/docs/html/817-0085-10/%`.

[29] The MPI Forum. MPI: A message-passing interface standard. *The International Journal of Supercomputing Applications*, 8(3/4), 1994.

[30] Thinking Machines Corporation, Boston, MA. *Connection Machine Model CM-2 Technical Summary*, April 1987. Technical Report HA87-4.

[31] L. Torvalds. Linux kernel implementation. In *Proceedings of the AUUG94 Conference: Open systems. Looking into the future: 6–9 September 1994, World Congress Centre, Melbourne, Australia*, pages 9–14, Kensington, NSW, Australia, 1994. AUUG Inc.

[32] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1998.

Appendix A

Supported Parts of the MPI 1.2 Specification

This appendix presents the supported parts of MPI 1.2 specification in USFMPI.

A.1 Supported Functions

MPI_Abort
MPI_Address
MPI_Allgather
MPI_Allgatherv
MPI_Alltoall
MPI_Barrier
MPI_Bcast
MPI_Comm_create
MPI_Comm_free
MPI_Comm_group
MPI_Comm_rank
MPI_Comm_size
MPI_Comm_split
MPI_Finalize
MPI_Gather
MPI_Gatherv

MPI_Get_count

MPI_Group_incl

MPI_Group_translate_ranks

MPI_Init

MPI_Iprobe

MPI_Irecv

MPI_Irsend

MPI_Isend

MPI_Issend

MPI_Op_create

MPI_Op_free

MPI_Pack

MPI_Probe

MPI_Recv

MPI_Recv_init

MPI_Reduce

MPI_Reduce_scatter

MPI_Rsend

MPI_Rsend_init

MPI_Scatter

MPI_Scatterv

MPI_Send

MPI_Send_init

MPI_Sendrecv

MPI_Sendrecv_replace

MPI_Ssend

MPI_Ssend_init

MPI_Start

MPI_Startall

MPI_Test

MPI_Testall

MPI_Testany

MPI_Type_contiguous

MPI_Type_indexed

MPI_Type_size

MPI_Type_struct

MPI_Type_vector

MPI_Unpack

MPI_Wait

MPI_Waitall

MPI_Waitany

MPI_Wtime

A.2 Supported Datatypes

MPI_CHAR

MPI_SHORT

MPI_INT

MPI_LONG

MPI_UNSIGNED_CHAR

MPI_UNSIGNED_SHORT

MPI_UNSIGNED

MPI_UNSIGNED_LONG

MPI_FLOAT

MPI_DOUBLE

MPI_LONG_DOUBLE

MPI_BYTE

MPI_LONG_LONG

MPI_PACKED

MPI_LB

MPI_UB

MPI_FLOAT_INT

MPI_DOUBLE_INT

MPI_LONG_INT

MPI_SHORT_INT

MPI_2INT

MPI_LONG_DOUBLE_INT

A.3 Supported Reduction Operators

MPI_MAX

MPI_MIN

MPI_SUM

MPI_PROD

MPI_LAND

MPI_BAND

MPI_LOR

MPI_BOR

MPI_LXOR

MPI_BXOR

MPI_MINLOC

MPI_MAXLOC

A.4 Supported Errors

MPI_ERRHANDLER_NULL

MPI_ERROR

MPI_ERR_ACCESS

MPI_ERR_AMODE

MPI_ERR_ARG

MPI_ERR_BAD_FILE

MPI_ERR_BUFFER

MPI_ERR_COMM

MPI_ERR_CONVERSION

MPI_ERR_COUNT

MPI_ERR_DIMS

MPI_ERR_DUP_DATAREP

MPI_ERR_FILE

MPI_ERR_FILE_EXISTS

MPI_ERR_FILE_IN_USE

MPI_ERR_GROUP

MPI_ERR_INFO

MPI_ERR_INFO_KEY

MPI_ERR_INFO_NOKEY

MPI_ERR_INFO_VALUE

MPI_ERR_INTERN

MPI_ERR_IN_STATUS

MPI_ERR_IO

MPI_ERR_LASTCODE

MPI_ERR_NAME

MPI_ERR_NOMEM

MPI_ERR_NOT_SAME

MPI_ERR_NO_SPACE

MPI_ERR_NO_SUCH_FILE

MPI_ERR_OP

MPI_ERR_OTHER

MPI_ERR_PENDING

MPI_ERR_PORT

MPI_ERR_QUOTA

MPI_ERR_RANK

MPI_ERR_READ_ONLY

MPI_ERR_REQUEST

MPI_ERR_ROOT

MPI_ERR_SERVICE

MPI_ERR_SPAWN

MPI_ERR_TAG

MPI_ERR_TOPOLOGY

MPI_ERR_TRUNCATE

MPI_ERR_TYPE

MPI_ERR_UNKNOWN

MPI_ERR_UNSUPPORTED_DATAREP

MPI_ERR_UNSUPPORTED_OPERATION

MPI_ERR_WIN

Appendix B

Source Code

This appendix presents the USFMPI source code.

B.1 mpi.h

```
#ifndef _MPI_H
#define _MPI_H 1

#include <pthread.h>
#include "pool.h"

#define _USF_SERVPORT 10239
#define _USF_MAXDATASIZE 1024*64
#define _USF_MAXPROCESSORS 128

// - Begin - by Qing Huang

#define SINGLE_THREAD
//#define DEBUG
#define _USF_MPI_TCP_
//#define _USF_MPI_GM_


/* If TCP is used  - Macros are used for now, function indirection may
   be used later */


#ifdef _USF_MPI_TCP_
#define _USF_send_all  _USF_TCP_send_all
#define _USF_recv_all  _USF_TCP_recv_all
#define _USF_finalize  _USF_TCP_Finalize
#endif

#ifdef _USF_MPI_GM_
#define _USF_send_all  _USF_GM_send_all
#define _USF_recv_all  _USF_GM_recv_all
#define _USF_finalize _USF_GM_Finalize
#define _USF_MAX_HEADERS_ 64
#define _USF_MAX_DMA_DATA_BUFFER 16
#define MAX_MEM_REG_ENTRY 1024
```

```
void *data_buffer[_USF_MAX_DMA_DATA_BUFFER];


typedef struct dma_mem_reg_table
{
  void *p;
  int size;
  struct dma_mem_reg_table *duplicate_link;
}
dma_mem_reg_table;


#endif

unsigned int send_post_account[_USF_MAXPROCESSORS];
unsigned int recv_post_account[_USF_MAXPROCESSORS];

//unsigned int sending_requests[_USF_MAXPROCESSORS] ;
//unsigned int recv_pending[_USF_MAXPROCESSORS] ;


/* added by Qing Huang */

#define MPI_COMM_NULL      ((MPI_Comm)0)
#define MPI_OP_NULL        ((MPI_Op)0)
#define MPI_GROUP_NULL     ((MPI_Group)0)
#define MPI_DATATYPE_NULL  ((MPI_Datatype)0)
#define MPI_REQUEST_NULL   ((MPI_Request)0)
#define MPI_ERRHANDLER_NULL ((MPI_Errhandler )0)
typedef int MPI_Errhandler;
typedef int MPI_Group;
/* Pre-defined constants */
#define MPI_UNDEFINED      (-32766)
#define MPI_UNDEFINED_RANK MPI_UNDEFINED
#define MPI_KEYVAL_INVALID 0

#define MPI_SUCCESS          0  /* Successful return code */
/* Communication argument parameters */
#define MPI_ERR_BUFFER       1  /* Invalid buffer pointer */
#define MPI_ERR_COUNT        2  /* Invalid count argument */
#define MPI_ERR_TYPE         3  /* Invalid datatype argument */
#define MPI_ERR_TAG          4  /* Invalid tag argument */
#define MPI_ERR_COMM         5  /* Invalid communicator */
#define MPI_ERR_RANK         6  /* Invalid rank */
#define MPI_ERR_ROOT         7  /* Invalid root */
#define MPI_ERR_TRUNCATE    14  /* Message truncated on receive */

/* MPI Objects (other than COMM) */
#define MPI_ERR_GROUP        8  /* Invalid group */
#define MPI_ERR_OP           9  /* Invalid operation */
#define MPI_ERR_REQUEST     19  /* Invalid mpi_request handle */

/* Special topology argument parameters */
#define MPI_ERR_TOPOLOGY    10  /* Invalid topology */
#define MPI_ERR_DIMS        11  /* Invalid dimension argument */

/* All other arguments.  This is a class with many kinds */
#define MPI_ERR_ARG         12  /* Invalid argument */

/* Other errors that are not simply an invalid argument */
#define MPI_ERR_OTHER       15  /* Other error; use Error_string */
```

```
#define MPI_ERR_UNKNOWN     13  /* Unknown error */
#define MPI_ERR_INTERN      16  /* Internal error code    */

/* Multiple completion has two special error classes */
#define MPI_ERR_IN_STATUS   17  /* Look in status for error value */
#define MPI_ERR_PENDING     18  /* Pending request */

/* New MPI-2 Error classes */
#define MPI_ERR_FILE        27  /* */
#define MPI_ERR_ACCESS      20  /* */
#define MPI_ERR_AMODE       21  /* */
#define MPI_ERR_BAD_FILE    22  /* */
#define MPI_ERR_FILE_EXISTS 25  /* */
#define MPI_ERR_FILE_IN_USE 26  /* */
#define MPI_ERR_NO_SPACE    36  /* */
#define MPI_ERR_NO_SUCH_FILE 37 /* */
#define MPI_ERR_IO          32  /* */
#define MPI_ERR_READ_ONLY   40  /* */
#define MPI_ERR_CONVERSION  23  /* */
#define MPI_ERR_DUP_DATAREP 24  /* */
#define MPI_ERR_UNSUPPORTED_DATAREP   43       /* */

/* MPI_ERR_INFO is NOT defined in the MPI-2 standard.  I believe that
   this is an oversight */
#define MPI_ERR_INFO        28  /* */
#define MPI_ERR_INFO_KEY    29  /* */
#define MPI_ERR_INFO_VALUE  30  /* */
#define MPI_ERR_INFO_NOKEY  31  /* */

#define MPI_ERR_NAME        33  /* */
#define MPI_ERR_NOMEM       34  /* */
#define MPI_ERR_NOT_SAME    35  /* */
#define MPI_ERR_PORT        38  /* */
#define MPI_ERR_QUOTA       39  /* */
#define MPI_ERR_SERVICE     41  /* */
#define MPI_ERR_SPAWN       42  /* */
#define MPI_ERR_UNSUPPORTED_OPERATION 44       /* */
#define MPI_ERR_WIN         45  /* */

#define MPI_ERR_LASTCODE    0x3FFFFFFF  /* Last error code */


/* These are only guesses; make sure you change them in mpif.h as well */
#define MPI_MAX_PROCESSOR_NAME 256
#define MPI_MAX_ERROR_STRING   512
#define MPI_MAX_NAME_STRING     63      /* How long a name do you need ? */

typedef long MPI_Aint;

/* Upper bound on the overhead in bsend for each message buffer */
#define MPI_BSEND_OVERHEAD 512

/* Datatypes */
typedef int MPI_Datatype;

//--- Add by Chaney, 6/17/02
//---
#define MPI_BOTTOM      (void *)0
#define MPI_SUCCESS          0  /* Successful return code */
#define MPI_Primitive_Type   64

typedef struct _USF_MPI_struct_type
{
```

```
  int idx;
  int count;
  int *block_lengths;
  char *buf;
  MPI_Aint *disp;
  MPI_Datatype *typelist;
  struct _USF_MPI_struct_type *next;
}
_USF_MPI_struct_type;

// User define function
typedef void (MPI_User_function) (void *, void *, int *, MPI_Datatype *);
//---

#define MPI_CHAR           ((MPI_Datatype)1)
#define MPI_SHORT          ((MPI_Datatype)2)
#define MPI_INT            ((MPI_Datatype)3)
#define MPI_LONG           ((MPI_Datatype)4)
#define MPI_UNSIGNED_CHAR  ((MPI_Datatype)5)
#define MPI_UNSIGNED_SHORT ((MPI_Datatype)6)
#define MPI_UNSIGNED       ((MPI_Datatype)7)
#define MPI_UNSIGNED_LONG  ((MPI_Datatype)8)
#define MPI_FLOAT          ((MPI_Datatype)9)
#define MPI_DOUBLE         ((MPI_Datatype)10)
#define MPI_LONG_DOUBLE    ((MPI_Datatype)11)
#define MPI_BYTE           ((MPI_Datatype)12)
#define MPI_LONG_LONG      ((MPI_Datatype)13)

#define MPI_PACKED         ((MPI_Datatype)14)
#define MPI_LB             ((MPI_Datatype)15)
#define MPI_UB             ((MPI_Datatype)16)

/*
   The layouts for the types MPI_DOUBLE_INT etc are simply
   struct {
       double var;
       int    loc;
   }
*/
#define MPI_FLOAT_INT         ((MPI_Datatype)17)
#define MPI_DOUBLE_INT        ((MPI_Datatype)18)
#define MPI_LONG_INT          ((MPI_Datatype)19)
#define MPI_SHORT_INT         ((MPI_Datatype)20)
#define MPI_2INT              ((MPI_Datatype)21)
#define MPI_LONG_DOUBLE_INT   ((MPI_Datatype)22)

/* Special Structs for MINLOC, MAXLOC operations */

typedef struct _USF_FLOAT_INT
{
  float fst;
  int sec;
}
_USF_FLOAT_INT;

typedef struct _USF_DOUBLE_INT
{
  double fst;
  int sec;
}
_USF_DOUBLE_INT;

typedef struct _USF_LONG_INT
```

```
{
  long fst;
  int sec;
}
_USF_LONG_INT;

typedef struct _USF_SHORT_INT
{
  short fst;
  int sec;
}
_USF_SHORT_INT;

typedef struct _USF_2INT
{
  int fst;
  int sec;
}
_USF_2INT;

typedef struct _USF_LONG_DOUBLE_INT
{
  long double fst;
  int sec;
}
_USF_LONG_DOUBLE_INT;


/* Collective operations */
typedef int MPI_Op;

#define MPI_MAX    (MPI_Op)(100)
#define MPI_MIN    (MPI_Op)(101)
#define MPI_SUM    (MPI_Op)(102)
#define MPI_PROD   (MPI_Op)(103)
#define MPI_LAND   (MPI_Op)(104)
#define MPI_BAND   (MPI_Op)(105)
#define MPI_LOR    (MPI_Op)(106)
#define MPI_BOR    (MPI_Op)(107)
#define MPI_LXOR   (MPI_Op)(108)
#define MPI_BXOR   (MPI_Op)(109)
#define MPI_MINLOC (MPI_Op)(110)
#define MPI_MAXLOC (MPI_Op)(111)



typedef struct MPI_Status
{
  int count;
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR;
}
MPI_Status;

typedef struct MPI_Request_Int
{
  int active;                   /* 0 if not active, 1 if active, 2 if persistent act */
  int done;                     /* Operation is complete */
  int type;                     /* 0 recv, 1 send, 2 probe */
  int blocking;                 /* 0 non-blocking, 1 blocking */
  int posted;                   /* 0 post not sent, 1 post sent */
```

```
  void *buf;                    /* The message buffer */
  int size;                     /* The size of the buffer */
  int tag;                      /* The tag of the message */
  int peer;                     /* The other peer of communication */
  int comm;                     /* The communicator of the message */
  int multiple;                 /* 1 if multiple chunks, 0 if one chunk */
  int nchunk;                   /* Number of chunks sent or received */
  //Add by Chaney 2002, used for derives datatype */
  //---
  _USF_MPI_struct_type *stype;  /* Pointer to derived datatype */
  int dcount;                   /* Counter used for send/receive */
  //---
  struct _USF_request_node *node;       /* Back pointer for easy removal */
  int end_of_transmission;      /* to sychronize command thread and main thread add by Qing Huang */
}
MPI_Request_Int;

typedef MPI_Request_Int *MPI_Request;

typedef struct _USF_request_node
{
  struct _USF_request_node *next;       /* required for the link list */
  struct _USF_request_node *back;       /* required for the link list */
  MPI_Request_Int *req;
}
_USF_request_node;

typedef struct _USF_sysreq_node
{
  struct _USF_sysreq_node *next;        /* required for hash table */
  int tag;                      /* the tag of the message */
  int source;                   /* the source of the message */
  int type;                     /* 0 recv request, 1 send request */
  int comm;                     /* The communicator */
  int size;                     /* The size of the message */
#ifdef _USF_MPI_GM_
  void *in_buf;                 // for remote direct memory access, added by Qing Huang
#endif
}
_USF_sysreq_node;



#define _USF_HEADER_TYPE 0

#define _USF_UNIX_SOC 0
#define _USF_TCP_SOC  1

/* Communicators */
typedef int MPI_Comm;
#define MPI_COMM_WORLD 91
#define MPI_COMM_SELF  92

#define MPI_ANY_SOURCE  (-2)
#define MPI_ROOT        (-3)
#define MPI_ANY_TAG     (-1)

//#define MPI_REQUEST_NULL NULL

#define ER_INCORRECT   1
#define ER_FILENAME    2
#define ER_NUMP        3
```

```
#define ER_NEMACS      4
#define ER_HOSTNM      5
#define ER_MACFILE     6
#define ERF_SOCKETINIT 30
#define ERF_BINDINIT   31
#define ERF_ACCEPT     32
#define ERF_RECV       33
#define ERF_SEND       34
#define ERF_GETHE      35
#define ERF_CONNECT    36
#define ERF_SETSOCKOPT 37
#define ERF_LISTEN     38
#define ERF_SELECT     39
#define ERF_PTHREAD    40

/* Message Packet
   args header
   0 - source rank
   1 - destination rank
   2 - tag
   3 - communicator
   4 - size
   5 - type:
     0) Recv post
     1) Send post
     2) First chunk of synch communication
     3) Non terminal, non first chunk of synch communication
     4) Terminal chunk of synch communication
     5) Acknowledgement of chunk
     6) Short message
     7) Communication termination message

*/

/*
#define _USF_SOURCE 0
#define _USF_DEST   1
#define _USF_TAG    2
#define _USF_COMM   3
#define _USF_SIZE   4
#define _USF_TYPE   5
*/

typedef struct _USF_message_header
{
  int source;
  int dest;
  int tag;
  int comm;
  int size;
  unsigned int send_seq_num;
  int type;
}
_USF_message_header;

typedef struct _USF_conn_struct
{
  int sd;                        /* the socket descriptor used for comm */
  char *hostname;                /* the hostname of the peer - Tcp sockets */
  int port;                      /* the port number of the peer - Tcp sockets */
  char *path;                    /* the path of the peer - unix sockets */
  int type;                      /* UNIX_SOC or TCP_SOC */
  int port_id;                   /* the port number of the processor - GM port */
```

```
  int node_id;                  /* GM node */
  int next_msg;                 /* The type of next MSG to recieve. */
  int args[7];                  /* The header that was received before. */
  _USF_message_header hdr;      /* The header that was received before. */
}
_USF_conn_struct;


#define _USF_RECVPOST 0
#define _USF_SENDPOST 1
#define _USF_FCHUNK   2
#define _USF_MCHUNK   3
#define _USF_LCHUNK   4
#define _USF_ACKCHUNK 5
#define _USF_SHORTMES 6
#define _USF_COMMTERM 7
#define _USF_ACK_GMSEND 18
//Added by Chaney 2002, used for derived datatype
//---
#define _USF_DD_SINGLE_COND      8
#define _USF_DD_SINGLE_END       9
#define _USF_DD_MULTI_COND_COND 10      //chunk of mid element of structure
#define _USF_DD_MULTI_COND_END  11      //chunk of the last element of structure
#define _USF_DD_MULTI_END_COND  12      //the last chunk of mid element of structure
#define _USF_DD_MULTI_END_END   13      //the last chunk of the last element of structure
#define _USF_DD_ACK             14
//---


#define _USF_RECEIVE 0
#define _USF_SEND 1
#define _USF_PROBE 2

typedef struct _USF_rank_type
{
  int rank;
  int comm;
  int nump;
  int group;
  int old_comm;
}
_USF_rank_type;

typedef struct _USF_group_type
{
  int *rank_global;
  int *rank_old;
  int group_no_local;
  int group_no_old;
  int group_size;
}
_USF_group_type;

typedef struct _USF_header
{
  int source;
  int dest;
  int tag;
  int comm;
  int size;
  int type;
  void *in_buf;
  unsigned int send_seq_num;
```

```
}
_USF_header;

extern int _USF_group_max;
extern int _USF_group_ref;
extern int _USF_num_of_group;
extern int _USF_comm_max;
extern int _USF_comm_ref;
extern int _USF_num_of_comm;
extern struct _USF_group_type *_USF_group_list;
extern struct _USF_rank_type *_USF_comm_list;


extern pthread_t _USF_comm_deamon;
extern int *_USF_finalize_list;
extern pthread_mutex_t _USF_mut_sysreq; /* sys_request link list */
extern pthread_mutex_t _USF_mut_req;    /* MPI_Request link list */
extern pthread_mutex_t _USF_mut_block;  /* Blocking communication */
extern pthread_cond_t _USF_cond_block;  /* Blocking Communication */
extern int _USF_s_pipe[2];      /* 0 read, 1 write */
// - Bengin - by Qing Huang
extern int _USF_gm_pipe[2];     /* 0 read, 1 write */// by Qing Huang
extern struct gm_port *local_port;
// - End - by Qing Huang

extern MPI_Request_Int *_USF_synchr_list[_USF_MAXPROCESSORS];
extern _USF_sysreq_node *_USF_sysreq_hashtable[_USF_MAXPROCESSORS];
extern _USF_request_node *_USF_request_hd, *_USF_request_tl;
                                              /* Head and tail of the list */
extern _USF_conn_struct *_USF_conn_list;
extern pool_t *_USF_mpi_request_int_pool;
extern pool_t *_USF_request_node_pool;
extern pool_t *_USF_sysreq_node_pool;

// Assume the 0th rank is the one that initiated the run.
// Assume returns 0 in success
int MPI_Init (int *argc, char ***argv);
int MPI_Finalize (void);
int MPI_Recv (void *buf, int count, MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm, MPI_Status * status);
int MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm);
int MPI_Ssend (void *buf, int count, MPI_Datatype datatype, int dest,
               int tag, MPI_Comm comm);
int MPI_Rsend (void *buf, int count, MPI_Datatype datatype, int dest,
               int tag, MPI_Comm comm);
int MPI_Comm_rank (MPI_Comm comm, int *rank);
int MPI_Comm_size (MPI_Comm comm, int *size);
int MPI_Get_count (MPI_Status * status, MPI_Datatype datatype, int *count);
int MPI_Irecv (void *buf, int count, MPI_Datatype datatype, int source,
               int tag, MPI_Comm comm, MPI_Request * request);
int MPI_Wait (MPI_Request * request, MPI_Status * status);
int MPI_Test (MPI_Request * request, int *flag, MPI_Status * status);
int MPI_Isend (void *buf, int count, MPI_Datatype datatype, int dest,
               int tag, MPI_Comm comm, MPI_Request * request);
int MPI_Irsend (void *buf, int count, MPI_Datatype datatype, int dest,
                int tag, MPI_Comm comm, MPI_Request * request);
int MPI_Issend (void *buf, int count, MPI_Datatype datatype, int dest,
                int tag, MPI_Comm comm, MPI_Request * request);
int MPI_Iprobe (int source, int tag, MPI_Comm comm,
                int *flag, MPI_Status * status);
int MPI_Probe (int source, int tag, MPI_Comm comm, MPI_Status * status);
int MPI_Waitany (int count, MPI_Request * array_of_requests, int *index,
```

```
                 MPI_Status * status);
int MPI_Testany (int count, MPI_Request * array_of_requests, int *index,
                 int *flag, MPI_Status * status);
int MPI_Waitall (int count, MPI_Request * array_of_requests,
                 MPI_Status * array_of_statuses);
int MPI_Testall (int count, MPI_Request * array_of_requests, int *flag,
                 MPI_Status * array_of_statuses);
int MPI_Send_init (void *buf, int count, MPI_Datatype datatype, int dest,
                   int tag, MPI_Comm comm, MPI_Request * request);
int MPI_Rsend_init (void *buf, int count, MPI_Datatype datatype, int dest,
                    int tag, MPI_Comm comm, MPI_Request * request);
int MPI_Ssend_init (void *buf, int count, MPI_Datatype datatype, int dest,
                    int tag, MPI_Comm comm, MPI_Request * request);
int MPI_Recv_init (void *buf, int count, MPI_Datatype datatype, int source,
                   int tag, MPI_Comm comm, MPI_Request * request);
int MPI_Start (MPI_Request * request);
int MPI_Startall (int count, MPI_Request * array_of_requests);
int MPI_Sendrecv (void *sendbuf, int sendcount, MPI_Datatype sendtype,
                  int dest, int sendtag, void *recvbuf, int recvcount,
                  MPI_Datatype recvtype, int source, int recvtag,
                  MPI_Comm comm, MPI_Status * status);
int MPI_Sendrecv_replace (void *buf, int count, MPI_Datatype datatype,
                          int dest, int sendtag, int source, int recvtag,
                          MPI_Comm comm, MPI_Status * status);
int MPI_Barrier (MPI_Comm comm);
int MPI_Bcast (void *buffer, int count, MPI_Datatype datatype,
               int root, MPI_Comm comm);
int MPI_Gather (void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void *recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm);
int MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int root, MPI_Comm comm);
int MPI_Allgather (void *sendbuf, int sendcount, MPI_Datatype sendtype,
                   void *recvbuf, int recvcount, MPI_Datatype recvtype,
                   MPI_Comm comm);
int MPI_Alltoall (void *sendbuf, int sendcount, MPI_Datatype sendtype,
                  void *recvbuf, int recvcount, MPI_Datatype recvtype,
                  MPI_Comm comm);
int MPI_Reduce (void *, void *, int, MPI_Datatype, MPI_Op, int, MPI_Comm);

//--- Add by Chaney 2002
//--- MPI derived data type modual
int MPI_Type_struct (int, int *, MPI_Aint *, MPI_Datatype *, MPI_Datatype *);
int MPI_Address (void *, MPI_Aint *);
int MPI_Pack (void *pack_data, int in_count, MPI_Datatype datatype,
              void *buffer, int buffer_size, int *position, MPI_Comm comm);
int MPI_Unpack (void *buffer, int size, int *position, void *unpack_data,
                int count, MPI_Datatype datatype, MPI_Comm comm);
int MPI_Type_indexed (int count, int block_lengths[],
                      MPI_Aint displacements[], MPI_Datatype old_type,
                      MPI_Datatype * new_mpi_t);
int MPI_Type_contiguous (int count, MPI_Datatype old_type,
                         MPI_Datatype * new_mpi_t);
int MPI_Type_vector (int count, int block_length, int stride,
                     MPI_Datatype element_type, MPI_Datatype * new_mpi_t);
//--- MPI collective
int MPI_Scatterv (void *sendbuf, int *sendcount, int *displs,
                  MPI_Datatype sendtype, void *recvbuf, int recvcount,
                  MPI_Datatype recvtype, int root, MPI_Comm comm);
int MPI_Gatherv (void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int *recvcount, int *displs,
                 MPI_Datatype recvtype, int root, MPI_Comm comm);
```

```
int MPI_Op_create (MPI_User_function *, int, MPI_Op *);
int MPI_op_free (MPI_Op * op);
MPI_User_function *Get_User_function (MPI_Op op);
//---
void _USF_handle_recv_post (int dest, int source, int tag, int comm,
                            void *buf, int size, int multiple);
void _USF_handle_send_init (int dest, int source, int tag, int comm,
                            void *buf, int *size);
void _USF_handle_msg (int source);
void _USF_serve_mpi_requests (int rank, int nump);
void *_USF_comm_thread (void *arg);
_USF_sysreq_node *_USF_sysreq_lookup (int tag, int source, int type,
                                      int comm);

#ifdef _USF_MPI_GM_
void _USF_sysreq_insert (int tag, int source, int type, int comm, int size,
                         void *in_buf, _USF_sysreq_node * value);
#endif
#ifdef _USF_MPI_TCP_
void _USF_sysreq_insert (int tag, int source, int type, int comm, int size,
                         _USF_sysreq_node * value);

#endif

_USF_sysreq_node *_USF_sysreq_remove (int tag, int source, int type,
                                      int comm);
void _USF_sysreq_free ();
_USF_request_node *_USF_request_lookup (int tag, int source, int type,
                                        int comm);
_USF_request_node *_USF_request_lookup_m (int tag, int source, int type,
                                          int comm, int multiple);
_USF_request_node *_USF_request_lookup_p (int tag, int type, int comm,
                                          int posted);
void _USF_request_inshd (_USF_request_node * value);
void _USF_request_instl (_USF_request_node * value);
void _USF_request_remove (_USF_request_node * elem);
void _USF_request_free ();

// - Begin - by Qing Huang
int _USF_GM_send_all (int dest, int size, void *buf, char *err_mes);
//int _USF_GM_recv_all (int source, int size, void* buf, char* err_mes);
// - End - by Qing Huang
int _USF_TCP_send_all (int dest, int size, void *buf, char *err_mes);
int _USF_TCP_recv_all (int source, int size, void *buf, char *err_mes);

int _USF_Error (int wh);
int _USF_mpi_sizeof (MPI_Datatype ty);
int _USF_make_conn_list (char *my_name, int nump);

int _USF_num_bits (int n);
int _USF_make_bm (int n);

int _USF_solve_max (void *, void *, int, MPI_Datatype);
int _USF_solve_min (void *, void *, int, MPI_Datatype);
int _USF_solve_sum (void *, void *, int, MPI_Datatype);
int _USF_solve_prod (void *, void *, int, MPI_Datatype);
int _USF_solve_land (void *, void *, int, MPI_Datatype);
int _USF_solve_band (void *, void *, int, MPI_Datatype);
int _USF_solve_lor (void *, void *, int, MPI_Datatype);
int _USF_solve_bor (void *, void *, int, MPI_Datatype);
int _USF_solve_lxor (void *, void *, int, MPI_Datatype);
int _USF_solve_bxor (void *, void *, int, MPI_Datatype);
int _USF_solve_minloc (void *, void *, int, MPI_Datatype);
```

```
int _USF_solve_maxloc (void *, void *, int, MPI_Datatype);


// Modified by Qing Huang
int MPI_Type_size (MPI_Datatype, int *);
double MPI_Wtime ();
int MPI_Reduce_scatter (void *, void *, int *, MPI_Datatype, MPI_Op,
                        MPI_Comm);
int MPI_Allgatherv (void *, int, MPI_Datatype, void *, int *, int *,
                    MPI_Datatype, MPI_Comm);
int MPI_Comm_group (MPI_Comm, MPI_Group *);
int MPI_Abort (MPI_Comm, int);
int MPI_Group_translate_ranks (MPI_Group, int, int *, MPI_Group, int *);
int MPI_Comm_free (MPI_Comm *);
int MPI_Comm_split (MPI_Comm, int, int, MPI_Comm *);
int MPI_Group_incl (MPI_Group old_group,       // in
                    int new_group_size, // in
                    int ranks_in_old_group[],   // in
                    MPI_Group * new_group       // in
  );

int MPI_Comm_create (MPI_Comm old_comm, // in
                     MPI_Group new_group,       // in
                     MPI_Comm * new_comm        // out
  );
// End of Modification QH

#endif /* _MPI_H */
```

B.2 debug.h

```
/*
 * debug.h
 *
 * macros for debugging
 *
 * 2000-07-27 Alex Fedosov
 * 2001-10-23 Gregory D. Benson
 *   Name change: dbgmsg to DBG (shorter and easier to see in code)
 */

// #define DEBUG

#ifndef __DEBUG_H__
#define __DEBUG_H__

#include <stdio.h>

#ifdef DEBUG
#define DBG(x) { \
printf x; \
fflush (stdout); \
}
#else
#define DBG(x)
#endif

#endif
```

B.3 gm_thread.h

```
#ifndef __GM_THREAD_H__
#define __GM_THREAD_H__

#define GM_MAX_RECV_TOKENS 10
#define GM_PKT_SIZE 30

extern int _USF_gm_pipe[2];
extern struct gm_port *local_port;

void _USF_GM_recv_all (int source, int size, void *buf, char *msg);
void get_rank_by_gm_id (int id, int port_id, int *rank);
void *gm_thread ();
void gm_thread_end ();

#endif
```

B.4 pool.h

```
/*
 * pool.h
 *
 * pool interface
 *
 * Alex Fedosov
 * Shaun Padden
 * 2001-10-23 Gregory D. Benson
 *   name change: Pool to pool_t
 *   added user-supplied init and destroy function callbacks
 *   removed lock from pool_t struct
 */

typedef int (*pool_func) (void *);
typedef void *(*pool_malloc) (size_t);
typedef void (*pool_free) (void *);

typedef struct pool_item
{
  struct pool_item *next;
}
pool_item;

typedef struct pool_t
{
  pool_item *head;              /* first element in pool */

  int available;                /* number of available elements */
  int size;                     /* current size, including allocated
                                   and avilable elements */
  int capacity;                 /* maximum size */
  int free_limit;               /* max number of available elements
                                   at any given time. extra ones will
                                   be discarded */
  int alloc_incr;               /* increments in which to allocate
                                   more elements */
  unsigned int item_size;       /* size of each element in bytes */
  pool_malloc malloc;           /* user-supplied malloc */
  pool_free free;               /* user-supplied free */
  pool_func init;               /* user-supplied init */
  pool_func destroy;            /* user-supplied destroy */
```

```
}
pool_t;

/*
   This function initializes the pool. Parameters are as follows:

   initial_size: number of elements initially available.
    must be greater than 0.

   max_size: maximum number of elements a pool can contain,
   including both available and allocated items.
   The pool will not grow above this size.
      if equal to -1, no limit on pool size.
    if equal to 0, returns error.

   alloc_incr: number of additional elements that will be allocated
    once no elements are available.
if equal to -1, initial_size will be used as increment.
    if equal to 0, pool size will never be increased,
    regardless of max_size.

   free_threshold: maximum number of available elements at any given
    time; all extra ones will be deallocated.
    if equal to -1, no limit on available elements.
    if equal to 0, all elements that become available will be immediately
    discarded. (Does not apply to elements available initially.)

   elem_size: size of the element that the pool will contain
    in bytes.

Return value: NULL on error, pointer to a new pool on success.

*/

pool_t *pool_fill (int initial_size, int max_size, int alloc_incr,
                   int free_threshold, unsigned int item_size,
                   pool_malloc u_malloc, pool_free u_free,
                   pool_func u_init, pool_func u_destroy);

/* get an available element from the specified pool */
pool_item *pool_take (pool_t * p);

/* return a no-longer-needed item back to the pool */
void pool_return (pool_t * p, pool_item * e);

/* destroy the pool, deallocating all elements currently in the pool.
   Thus for proper memory management all elements taken from the pool
   should be returned before calling this function. */
void pool_drain (pool_t * p);

/* return nonzero value if pool is empty, zero otherwise */
int pool_empty (pool_t * p);
```

B.5 allgather.c

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<unistd.h>
#include<pthread.h>
#include"errno.h"
```

```
#include"mpi.h"

// tag -14 is used for allgather.
int
MPI_Allgather (void *sendbuf, int sendcount, MPI_Datatype sendtype,
               void *recvbuf, int recvcount, MPI_Datatype recvtype,
               MPI_Comm comm)
{
  int nump, rank, s, i;
  int bm, nb, cb;
  int partner, cont;
  int scount, rcount, sendind, recvind;
  MPI_Status status;

  MPI_Comm_size (comm, &nump);
  MPI_Comm_rank (comm, &rank);

  if (rank == -1) {
    printf ("Wrong communicator in calling MPI_Allgather\n");
    fflush (stdout);
    return 1;
  }


  s = _USF_mpi_sizeof (recvtype);
  nb = _USF_num_bits (nump - 1);
  bm = _USF_make_bm (nb);


  memcpy ((recvbuf + (rank * recvcount * s)), sendbuf, recvcount * s);
  cb = 1;
  for (cb = 1; cb <= bm; cb <<= 1) {
    partner = rank ^ cb;
    if (((rank | (cb - 1)) >= (nump - 1))
        && ((cb - (bm * 2 - nump)) > 0)) {
      scount = sendcount * (cb - (bm * 2 - nump));
    } else {
      scount = sendcount * cb;
    }
    if (((partner | (cb - 1)) >= (nump - 1))
        && ((cb - (bm * 2 - nump)) > 0)) {
      rcount = recvcount * (cb - (bm * 2 - nump));
    } else {
      rcount = recvcount * cb;
    }

    sendind = (rank & (~(cb - 1))) * (recvcount * s);
    recvind = (partner & (~(cb - 1))) * (recvcount * s);
    if (partner >= nump) {      // do a recv from last processor!
      if ((rank != (nump - 1))
          && ((partner & (~(cb - 1))) < nump)) {
        MPI_Recv (recvbuf + recvind, rcount, recvtype,
                  (nump - 1), -14, comm, &status);
      }
      continue;
    }
    MPI_Sendrecv (recvbuf + sendind, scount, sendtype, partner, -14,
                  recvbuf + recvind, rcount, recvtype, partner, -14,
                  comm, &status);
    // Now the last processor serves for the missing ones
    if (rank == (nump - 1)) {
      for (i = rank + 1; i < bm * 2; i++) {
        partner = i ^ cb;
```

```
        if (partner >= rank)
          continue;
        MPI_Send (recvbuf + sendind, scount, sendtype, partner, -14, comm);
      }
    }
  }
  return 0;
}

// tag -19 is used for allgather.
int
MPI_Allgatherv (void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void *recvbuf, int *recvcount, int *displs,
                MPI_Datatype recvtype, MPI_Comm comm)
{
  int nump, rank, s, i;
  int bm, nb, cb;
  int partner, cont;
  int scount, rcount, sendind, recvind;
  MPI_Status status;

  MPI_Comm_size (comm, &nump);
  MPI_Comm_rank (comm, &rank);

  if (rank == -1) {
    printf ("Wrong communicator in calling MPI_Allgather\n");
    fflush (stdout);
    return 1;
  }


  s = _USF_mpi_sizeof (recvtype);
  memcpy (recvbuf + displs[rank], sendbuf, sendcount);
  for (i = 0; i < nump; i++) {
    MPI_Bcast (recvbuf + displs[i], recvcount[i], recvtype, i, comm);
  }
  return MPI_SUCCESS;
}
```

B.6 allreduce.c

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<unistd.h>
#include<pthread.h>
#include"errno.h"
#include"mpi.h"

// tag -16 is used for allgather.
int
MPI_Allreduce (void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
{
  int nump, rank, i;
  int bm, nb, cb;
  int partner;
  void *buf1, *buf2;
  MPI_Status status;
  int *z;
  MPI_User_function *userop;
```

```
MPI_Comm_size (comm, &nump);
MPI_Comm_rank (comm, &rank);

if (rank == -1) {
  printf ("Wrong communicator in calling MPI_Allreduce\n");
  fflush (stdout);
  return 1;
}

nb = _USF_num_bits (nump - 1);
bm = _USF_make_bm (nb);

cb = 1;
for (cb = 1; cb <= bm; cb <<= 1) {

  if (rank == (nump - 1)) {
    for (i = rank + 1; i < bm * 2; i++) {
      partner = i ^ cb;
      if ((partner >= rank) || ((i & (~(cb - 1))) >= nump))
        continue;
      MPI_Send (sendbuf, count, datatype, partner, -16, comm);
    }
  }

  partner = rank ^ cb;
  if (partner >= nump) {      // do a recv from last processor!
    if ((rank != (nump - 1))
        && ((partner & (~(cb - 1))) < nump)) {
      MPI_Recv (recvbuf, count, datatype, (nump - 1), -16, comm, &status);
    } else
      continue;
  } else {
    MPI_Sendrecv (sendbuf, count, datatype, partner, -16,
                  recvbuf, count, datatype, partner, -16, comm, &status);
  }

  if (cb == bm) {
    buf1 = recvbuf;
    buf2 = sendbuf;
  } else {
    buf1 = sendbuf;
    buf2 = recvbuf;
  }
  switch (op) {
  case MPI_MAX:
    _USF_solve_max (buf1, buf2, count, datatype);
    break;
  case MPI_MIN:
    _USF_solve_min (buf1, buf2, count, datatype);
    break;
  case MPI_SUM:
    _USF_solve_sum (buf1, buf2, count, datatype);
    break;
  case MPI_PROD:
    _USF_solve_prod (buf1, buf2, count, datatype);
    break;
  case MPI_LAND:
    _USF_solve_land (buf1, buf2, count, datatype);
    break;
  case MPI_BAND:
    _USF_solve_band (buf1, buf2, count, datatype);
    break;
```

```
    case MPI_LOR:
      _USF_solve_lor (buf1, buf2, count, datatype);
      break;
    case MPI_BOR:
      _USF_solve_bor (buf1, buf2, count, datatype);
      break;
    case MPI_LXOR:
      _USF_solve_lxor (buf1, buf2, count, datatype);
      break;
    case MPI_BXOR:
      _USF_solve_bxor (buf1, buf2, count, datatype);
      break;
    case MPI_MINLOC:
      _USF_solve_2loc (buf1, buf2, count, datatype, 1);
      break;
    case MPI_MAXLOC:
      _USF_solve_2loc (buf1, buf2, count, datatype, 0);
      break;
    default:
      if (op >= 120) {
        //This is a user defined opreation
        userop = Get_User_function (op);
        if (userop == NULL) {
          printf ("This operation isn't defined!\n");
          exit (1);
        } else {
          userop (buf2, buf1, &count, &datatype);
        }
      } else {
        printf ("This operation isn't defined!\n");
        exit (1);
      }
    }
  }
  return 0;
}
```

B.7 alltoall.c

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<unistd.h>
#include"errno.h"
#include"mpi.h"

//Tag 21 is used for Alltoall
int
MPI_Alltoall (void *sendbuf, int sendcount, MPI_Datatype sendtype,
              void *recvbuf, int recvcount, MPI_Datatype recvtype,
              MPI_Comm comm)
{
  int nump, rank, i, j, rsize, ssize, recvind;
  MPI_Request *send_req;
  MPI_Request *recv_req;
  MPI_Status *status;

  MPI_Comm_size (comm, &nump);
  MPI_Comm_rank (comm, &rank);

  if (rank == -1) {
```

```
    printf ("Wrong communicator in calling MPI_Allgather\n");
    fflush (stdout);
    return 1;
  }

  rsize = _USF_mpi_sizeof (recvtype);
  ssize = _USF_mpi_sizeof (sendtype);
  recv_req = (MPI_Request *) malloc ((nump - 1) * sizeof (MPI_Request));
  send_req = (MPI_Request *) malloc ((nump - 1) * sizeof (MPI_Request));
  status = (MPI_Status *) malloc ((nump - 1) * sizeof (MPI_Status));

  // First set the receives, then send
  for (i = 0, j = 0; i < nump; i++, j++) {
    if (rank == i) {
      j--;
      continue;
    }
    MPI_Irecv ((recvbuf + (i * recvcount * rsize)), recvcount,
               recvtype, i, -21, comm, &(recv_req[j]));

  }
  for (i = 0, j = 0; i < nump; i++, j++) {
    if (rank == i) {
      j--;
      continue;
    }
    MPI_Isend ((sendbuf + (i * sendcount * ssize)), sendcount,
               sendtype, i, -21, comm, &(send_req[j]));

  }
  if (recvbuf != sendbuf) {
    memcpy ((recvbuf + (rank * recvcount * rsize)),
            (sendbuf + (rank * sendcount * ssize)), sendcount * ssize);
  }
  MPI_Waitall ((nump - 1), recv_req, status);
  MPI_Waitall ((nump - 1), send_req, status);
  return 0;
}
```

B.8 barrier.c

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<unistd.h>
#include<pthread.h>
#include"errno.h"
#include"mpi.h"


int
_USF_num_bits (int n)
{
  int i = 0;

  if (n == 0)
    return 1;
  for (; n > 0; i++) {
    n >>= 1;

  }
```

```
  return i;
}

int
_USF_make_bm (int n)
{
  int i, rv = 1;
  for (i = 1; i < n; i++) {
    rv <<= 1;
  }
  return rv;

}

// tag -10 is used for barrier.
int
MPI_Barrier (MPI_Comm comm)
{
  int nump, rank, my_rank, mes;
  int bm, nb, cb;
  int partner, cont;
  MPI_Status status;

  MPI_Comm_size (comm, &nump);
  MPI_Comm_rank (comm, &rank);
  my_rank = rank;

  if (rank == -1) {
    printf ("Wrong communicator in calling MPI_Barrier\n");
    fflush (stdout);
    return 1;
  }

  nb = _USF_num_bits (nump - 1);
  bm = _USF_make_bm (nb);

  cb = bm;
  while (cb != 0) {
    partner = rank ^ cb;
    cb >>= 1;
    if (rank < partner) {
      if (partner >= nump) {
        continue;
      }
      MPI_Recv (&mes, 1, MPI_INT, partner, -10, comm, &status);
    } else {
      MPI_Send (&mes, 1, MPI_INT, partner, -10, comm);
      break;
    }
  }

  // May just use Bcast instead, to get rid of code duplication
  cb = 1;
  while (cb <= bm) {
    partner = rank ^ cb;
    cb <<= 1;
    fflush (stdout);
    if (_USF_num_bits (rank) > cb) {
      continue;
    }
    if (rank > partner) {
      MPI_Recv (&mes, 1, MPI_INT, partner, -10, comm, &status);
    } else {
```

```
      if (partner >= nump) {
        continue;
      }
      MPI_Send (&mes, 1, MPI_INT, partner, -10, comm);
    }
  }

  return 0;
}
```

B.9 bcast.c

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<unistd.h>
#include<pthread.h>
#include"errno.h"
#include"mpi.h"

// tag -11 is used for bcast.
int
MPI_Bcast (void *buffer, int count, MPI_Datatype datatype,
           int root, MPI_Comm comm)
{
  int nump, rank, mes, my_rank, i;
  int bm, nb, cb;
  int partner, cont;
  MPI_Status status;
  int orrank;

  MPI_Comm_size (comm, &nump);
  MPI_Comm_rank (comm, &rank);

  if (rank == -1) {
    printf ("Wrong communicator in calling MPI_Bcast\n");
    fflush (stdout);
    return 1;
  }

  my_rank = rank;
  nb = _USF_num_bits (nump - 1);
  bm = _USF_make_bm (nb);

  cb = 1;
  orrank = rank;
  if (rank == 0)
    rank = root;
  else if (rank == root)
    rank = 0;

  while (cb <= bm) {
    partner = rank ^ cb;
    cb <<= 1;

    if (_USF_num_bits (rank) > cb)
      continue;
    if (rank > partner) {
      if (partner == 0)
        partner = root;
      else if (partner == root)
```

```
        partner = 0;
      MPI_Recv (buffer, count, datatype, partner, -11, comm, &status);
    } else {
      if (partner == 0)
        partner = root;
      else if (partner == root)
        partner = 0;
      if (partner >= nump)
        continue;
      MPI_Send (buffer, count, datatype, partner, -11, comm);
    }
  }

  return 0;
}
```

B.10 comm_rank.c

```
#include"mpi.h"
#include <stdlib.h>
#include <stdio.h>

inline int
MPI_Comm_rank (MPI_Comm comm, int *rank)
{
  int i;

  for (i = 0; i < _USF_num_of_comm; i++) {
    if (_USF_comm_list[i].comm == comm) {
      *rank = _USF_comm_list[i].rank;
      return 0;
    }
  }
  *rank = -1;
  return -1;
}
```

B.11 comm_size.c

```
#include "mpi.h"
#include "stdio.h"

inline int
MPI_Comm_size (MPI_Comm comm, int *size)
{
  int i;
  for (i = 0; i < _USF_num_of_comm; i++) {
    if (_USF_comm_list[i].comm == comm) {
      *size = _USF_comm_list[i].nump;
      return 0;
    }
  }

  printf ("Error, this communicator doesn't exist\n");
  fflush (stdout);
  *size = -1;
  return -1;
}
```

B.12 comm_thread.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <pthread.h>
#include "errno.h"
#include "mpi.h"

#ifdef _USF_MPI_TCP_
int blocking_for_big_message[_USF_MAXPROCESSORS] = { 0 };
#define DB(a,b,c)
#ifdef DEBUG
#undef DB
#define DB(a,b,c) printf (a,b,c); fflush (stdout);
#endif

void
_USF_handle_recv_post (int dest, int source, int tag, int comm, void *buf,
                       int size, int multiple)
{
  _USF_message_header hdr;      /* The header for sends */
  int send_size;                /* The number of bytes sent */
  int recv_size;                /* The number of bytes sent */

  hdr.source = source;
  hdr.dest = dest;
  hdr.tag = tag;
  hdr.comm = comm;
  if (!(multiple)) {            /* Ready mode short message */
    /* Send send init */
    hdr.size = size;
    hdr.type = _USF_SHORTMES;
    _USF_send_all (dest, sizeof (_USF_message_header), &hdr,
                   "Header isn't transmitted in handle_recv_post");
    _USF_send_all (dest, size, buf, "Can't sent ready mode message");
  } else {                      /* First of multiple chunks */
    /* Send send init */
    hdr.size = _USF_MAXDATASIZE;
    hdr.type = _USF_FCHUNK;
    _USF_send_all (dest, sizeof (_USF_message_header), &hdr,
                   "Header isn't transmitted in handle_recv_post");
    _USF_send_all (dest, _USF_MAXDATASIZE, buf,
                   "Can't sent ready mode message");
    DB ("rank %d, handle recv post multip2 %d\n", source, multiple);
  }
  /* updatie mpi_request in handle_msg */
  return;
}

void
_USF_serve_sysreq (int source)
{
  int i;                        /* Loop variable */
  int nump;                     /* # of procs in COMM_WORLD */
  int rank;                     /* the rank in COMM_WORLD */
  _USF_message_header hdr;      /* The message header */
  _USF_request_node *req1;      /* The request node used for lookup */
  _USF_sysreq_node *node;       /* The sysreq node used to search */
  MPI_Comm_size (MPI_COMM_WORLD, &nump);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
```

```
  /* Search for a recv_post in sysreq list that matches a synch
     (multiple chunks) send from the current processor */
  /* This function uses the internal data structure ofd sysrequest
     data sturucture */
  DB ("rank %d, serve sysreq\n", rank, 1);
  pthread_mutex_lock (&_USF_mut_sysreq);
  source = source % _USF_MAXPROCESSORS;
  for (node = _USF_sysreq_hashtable[source]; node != NULL; node = node->next) {
    DB ("rank %d, forloop\n", rank, 1);
    if ((source == node->source) && (node->type == _USF_RECEIVE)) {
      break;
    }
  }

  if (node == NULL) {
    pthread_mutex_unlock (&_USF_mut_sysreq);
    return;
  }
  pthread_mutex_lock (&_USF_mut_req);
  req1 = _USF_request_lookup (node->tag, source, _USF_SEND, node->comm);


  if (req1 == NULL) {
    pthread_mutex_unlock (&_USF_mut_sysreq);
    pthread_mutex_unlock (&_USF_mut_req);
    return;
  }
  /* Found the matching long send */
  hdr.source = rank;
  hdr.dest = source;
  hdr.tag = node->tag;
  hdr.comm = node->comm;
  /* Remove the sysreq */
  pool_return (_USF_sysreq_node_pool, (pool_item *)
               _USF_sysreq_remove (node->tag, source, _USF_RECEIVE,
                                   node->comm));
  pthread_mutex_unlock (&_USF_mut_sysreq);
  /* Start communication */
  DB ("rank %d, start to handle recv post in servsys\n", rank, 1);
  _USF_handle_recv_post (source, rank, req1->req->tag, req1->req->comm,
                         req1->req->buf, req1->req->size,
                         req1->req->multiple);
  blocking_for_big_message[source]--;
  send_post_account[req1->req->peer]++;
  if (req1->req->multiple) {
    req1->req->nchunk = 1;
    _USF_synchr_list[source] = req1->req;
  } else {
    req1->req->done = 1;
    /* Check if it was blocking !!! */
    pthread_mutex_lock (&_USF_mut_block);
    if (req1->req->blocking) {
      DB ("rank %d, wake up main\n", rank, 1);
      req1->req->end_of_transmission = 1;
      pthread_cond_signal (&_USF_cond_block);
    } else
      req1->req->end_of_transmission = 1;
    pthread_mutex_unlock (&_USF_mut_block);
  }

  pthread_mutex_unlock (&_USF_mut_req);
}
```

```
void
_USF_handle_msg (int source)
{
  int i;                        /* Loop variable */
  int nump;                     /* # of procs in COMM_WORLD */
  int rank;                     /* the rank in COMM_WORLD */
  int recv_size;                /* The size of received packet */
  _USF_message_header hdr;      /* The message header */
  int size;                     /* Used for ready mode receive */
  _USF_request_node *req1, *req_probe;  /* The request node used for lookup */
  _USF_sysreq_node *sysreq1;    /* The sysreq node used to insert a node */
  int isongcomm = 0;            /* Is there an ongoiing communication */
  int header_msg = 0;

  MPI_Comm_size (MPI_COMM_WORLD, &nump);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);


  //Read data from source node
  // Modified by Chaney 10/3
  if (_USF_conn_list[source].next_msg == _USF_HEADER_TYPE) {
    _USF_recv_all (source, sizeof (_USF_message_header), &hdr,
                   "Header isn't received for handle_msg");
    header_msg = 1;
  } else {
    memcpy (&hdr, &(_USF_conn_list[source].hdr),
            sizeof (_USF_message_header));
  }
  // End of change by Chaney
#ifdef DEBUG
  printf ("rank %d, Message: so %d de %d ta %d co %d si %d ty %d\n",
          rank, hdr.source, hdr.dest, hdr.tag, hdr.comm, hdr.size, hdr.type);
  fflush (stdout);
#endif
  switch (hdr.type) {
  case _USF_RECVPOST:
    /* These are removed from sysreq by MPI_(I)SEND calls */
    /* Look for a send request in request list */
    DB ("rank %d, Get a recv_post\n", rank, 1);
    pthread_mutex_lock (&_USF_mut_req);
    req1 = _USF_request_lookup (hdr.tag, hdr.source, _USF_SEND, hdr.comm);
    if (req1 != NULL) {
      DB ("rank %d, NONNULL \n", rank, 1);
      if (_USF_synchr_list[source] != NULL)
        isongcomm = 1;
    }
    DB ("rank %d, isong %d\n", rank, isongcomm);
    if ((req1 == NULL) || isongcomm) {
      sysreq1 = (_USF_sysreq_node *) pool_take (_USF_sysreq_node_pool);
      if (sysreq1 == NULL)
        DB ("rank %d Memory error", rank, 1);
      pthread_mutex_lock (&_USF_mut_sysreq);
      _USF_sysreq_insert (hdr.tag, hdr.source,
                          _USF_RECEIVE, hdr.comm, hdr.size, sysreq1);
      pthread_mutex_unlock (&_USF_mut_sysreq);
    } else {
      _USF_handle_recv_post (hdr.source,
                             hdr.dest, hdr.tag, hdr.comm, req1->req->buf,
                             req1->req->size, req1->req->multiple);
      /* update mpi_request */
      if (req1->req->posted == 0)
        ++send_post_account[hdr.source];
      if (req1->req->multiple) {
```

```
      req1->req->nchunk = 1;
      _USF_synchr_list[source] = req1->req;
    } else {
      req1->req->done = 1;
      /* Check if it was blocking !!! */
      pthread_mutex_lock (&_USF_mut_block);
      if (req1->req->blocking) {
        DB ("rank %d, wake up main\n", rank, 1);
        req1->req->end_of_transmission = 1;
        pthread_cond_signal (&_USF_cond_block);
      } else
        req1->req->end_of_transmission = 1;
      pthread_mutex_unlock (&_USF_mut_block);
    }
  }
  pthread_mutex_unlock (&_USF_mut_req);
  break;
case _USF_SENDPOST:
  /* These are removed from sysreq by MPI_(I)RECV calls */
  /* Send a recv post if unsent recv post in mpi req list */
  DB ("rank %d Get a send_post\n", rank, 1);

  if (hdr.send_seq_num <= recv_post_account[hdr.source])
    break;

  pthread_mutex_lock (&_USF_mut_req);

  /* Continue with the operation */
  req1 = _USF_request_lookup_p (hdr.tag, _USF_RECEIVE, hdr.comm, 0);
  if (req1 == NULL) {         /* put in the sys req queue */
    //Received a send-post but there is no recv req in system queue
    DB ("rank %d SENDPOST NULL, put in sysreq Q\n", rank, 1);
    sysreq1 = (_USF_sysreq_node *) pool_take (_USF_sysreq_node_pool);
    pthread_mutex_lock (&_USF_mut_sysreq);
    _USF_sysreq_insert (hdr.tag, hdr.source, _USF_SEND, hdr.comm,
                        hdr.size, sysreq1);
    pthread_mutex_unlock (&_USF_mut_sysreq);

    /* check if there is a probe */
    req_probe = _USF_request_lookup_p (MPI_ANY_TAG, _USF_PROBE,
                                       MPI_COMM_WORLD, 0);
    if (req_probe != NULL) {
      pthread_mutex_lock (&_USF_mut_block);
      if (req_probe->req->blocking) {
        req_probe->req->end_of_transmission = 1;
        pthread_cond_signal (&_USF_cond_block);
      } else
        req_probe->req->end_of_transmission = 1;
      pthread_mutex_unlock (&_USF_mut_block);
    }

  } else {
    /* will match a MPI_ANY_SOURCE nonsent recv */
    /* Send a recv_post */
    if (req1->req->peer != MPI_ANY_SOURCE) {
      pthread_mutex_unlock (&_USF_mut_req);
      break;
    }
    DB ("rank %d SP will match a MPI_ANY_SOURCE\n", rank, 1);
    hdr.source = rank;
    hdr.dest = source;
    hdr.tag = req1->req->tag;
    hdr.comm = req1->req->comm;
```

```
    hdr.size = 0;
    hdr.type = _USF_RECVPOST;
    req1->req->peer = hdr.dest;       // not sure
    _USF_send_all (req1->req->peer, sizeof (_USF_message_header), &hdr,
                   "Packet isn't transmitted");
    req1->req->posted = 1;
    recv_post_account[req1->req->peer]++;
  }
  pthread_mutex_unlock (&_USF_mut_req);
  break;
case _USF_FCHUNK:
  DB ("rank %d, FCHUNK\n", rank, 1);
  if (header_msg) {
    _USF_conn_list[source].next_msg = hdr.type;
    memcpy (&(_USF_conn_list[source].hdr), &hdr,
            sizeof (_USF_message_header));
    break;
  }
  pthread_mutex_lock (&_USF_mut_req);
  req1 =
    _USF_request_lookup_m (hdr.tag, hdr.source, _USF_RECEIVE, hdr.comm, 0);
  _USF_recv_all (hdr.source, _USF_MAXDATASIZE, req1->req->buf,
                 "Can't receive ready mode message");
  req1->req->multiple = 1;
  req1->req->nchunk = 1;
  pthread_mutex_unlock (&_USF_mut_req);
  hdr.source = rank;
  hdr.dest = source;
  hdr.size = 0;
  hdr.type = _USF_ACKCHUNK;
  _USF_send_all (source, sizeof (_USF_message_header), &hdr,
                 "Packet isn't transmitted");
  _USF_conn_list[source].next_msg = _USF_HEADER_TYPE;
  break;
case _USF_MCHUNK:
  if (header_msg) {
    _USF_conn_list[source].next_msg = hdr.type;
    memcpy (&(_USF_conn_list[source].hdr), &hdr,
            sizeof (_USF_message_header));
    break;
  }
  pthread_mutex_lock (&_USF_mut_req);
  req1 =
    _USF_request_lookup_m (hdr.tag, hdr.source, _USF_RECEIVE, hdr.comm, 1);
  _USF_recv_all (hdr.source, _USF_MAXDATASIZE,
                 req1->req->buf + _USF_MAXDATASIZE * req1->req->nchunk,
                 "Can't receive ready mode message");
  req1->req->nchunk++;
  pthread_mutex_unlock (&_USF_mut_req);
  hdr.source = rank;
  hdr.dest = source;
  hdr.size = 0;
  hdr.type = _USF_ACKCHUNK;
  _USF_send_all (source, sizeof (_USF_message_header), &hdr,
                 "Packet isn't transmitted");
  _USF_conn_list[source].next_msg = _USF_HEADER_TYPE;
  break;
case _USF_LCHUNK:
  if (header_msg) {
    _USF_conn_list[source].next_msg = hdr.type;
    memcpy (&(_USF_conn_list[source].hdr), &hdr,
            sizeof (_USF_message_header));
    break;
```

```
    }
    pthread_mutex_lock (&_USF_mut_req);
    req1 =
      _USF_request_lookup_m (hdr.tag, hdr.source, _USF_RECEIVE, hdr.comm, 1);
    _USF_recv_all (hdr.source, hdr.size,
                   req1->req->buf + _USF_MAXDATASIZE * req1->req->nchunk,
                   "Can't receive ready mode message");
    req1->req->size = _USF_MAXDATASIZE * req1->req->nchunk + hdr.size;
    req1->req->done = 1;
    DB ("rank %d, done receiving!\n", rank, 1);
    /* Check if it was blocking !!! */
    pthread_mutex_lock (&_USF_mut_block);
    if (req1->req->blocking) {
      DB ("rank %d, wake up main\n", rank, 1);
      req1->req->end_of_transmission = 1;
      pthread_cond_signal (&_USF_cond_block);
    } else
      req1->req->end_of_transmission = 1;
    pthread_mutex_unlock (&_USF_mut_block);
    pthread_mutex_unlock (&_USF_mut_req);
    _USF_conn_list[source].next_msg = _USF_HEADER_TYPE;
    DB ("rank %d, LCHUNK ends\n", rank, 1);
    break;
  case _USF_ACKCHUNK:
    pthread_mutex_lock (&_USF_mut_req);
    hdr.source = rank;
    hdr.dest = source;
    hdr.tag = _USF_synchr_list[source]->tag;
    hdr.comm = _USF_synchr_list[source]->comm;
    /* Check if it the final message to send out */
#ifdef DEBUG
    printf ("rank %d, --> %d %d \n", rank,
            ((_USF_synchr_list[source]->nchunk + 1) * _USF_MAXDATASIZE)
            , _USF_synchr_list[source]->size);
    fflush (stdout);
#endif
    if (((_USF_synchr_list[source]->nchunk + 1) *       /* MCHUNK */
         _USF_MAXDATASIZE) < _USF_synchr_list[source]->size) {
      hdr.size = _USF_MAXDATASIZE;
      hdr.type = _USF_MCHUNK;
      _USF_send_all (source, sizeof (_USF_message_header), &hdr,
                     "Header isn't transmitted in ACKCHU");
      _USF_send_all (source, _USF_MAXDATASIZE,
                     _USF_synchr_list[source]->buf
                     + (_USF_synchr_list[source]->nchunk * _USF_MAXDATASIZE)
                     , "Can't sent ACKCHU, MCHUNK");
      _USF_synchr_list[source]->nchunk++;
      pthread_mutex_unlock (&_USF_mut_req);
    } else {                    /* LCHUNK */
      hdr.size = _USF_synchr_list[source]->size -
        (_USF_synchr_list[source]->nchunk * _USF_MAXDATASIZE);
      hdr.type = _USF_LCHUNK;
      _USF_send_all (source, sizeof (_USF_message_header), &hdr,
                     "Header isn't transmitted in ACKCHU");
      _USF_send_all (source, hdr.size,
                     _USF_synchr_list[source]->buf
                     + (_USF_synchr_list[source]->nchunk * _USF_MAXDATASIZE)
                     , "Can't sent ACKCHU, FCHUNK");
      DB ("rank %d, !!! %d \n", rank, _USF_synchr_list[source]->blocking);
      _USF_synchr_list[source]->done = 1;
      /* Check if it was blocking !!! */
      pthread_mutex_lock (&_USF_mut_block);
      if (_USF_synchr_list[source]->blocking) {
```

```
      DB ("rank %d, wake up main\n", rank, 1);
      _USF_synchr_list[source]->end_of_transmission = 1;
      pthread_cond_signal (&_USF_cond_block); // we might have problem here
    } else
      _USF_synchr_list[source]->end_of_transmission = 1;
    pthread_mutex_unlock (&_USF_mut_block);
    DB ("rank %d, signal sent! \n", rank, 1);
    _USF_synchr_list[source] = NULL;
    /* Done transmitting the message serve the recv post
       requests that came during transmission */
    pthread_mutex_unlock (&_USF_mut_req);
    _USF_serve_sysreq (source);
  }
  DB ("rank %d, mut_req unlocked! \n", rank, 1);
  break;
case _USF_SHORTMES:
  DB ("rank %d Get a short mes \n", rank, 1);
  if (header_msg) {
    _USF_conn_list[source].next_msg = hdr.type;
    memcpy (&(_USF_conn_list[source].hdr), &hdr,
            sizeof (_USF_message_header));
    if (hdr.size != 0)
      break;
  }
  pthread_mutex_lock (&_USF_mut_req);
  req1 =
    _USF_request_lookup_m (hdr.tag, hdr.source, _USF_RECEIVE, hdr.comm, 0);
  if (hdr.size > req1->req->size) {
    printf ("Incoming message is too big in ready mode\n");
    Error (ERF_RECV);
  }
  req1->req->size = hdr.size;
  _USF_recv_all (hdr.source, req1->req->size,
                 req1->req->buf, "Can't receive ready mode message");
  req1->req->done = 1;
  /* Check if it was blocking !!! */
  pthread_mutex_lock (&_USF_mut_block);
  if (req1->req->blocking) {
    DB ("rank %d, wake up main\n", rank, 1);
    req1->req->end_of_transmission = 1;
    pthread_cond_signal (&_USF_cond_block);
  } else
    req1->req->end_of_transmission = 1;
  pthread_mutex_unlock (&_USF_mut_block);
  pthread_mutex_unlock (&_USF_mut_req);
  _USF_conn_list[source].next_msg = _USF_HEADER_TYPE;
  DB ("rank %d short mes received\n", rank, 1);
  break;
case _USF_COMMTERM:
  if (rank == 0) {
    _USF_finalize_list[source] = 1;
  } else {
    printf ("Comm_term mes received in handle mes rank %d\n", rank);
    fflush (stdout);
    exit (0);
  }
  DB ("rank %d commterm received\n", rank, 1);
  break;
default:                      /* Error! */
  printf ("Unexpected message type in handle_msg!\n");
  fflush (stdout);
  Error (ERF_RECV);
  break;
```

```
  }
  return;
}


void
_USF_serve_mpi_requests (int rank, int nump)
{
  int i;                        /* Loop variable */
  _USF_message_header hdr;      /* The message header */
  _USF_request_node *curr;      /* The pointer used to traverse the list */
  MPI_Request_Int *req;         /* The pointer used to store MPI_Request_Int */
  _USF_sysreq_node *sysreq1;    /* The pointer used to retrieve sysreq */
  int send_size;                /* The number of bytes sent */
  int sysreq_flag;              /* Shows if there was a matching sysreq */


  pthread_mutex_lock (&_USF_mut_req);
  curr = _USF_request_hd;
  /* Traverse and serve all the requests */
  while (curr != NULL) {
    req = curr->req;
    DB ("rank %d s_mpi_req fst type %d\n", rank, req->type);
    /* continue if already completed */
    if ((req->done) || (req->posted) || (req->active == 0)) {
      DB ("rank %d found completed or posted or inactive\n", rank, 1);
      curr = curr->next;
      continue;
    }
    if (req->type == _USF_SEND) {       /* type send */
      pthread_mutex_lock (&_USF_mut_sysreq);

      /* Check for a matching sysreq recv post */
      sysreq1 = _USF_sysreq_lookup (req->tag, req->peer,
                                    _USF_RECEIVE, req->comm);
      DB ("rank %d serve mpi _USF_SEND request.\n", rank, 1);
      if (sysreq1 == NULL) {    /* send send_post */
        /* if it is a ready mode message send
         * immediately */
        DB ("rank %d Send a send_post\n", rank, 1);
        hdr.source = rank;
        hdr.dest = req->peer;
        hdr.tag = req->tag;
        hdr.comm = req->comm;
        hdr.size = req->size;
        hdr.type = _USF_SENDPOST;
        hdr.send_seq_num = ++send_post_account[req->peer];
        _USF_send_all (req->peer, sizeof (_USF_message_header), &hdr,
                       "Packet isn't transmitted");
        req->posted = 1;
        DB ("rank %d Sent a send_post!!!\n", rank, 1);
      } else {                  /* if there is a request send the message */
        if (_USF_synchr_list[req->peer] == NULL) {
          send_post_account[req->peer]++;
          DB ("rank %d Send the message2\n", rank, 1);
          /* only serve long communication if no ongoing long communication */
          _USF_handle_recv_post (sysreq1->source,
                                 rank, sysreq1->tag, sysreq1->comm,
                                 req->buf, req->size, req->multiple);
          /* update mpi_request */
          if (req->multiple) {
            req->posted = 1;
            req->nchunk = 1;
```

```
          _USF_synchr_list[req->peer] = req;
        } else {
          req->done = 1;
          /* Check if it was blocking !!! */
          pthread_mutex_lock (&_USF_mut_block);
          if (req->blocking) {
            DB ("rank %d, wake up main\n", rank, 1);
            req->end_of_transmission = 1;
            pthread_cond_signal (&_USF_cond_block);
          } else
            req->end_of_transmission = 1;
          pthread_mutex_unlock (&_USF_mut_block);
        }
        pool_return (_USF_sysreq_node_pool, (pool_item *)
                     _USF_sysreq_remove (req->tag, req->peer,
                                         _USF_RECEIVE, req->comm));
      } else
        blocking_for_big_message[req->peer]++;
    }
    pthread_mutex_unlock (&_USF_mut_sysreq);
  } else if (req->type == _USF_RECEIVE) {     /* type receive */
    DB ("rank %d serve _USF_RECEIVE request\n", rank, 1);
    pthread_mutex_lock (&_USF_mut_sysreq);
    /* Check for a matching sysreq send post */
    DB ("rank %d, req->peer %d\n", rank, req->peer);
    sysreq1 = _USF_sysreq_lookup (req->tag, req->peer,
                                  _USF_SEND, req->comm);
    /* If there is remove it */
    if (sysreq1 == NULL) {
      DB ("rank %d, NOT FOUND SYSREQ\n", rank, 1);
      sysreq_flag = 0;
    } else {
      DB ("rank %d, FOUND SYSREQ src %d\n", rank, sysreq1->source);
      sysreq_flag = 1;
      req->peer = sysreq1->source;
      pool_return (_USF_sysreq_node_pool, (pool_item *)
                   _USF_sysreq_remove (req->tag, req->peer, _USF_SEND,
                                       req->comm));
      DB ("rank %d, After _USF_SEND Remove %d\n", rank, req->peer);
    }
    pthread_mutex_unlock (&_USF_mut_sysreq);

    if ((req->peer != MPI_ANY_SOURCE) || sysreq_flag) {
      DB ("rank %d, GO into send recv_post\n", rank, 1);
      /* Send a recv_post */
      hdr.source = rank;
      hdr.dest = req->peer;
      hdr.tag = req->tag;
      hdr.comm = req->comm;
      hdr.size = 0;
      hdr.type = _USF_RECVPOST;
      DB ("rank %d Send recv post\n", rank, 1);
      _USF_send_all (req->peer, sizeof (_USF_message_header), &hdr,
                     "Packet isn't transmitted");
      req->posted = 1;
      recv_post_account[req->peer]++;
    }
  }
  curr = curr->next;
}
pthread_mutex_unlock (&_USF_mut_req);
DB ("rank %d, unlock mut_req before exiting serve_mpi_req\n", rank, 1);
return;
```

```
}

void *
_USF_comm_thread (void *arg)
{
#ifdef _USF_MPI_TCP_
  _USF_TCP_comm_thread ();
#endif
#ifdef _USF_MPI_GM_
  _USF_GM_comm_thread ();
#endif
  pthread_exit ((void *) 1);
}


#endif
```

B.13 datatype.c

```
// MPI derived datatype implementation
// Created by Chaney 2002
// Uniersity of San Francisco

#include <string.h>
#include "mpi.h"
int DDType_Count = MPI_Primitive_Type;
_USF_MPI_struct_type *struct_head = NULL;

//MPI pack
int
MPI_Pack (void *pack_data, int in_count, MPI_Datatype datatype, void *buffer,
          int buffer_size, int *position, MPI_Comm comm)
{
  memcpy (buffer + *position, pack_data,
          _USF_mpi_sizeof (datatype) * in_count);
  *position += _USF_mpi_sizeof (datatype) * in_count;
}

//MPI unpack
int
MPI_Unpack (void *buffer, int size, int *position, void *unpack_data,
            int count, MPI_Datatype datatype, MPI_Comm comm)
{
  memcpy (unpack_data, buffer + *position,
          _USF_mpi_sizeof (datatype) * count);
  *position += _USF_mpi_sizeof (datatype) * count;
}

//MPI Address
int
MPI_Address (void *location, MPI_Aint * address)
{
  *address = (MPI_Aint) ((char *) location - (char *) MPI_BOTTOM);
  return MPI_SUCCESS;
}

//_USF_DDType_Get_Type
int
_USF_DDType_Get_Type (int idx, _USF_MPI_struct_type ** st)
{
  _USF_MPI_struct_type *ptr;
```

```
  if (idx < DDType_Count) {
    ptr = struct_head;
    while (ptr != NULL) {
      if (ptr->idx == idx) {
        break;
      }
      ptr = ptr->next;
    }
    *st = ptr;
    if (*st != NULL) {
      return MPI_SUCCESS;
    }
    return -1;
  }
  return -1;
}

//MPI Type struct
int
MPI_Type_struct (int count, int blocklens[], MPI_Aint indices[],
                 MPI_Datatype old_types[], MPI_Datatype * newtype)
{
  _USF_MPI_struct_type *newstruct;
  int i;

  newstruct = (_USF_MPI_struct_type *) malloc (sizeof (_USF_MPI_struct_type));
  newstruct->count = count;
  newstruct->block_lengths = (int *) malloc (sizeof (int) * count);
  newstruct->disp = (MPI_Aint *) malloc (sizeof (MPI_Aint) * count);
  newstruct->typelist =
    (MPI_Datatype *) malloc (sizeof (MPI_Datatype) * count);
  for (i = 0; i < count; i++) {
    newstruct->block_lengths[i] = blocklens[i];
    newstruct->disp[i] = indices[i];
    newstruct->typelist[i] = old_types[i];
  }
  newstruct->idx = DDType_Count;
  //link new node to linked list
  if (struct_head == NULL) {
    struct_head = newstruct;
    newstruct->next = NULL;
  } else {
    newstruct->next = struct_head;
    struct_head = newstruct;
  }
  *newtype = DDType_Count;
  DDType_Count++;
  return MPI_SUCCESS;
}

//MPI Type vector
int
MPI_Type_vector (int count, int block_length, int stride,
                 MPI_Datatype element_type, MPI_Datatype * new_mpi_t)
{
  _USF_MPI_struct_type *newstruct;
  int i;

  newstruct = (_USF_MPI_struct_type *) malloc (sizeof (_USF_MPI_struct_type));
  newstruct->count = count;
  newstruct->block_lengths = (int *) malloc (sizeof (int) * count);
  newstruct->disp = (MPI_Aint *) malloc (sizeof (MPI_Aint) * count);
```

```
  newstruct->typelist =
    (MPI_Datatype *) malloc (sizeof (MPI_Datatype) * count);
  for (i = 0; i < count; i++) {
    newstruct->block_lengths[i] = block_length;
    newstruct->disp[i] = stride * i;
    newstruct->typelist[i] = element_type;
  }
  newstruct->idx = DDType_Count;
  //add new node to linked list
  if (struct_head == NULL) {
    struct_head = newstruct;
    newstruct->next = NULL;
  } else {
    newstruct->next = struct_head;
    struct_head = newstruct;
  }
  *new_mpi_t = DDType_Count;
  DDType_Count++;
  return MPI_SUCCESS;
}

//MPI Type Contiguous
int
MPI_Type_contiguous (int count, MPI_Datatype old_type,
                     MPI_Datatype * new_mpi_t)
{
  _USF_MPI_struct_type *newstruct;
  int i, size;

  newstruct = (_USF_MPI_struct_type *) malloc (sizeof (_USF_MPI_struct_type));
  newstruct->count = 1;
  newstruct->block_lengths = (int *) malloc (sizeof (int));
  newstruct->disp = (MPI_Aint *) malloc (sizeof (MPI_Aint));
  newstruct->typelist = (MPI_Datatype *) malloc (sizeof (MPI_Datatype));
  size = _USF_mpi_sizeof (old_type);
  newstruct->block_lengths[0] = count;
  newstruct->disp[0] = 0;
  newstruct->typelist[0] = old_type;
  newstruct->idx = DDType_Count;
  if (struct_head == NULL) {
    struct_head = newstruct;
    newstruct->next = NULL;
  } else {
    newstruct->next = struct_head;
    struct_head = newstruct;
  }
  *new_mpi_t = DDType_Count;
  DDType_Count++;
  return MPI_SUCCESS;
}

//MPI Type indexed
int
MPI_Type_indexed (int count, int block_lengths[], MPI_Aint displacements[],
                  MPI_Datatype old_type, MPI_Datatype * new_mpi_t)
{
  _USF_MPI_struct_type *newstruct;
  int i;

  newstruct = (_USF_MPI_struct_type *) malloc (sizeof (_USF_MPI_struct_type));
  newstruct->count = count;
  newstruct->block_lengths = (int *) malloc (sizeof (int) * count);
  newstruct->disp = (MPI_Aint *) malloc (sizeof (MPI_Aint) * count);
```

```
  newstruct->typelist =
    (MPI_Datatype *) malloc (sizeof (MPI_Datatype) * count);
  for (i = 0; i < count; i++) {
    newstruct->block_lengths[i] = block_lengths[i];
    newstruct->disp[i] = displacements[i];
    newstruct->typelist[i] = old_type;
  }
  newstruct->idx = DDType_Count;
  //add new node to linked list
  if (struct_head == NULL) {
    struct_head = newstruct;
    newstruct->next = NULL;
  } else {
    newstruct->next = struct_head;
    struct_head = newstruct;
  }
  *new_mpi_t = DDType_Count;
  DDType_Count++;
  return MPI_SUCCESS;
}
```

B.14 finalize.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>
#include <pthread.h>
#include "mpi.h"
#include "gm_thread.h"
#ifdef _USF_MPI_GM_
#include "gm.h"
#endif

//#define DEBUG
#ifdef _USF_MPI_GM_
extern pthread_t _USF_gm_deamon;        /* The gm polling thread thread */
#endif

extern int ack_pending;

int
MPI_Finalize (void)
{
  int i;                        /* Loop variable. */
  int nump;                     /* Number of processors. */
  int rank;                     /* The rank of the current processor. */
  _USF_request_node *curr;      /* The pointer used to traverse the list */
  MPI_Request_Int *req;         /* The pointer used to store MPI_Request_Int */
  _USF_message_header hdr;
  MPI_Status status;

#ifdef _USF_MPI_GM_
```

```
  gm_recv_event_t *event;
  while (ack_pending) {
    event = gm_receive (local_port);
    switch (gm_ntoh_u8 (event->recv.type)) {
    case GM_NO_RECV_EVENT:
      break;
    default:
      gm_unknown (local_port, event);
    }
  }

#endif

  MPI_Comm_size (MPI_COMM_WORLD, &nump);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);



#ifdef DEBUG
  printf ("Node %d Finalize start to MPI_wait\n", rank);
  fflush (stdout);

#endif
  /* Wait for the uncompleted MPI_Requests */
  for (curr = _USF_request_hd; curr != NULL; curr = curr->next) {
    /* These requests can't be blocking */
#ifdef DEBUG
    printf ("Node %d Finalize start to _usf_req \n", rank);
    fflush (stdout);

#endif

    req = curr->req;
    if (!(req->done))
      MPI_Wait (&req, &status);
  }
  i = 1;                        /* Used to msg the comm_thread */
#ifdef DEBUG
  printf ("Node %d Finalize start to kill comm_thread\n", rank);
  fflush (stdout);

#endif


#ifdef _USF_MPI_TCP_
  write (_USF_s_pipe[1], &i, sizeof (int));
  pthread_join (_USF_comm_deamon, (void *) &i);
  /* if 0 receive the messages that haven't already sent,
     then send the termination messages. Like a mpi barrier
     if != 0 send the termination message to zero and wait for
     the termination message from zero
   */

#ifdef DEBUG
  printf ("Node %d Finalize start to recv and send\n", rank);
  fflush (stdout);

#endif

  if (rank == 0) {
    for (i = 1; i < nump; i++) {
      if (_USF_finalize_list[i] != 1) {
        _USF_recv_all (i, sizeof (_USF_message_header), &hdr,
```

```
                       "Couldn't receive comm term mes");
      }
    }
    hdr.source = 0;
    hdr.dest = i;
    hdr.tag = 0;
    hdr.comm = MPI_COMM_WORLD;
    hdr.size = 0;
    hdr.type = _USF_COMMTERM;
    for (i = 1; i < nump; i++) {
      hdr.dest = i;
      _USF_send_all (i, sizeof (_USF_message_header), &hdr,
                     "Couldn't send comm term mes");
    }
  } else {
    hdr.source = rank;
    hdr.dest = 0;
    hdr.tag = 0;
    hdr.comm = MPI_COMM_WORLD;
    hdr.size = 0;
    hdr.type = _USF_COMMTERM;
    _USF_send_all (0, sizeof (_USF_message_header), &hdr,
                   "Couldn't send comm term mes");
    _USF_recv_all (0, sizeof (_USF_message_header), &hdr,
                   "Couldn't receive comm term mes");
    if (hdr.type != _USF_COMMTERM) {
      printf ("Should have been a comm_term mes\n");
      exit (0);
    }
  }

#endif
  /* if 0 receive the messages that haven't already sent,
     then send the termination messages. Like a mpi barrier
     if != 0 send the termination message to zero and wait for
     the termination message from zero
   */

  //--killing gm thread
#ifdef DEBUG
  printf ("Node %d Finalize start to kill gm_thread\n", rank);
  fflush (stdout);
#endif

#ifdef _USF_MPI_GM_
  gm_thread_end ();
  pthread_join (_USF_gm_deamon, (void *) &i);
#endif


#ifdef DEBUG
  printf ("Node %d Starting _USF_finalize\n", rank);
  fflush (stdout);
#endif


  _USF_finalize (rank, nump);
  pool_drain (_USF_mpi_request_int_pool);
  pool_drain (_USF_request_node_pool);
  pool_drain (_USF_sysreq_node_pool);
  /* Free the receive buffer  Not implemented yet! */
  return 0;
}
```

B.15 gather.c

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<unistd.h>
#include"errno.h"
#include"mpi.h"

// tag -12 is used for gather.
int
MPI_Gather (void *sendbuf, int sendcount, MPI_Datatype sendtype,
            void *recvbuf, int recvcount, MPI_Datatype recvtype,
            int root, MPI_Comm comm)
{
  int nump, rank, i, j, s;
  MPI_Request *req;
  MPI_Status *status;

  MPI_Comm_size (comm, &nump);
  MPI_Comm_rank (comm, &rank);

  s = _USF_mpi_sizeof (recvtype);

  req = (MPI_Request *) malloc ((nump - 1) * sizeof (MPI_Request));
  status = (MPI_Status *) malloc ((nump - 1) * sizeof (MPI_Status));

  if (rank == root) {
    for (i = 0, j = 0; i < nump; i++, j++) {
      if (i == rank) {
        j--;
        continue;
      }
      MPI_Irecv ((recvbuf + (i * recvcount * s)), recvcount,
                 recvtype, i, -12, comm, &(req[j]));
    }

    memcpy ((recvbuf + (root * recvcount * s)), sendbuf, recvcount * s);
    MPI_Waitall ((nump - 1), req, status);
  } else {
    MPI_Send (sendbuf, sendcount, sendtype, root, -12, comm);
  }
  return 0;
}

//tag -17 is used for gatherv
int
MPI_Gatherv (void *sendbuf, int sendcount, MPI_Datatype sendtype,
             void *recvbuf, int *recvcount, int *displs,
             MPI_Datatype recvtype, int root, MPI_Comm comm)
{
  int nump, rank, i, j, s;
  MPI_Request *req;
  MPI_Status *status;

  MPI_Comm_size (comm, &nump);
  MPI_Comm_rank (comm, &rank);

  if (rank == -1) {
    printf ("Wrong communicator in calling MPI_Gather\n");
    fflush (stdout);
    return 1;
  }
```

```
  s = _USF_mpi_sizeof (recvtype);

  req = (MPI_Request *) malloc ((nump - 1) * sizeof (MPI_Request));
  status = (MPI_Status *) malloc ((nump - 1) * sizeof (MPI_Status));

  if (rank == root) {
    for (i = 0, j = 0; i < nump; i++, j++) {
      if (i == rank) {
        j--;
        continue;
      }
      MPI_Irecv (recvbuf + displs[i], recvcount[i],
                 recvtype, i, -17, comm, &(req[j]));
    }

    memcpy (recvbuf + displs[root], sendbuf, recvcount[root] * s);
    MPI_Waitall ((nump - 1), req, status);
  } else {
    MPI_Send (sendbuf, sendcount, sendtype, root, -17, comm);
  }
  return 0;
}
```

B.16 get_count.c

```
#include "mpi.h"

int
MPI_Get_count (MPI_Status * status, MPI_Datatype datatype, int *count)
{
  *count = status->count / _USF_mpi_sizeof (datatype);
  return *count;
}
```

B.17 gm.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/tcp.h>
#include <sys/wait.h>
#include <signal.h>
#include "mpi.h"
#include "gm.h"
#include "gm_thread.h"

#ifdef _USF_MPI_GM_
struct gm_port *local_port;     // this is the GM port for communication
```

```
int local_port_id;              // this is the GM port ID
int my_board_num = 0;           // the first GM network card number,
                                // if you have multiple board,
                                // it should be 0, 1, 2, 3 ...
extern dma_mem_reg_table *mem_reg_table;
void *send_header_list[_USF_MAX_HEADERS_];      // for sending post
void *debug_send_header_list[_USF_MAX_HEADERS_];
int send_header_stack[_USF_MAX_HEADERS_];
int send_header_stack_top = 0;

#define DB(a,b,c)
#ifdef DEBUG
#undef DB
#define DB(a,b,c) printf (a,b,c); fflush (stdout);
#endif /*  */
int _USF_num_of_group = 0;      // group reference num
int _USF_group_max = 64;        // maximum group number
int _USF_group_ref = 0;
int _USF_comm_max = 64;         // maximum communicator number;
int _USF_comm_ref = 200;
struct _USF_rank_type *_USF_comm_list;  /* The communicator list */
struct _USF_group_type *_USF_group_list;        /* The group list */
int _USF_num_of_comm = 0;       /* The number of communicators present */
pthread_t _USF_comm_deamon;     /* The communication thread */
pthread_t _USF_gm_deamon;       /* The gm polling thread thread */
int *_USF_finalize_list;        /* The list that holds the execution completion
                                 * signals */
pthread_mutex_t _USF_mut_sysreq;        /* sys_request link list */
pthread_mutex_t _USF_mut_req;   /* MPI_Request_Int link list */
pthread_mutex_t _USF_mut_block; /* Blocking communication */
pthread_cond_t _USF_cond_block; /* Blocking Communication */
pthread_mutex_t _USF_mut_gm_send;
pthread_cond_t _USF_cond_gm_send;
pthread_mutex_t _USF_mut_gm_thread;
pthread_cond_t _USF_cond_gm_thread;
int _USF_s_pipe[2];             /* 0 read, 1 write */
int _USF_gm_pipe[2];            // GM thread pipe
//int *heads;
#define HEAD_LENGTH sizeof(int)*6

// Used for searching GM id
int _USF_number_proc;

MPI_Request_Int *_USF_synchr_list[_USF_MAXPROCESSORS];

  /* The array that keeps the requests ptrs of ongoing
   * synch communications NULL means no comm. */
void usf_gm_send_callback (struct gm_port *port, void *context,
                           gm_status_t the_status);
int _USF_GM_open_port (struct gm_port **p);
void *_USF_comm_thread (void *arg);
int
_USF_GM_init (int *argc, char ***argv)
{
  int i;
  char *my_name;
  int err;                      // The error value.
  int nump;                     /* Number of processors. */
  int rank;                     /* The rank of the current processor. */
  void *ptr;
  int recv_tokens;

  // initialize GM system
```

```
#ifdef DEBUG
  printf (" --> Start to run _USF_GM_init \n");
  fflush (stdout);

#endif

// open GM port
  gm_init ();


  // create communicator
  // create the default comm and group list, the length is 16,
  // we can enlarge it later when needed
  _USF_comm_list =
    (struct _USF_rank_type *) malloc (_USF_comm_max *
                                      sizeof (struct _USF_rank_type));
  _USF_group_list =
    (struct _USF_group_type *) malloc (_USF_group_max *
                                       sizeof (struct _USF_group_type));
  _USF_comm_list[0].nump = atoi (*((*argv) + 2));
  nump = _USF_comm_list[0].nump;
  _USF_number_proc = nump;
  for (i = 0; i < _USF_group_max; i++) {
    _USF_group_list[i].rank_old = (int *) malloc (nump * sizeof (int));
    _USF_group_list[i].rank_global = (int *) malloc (nump * sizeof (int));
  }

  // Get your own name
  my_name = (char *) malloc (30 * sizeof (char));
  if (gethostname (my_name, 30) != 0)
    return Error (ER_HOSTNM);

  // Assume argc will be true since called internally.
  _USF_comm_list[0].rank = atoi (*((*argv) + 1));
  rank = _USF_comm_list[0].rank;

  // initialize the MPI_COMM_WORLD
  _USF_finalize_list = (int *) malloc (nump * sizeof (int));
  if ((err = _USF_make_conn_list (my_name, nump)) != 0)
    return err;
  _USF_comm_list[0].comm = MPI_COMM_WORLD;
  _USF_comm_list[0].old_comm = MPI_COMM_WORLD;
  _USF_num_of_comm = 1;
  _USF_num_of_group = 1;

  // initialize the group information for MPI_COMM_WORLD
  _USF_comm_list[0].group = 0;
  _USF_group_list[0].group_no_old = 0;
  _USF_group_list[0].group_no_local = 0;
  _USF_group_ref++;             // intial value is 0
  _USF_group_list[0].group_size = nump;
  for (i = 0; i < nump; i++) {
    _USF_group_list[0].rank_global[i] = _USF_group_list[0].rank_old[i] = i;
  }

  // open a GM port
  _USF_GM_open_port (&local_port);
  gm_allow_remote_memory_access (local_port);

  //provide enough memory to gm_recv
  gm_free_send_tokens (local_port, GM_LOW_PRIORITY,
                       gm_num_send_tokens (local_port));
```

```
  // get node_id
  for (i = 0; i < nump; i++) {
    _USF_conn_list[i].node_id =
      gm_host_name_to_node_id (local_port, _USF_conn_list[i].hostname);

  }

  for (i = 0; i < _USF_MAX_HEADERS_; i++) {
    send_header_list[i] =
      gm_dma_malloc (local_port, sizeof (struct _USF_header) * 64);
    debug_send_header_list[i] =
      gm_dma_malloc (local_port, sizeof (struct _USF_header) * 64);
    send_header_stack[i] = i;

  }

  mem_reg_table =
    (dma_mem_reg_table *) malloc (sizeof (dma_mem_reg_table) *
                                  MAX_MEM_REG_ENTRY);
  for (i = 0; i < MAX_MEM_REG_ENTRY; i++) {
    mem_reg_table[i].p = NULL;
    mem_reg_table[i].duplicate_link = NULL;
  }

  // get rid of extra three parameters
  *((*argv) + 3) = *((*argv) + 0);
  (*argv) = (*argv) + 3;
  *argc -= 3;
  return 0;
}

int
_USF_GM_init_comm ()
{
}
int
_USF_GM_open_port (struct gm_port **p)
{
  gm_status_t my_status;
  int i;

  // open local GM port
  local_port_id = _USF_conn_list[_USF_comm_list[0].rank].port_id;
  my_status = gm_open (p, my_board_num, local_port_id, "USFMPI_GM",
                       (enum gm_api_version) GM_API_VERSION_1_1);
  if (my_status != GM_SUCCESS) {
    gm_perror ("[Init] Couldn't open GM port", my_status);
    gm_exit (1);
  }
#ifdef DEBUG
  gm_get_node_id (local_port, &i);
  gm_printf ("[Init] --> Opened board %d port %d on node id %d\n",
             my_board_num, local_port_id, i);

#endif
}

int
_USF_GM_thread_init ()
{

  pthread_mutex_init (&_USF_mut_sysreq, NULL);
  pthread_mutex_init (&_USF_mut_req, NULL);
```

```
  pthread_mutex_init (&_USF_mut_block, NULL);
  pthread_cond_init (&_USF_cond_block, NULL);
  gm_thread_init ();

#ifndef SINGLE_THREAD
  if (pthread_create (&_USF_gm_deamon, NULL, gm_thread, NULL) != 0) {
    return Error (ERF_PTHREAD);
  }
#endif /*  */

  //      create GM polling thread here
  return 0;
}

void
_USF_GM_comm_thread ()
{
}
int
_USF_GM_prepare_set (fd_set * set)
{
  int i;                        /* Loop variable */
  int nump;                     /* # of procs in COMM_WORLD */
  int rank;                     /* the rank in COMM_WORLD */
  int max = -1;                 /* The maximum file descriptor */
  MPI_Comm_size (MPI_COMM_WORLD, &nump);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  FD_ZERO (set);
  if (_USF_s_pipe[0] > max)
    max = _USF_s_pipe[0];
  FD_SET (_USF_s_pipe[0], set);

#ifdef DEBUG
  printf("rank--> %i s_pipe[0] %d\n",rank,_USF_s_pipe[0]); fflush(stdout);
#endif /*  */
  if (_USF_gm_pipe[0] > max)
    max = _USF_gm_pipe[0];
  FD_SET (_USF_gm_pipe[0], set);

  return max;
}

int
_USF_GM_send_all (int dest, int size, void *buf, char *err_mes)
{
  int total = 0;
  int send_size;
  int rank;
  int dest_node_id;
  int dest_port_id;
  int i;
  unsigned int test_node_id;
  int *buf_ptr;
  int blocked = 1;
  gm_register_memory (local_port, buf, size);
  dest_node_id = _USF_conn_list[dest].node_id;
  dest_port_id = _USF_conn_list[dest].port_id;

  gm_send_with_callback (local_port, buf, GM_PKT_SIZE,
                         size + 1, GM_LOW_PRIORITY, dest_node_id,
                         dest_port_id, usf_gm_send_callback, &blocked);

  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
```

```
  pthread_mutex_lock (&_USF_mut_gm_send);
  if (blocked == 1)             //modified by Qing Huang
  {
    pthread_cond_wait (&_USF_cond_gm_send, &_USF_mut_gm_send);
  }
  pthread_mutex_unlock (&_USF_mut_gm_send);

  return size;
}

int
_USF_GM_Finalize ()
{
  int i;
  dma_mem_reg_table *pre, *next;
  for (i = 0; i < _USF_group_max; i++) {
    free (_USF_group_list[i].rank_old);
    free (_USF_group_list[i].rank_global);
  }

#ifdef DEBUG
  printf ("Start to free header buffer\n");
  fflush (stdout);

#endif
  for (i = 0; i < _USF_MAX_HEADERS_; i++) {
    gm_dma_free (local_port, send_header_list[i]);
  }

#ifdef DEBUG
  printf ("Start to deregister DMA buffer\n");
  fflush (stdout);
#endif
  for (i = 0; i < MAX_MEM_REG_ENTRY; i++) {
    if (mem_reg_table[i].p != NULL) {
      gm_deregister_memory (local_port, mem_reg_table[i].p,
                            mem_reg_table[i].size);
      next = mem_reg_table[i].duplicate_link;
      while (next != NULL) {
        pre = next;
        next = next->duplicate_link;
        gm_deregister_memory (local_port, pre->p, pre->size);
        free (pre);
      }
    }
  }

#ifdef DEBUG
  printf ("Start to free local memory data\n");
  fflush (stdout);

#endif
  free (mem_reg_table);
  free (_USF_comm_list);        // use array, not list
  free (_USF_group_list);
  free (_USF_conn_list);
  free (mem_reg_table);

#ifdef DEBUG
  printf ("Start to close GM\n");
  fflush (stdout);

#endif /*  */
```

```
#ifdef DEBUG
  printf ("Start to GM finalize\n");
  fflush (stdout);

#endif /*  */
  gm_finalize ();

#ifdef DEBUG
  printf ("Finish all GM\n");
  fflush (stdout);

#endif /*  */
}

void
usf_gm_send_callback (struct gm_port *port, void *context,
                      gm_status_t the_status)
{

  // One pending callback has been received
  //
  int rank;
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);

  pthread_mutex_lock (&_USF_mut_gm_send);
  (*(int *) context)--;
  pthread_mutex_unlock (&_USF_mut_gm_send);
  pthread_cond_signal (&_USF_cond_gm_send);
  if (the_status != GM_SUCCESS) {
    printf ("GM sending is wrong\n");
    fflush (stdout);
    if (the_status != GM_SEND_DROPPED) {
      gm_perror ("send completed with error\n", the_status);
      exit (1);
    }
  }
}


//-----------------------------------------------------------------------
// Added by Chaney
// get_rank_by_gm_id : Translate gm id to rank.
// Parameters :
// id : GM id.
// rank : mpi rank
//-----------------------------------------------------------------------
void
get_rank_by_gm_id (int node_id, int port_id, int *rank)
{
  int i;
  for (i = 0; i < _USF_number_proc; i++) {
    if (_USF_conn_list[i].node_id == node_id
        && _USF_conn_list[i].port_id == port_id) {
      *rank = i;
      return;
    }
  }
  printf ("Can't find gm id %d.", node_id);
  fflush (stdout);
  *rank = -100;
}
```

```
#endif
```

B.18 gm_thread.c

```
//-------------------------------------------------------------------------
// Gm_thread.c : GM message polling thread. Used for listening incomming
// message
// Author : Chaney Tsai 9/12/2002
// University of San Francisco
//-------------------------------------------------------------------------
#include <string.h>
#include <stdio.h>
#include "mpi.h"
#include "gm.h"
#include "gm_thread.h"

//#define DEBUG

#ifdef _USF_MPI_GM_
//#define USF_GM_DEBUG

#define DB(a,b,c)
#ifdef DEBUG
#undef DB
#define DB(a,b,c) printf (a,b,c); fflush (stdout);
#endif /*  */
int thread_recving;
int gm_thread_polling = 1;

//gm data queue node
typedef struct gm_msg_node
{
  int size;
  void *buf;
  int next;
  int prev;
}
gm_msg_node;
typedef struct gm_conn_node
{
  int head;
  int tail;
}
gm_conn_node;
gm_conn_node *gm_conn_msg_queue;
gm_msg_node *gm_msg_queue;
int gm_msg_queue_head;
pthread_mutex_t _USF_mut_gm_recv;
pthread_mutex_t _USF_mut_gm_pending;    // added by Qing
int request_pending = 0;
pthread_cond_t _USF_cond_gm_recv;
int waiting_for_data;
static int my_rank;
static int total_nodes;
int blocking_recv;
int ack_pending = 0;
extern void *send_header_list[_USF_MAX_HEADERS_];       // for sending post
extern void *debug_send_header_list[_USF_MAX_HEADERS_];
extern int send_header_stack[_USF_MAX_HEADERS_];
extern int send_header_stack_top;
```

```
void *request_list[_USF_MAX_HEADERS_];
int send_request_counter = 0;
int recv_request_counter = 0;
int message_counter = 0;
int recv_post_counter = 0;
int ack_post_counter = 0;
int send_post_counter = 0;
int direct_mem_counter = 0;
int direct_mem_callback_counter = 0;
int dest_node_flag = -1;
int dest_ack_flag = -1;
extern int isend_call;
extern int recv_call;
int ack_signal_counter = 0;
int ack_request_counter = 0;
int debug_counter = 0;
MPI_Request_Int *debug_current_req = NULL;
MPI_Request_Int *debug_current_req0 = NULL;

// gm_thread_int : Thread end function
// Set thread_recving to end gm polling thread.
void
gm_thread_init (void)
{
  void *ptr;
  int recv_tokens;
  int i;
  MPI_Comm_size (MPI_COMM_WORLD, &total_nodes);

  //Initial connection message queue
  //Provide buffer pool to GM
  recv_tokens =
    ((gm_num_receive_tokens (local_port) >
      GM_MAX_RECV_TOKENS) ? GM_MAX_RECV_TOKENS :
     gm_num_receive_tokens (local_port));
  for (i = 0; i < recv_tokens * 2; i++) {
    ptr = gm_dma_malloc (local_port, _USF_MAXDATASIZE);
    gm_provide_receive_buffer (local_port, ptr, _USF_MAXDATASIZE,
                               GM_LOW_PRIORITY);

#ifdef USF_GM_DEBUG
    printf ("Addredd = %p.\n", ptr);
    fflush (stdout);
#endif /*  */
  }
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
}


// gm_thread_end : Thread end function
// Set thread_recving to end gm polling thread.
void
gm_thread_end (void)
{
  thread_recving = 0;
}


int
_USF_send_header_pop ()
{

  if (send_header_stack_top < _USF_MAX_HEADERS_) {
```

```
    send_header_stack_top++;
    return send_header_stack[send_header_stack_top - 1];
  }

  else {
    printf ("send stack pop error\n");
    fflush (stdout);
    return -1;
  }
}
void
_USF_send_header_push (int free_header)
{
  if (send_header_stack_top > 0) {
    send_header_stack_top--;
    send_header_stack[send_header_stack_top] = free_header;
  }

  else
    printf ("rank %d error of push op\n", my_rank);
}

void
gm_send_post_callback (struct gm_port *port, void *context,
                       gm_status_t the_status)
{
  struct _USF_header *header, *debug_header;
  _USF_send_header_push ((int) context);

#ifdef SINGLE_THREAD
  thread_recving = 0;

#endif /*  */

}

void
_USF_gm_send_post (struct _USF_header *header, int i)
{
  gm_send_with_callback (local_port, header, _USF_MAXDATASIZE,
                         sizeof (struct _USF_header), GM_LOW_PRIORITY,
                         _USF_conn_list[header->dest].node_id,
                         _USF_conn_list[header->dest].port_id,
                         gm_send_post_callback, (void *) i);
}

void
gm_send_ack_post_callback (struct gm_port *port, void *context,
                           gm_status_t the_status)
{

  struct _USF_header *header, *debug_header;

  _USF_send_header_push ((int) context);
  ack_pending--;

#ifdef SINGLE_THREAD
  thread_recving = 0;

#endif /*  */

#ifdef USF_GM_DEBUG
  printf ("rank %d ark is over\n", my_rank);
```

```
  fflush (stdout);

#endif /*  */
  ack_request_counter++;

}

void
gm_handle_recv_post_callback (struct gm_port *port, void *context,
                              gm_status_t the_status)
{
  int i;
  struct _USF_header *header, *debug_header;
  if (the_status != GM_SUCCESS) {
    printf ("rank %d send data error\n", my_rank);
    fflush (stdout);
  }
  i = _USF_send_header_pop ();
  if (i == -1) {
    printf ("running out of header\n");
    fflush (stdout);
    exit (1);
  }
  header = (struct _USF_header *) send_header_list[i];
  header->source = my_rank;

  header->dest = ((MPI_Request_Int *) context)->peer;
  header->tag = ((MPI_Request_Int *) context)->tag;

  header->comm = ((MPI_Request_Int *) context)->comm;
  header->type = _USF_ACK_GMSEND;
  header->size = 0;

#ifdef USF_GM_DEBUG
  printf ("rank %d send ACK header to rank %d\n", my_rank, header->dest);
  fflush (stdout);

#endif
  ack_pending++;

  gm_send_with_callback (local_port, header, _USF_MAXDATASIZE,
                         sizeof (struct _USF_header), GM_LOW_PRIORITY,
                         _USF_conn_list[header->dest].node_id,
                         _USF_conn_list[header->dest].port_id,
                         gm_send_ack_post_callback, (void *) i);

  dest_ack_flag = _USF_conn_list[header->dest].node_id;

  pthread_mutex_lock (&_USF_mut_block);
  if (((MPI_Request_Int *) context)->blocking) {
    ((MPI_Request_Int *) context)->end_of_transmission = 1;
    ((MPI_Request_Int *) context)->done = 1;
    pthread_cond_signal (&_USF_cond_block);
  }

  else {
    ((MPI_Request_Int *) context)->done = 1;
    ((MPI_Request_Int *) context)->end_of_transmission = 1;
  }
  pthread_mutex_unlock (&_USF_mut_block);

}
```

```
void
_USF_gm_handle_recv_post (int dest, int source, int tag, int comm,
                          int in_size, void *in_buf, void *out_buf,
                          int out_size, MPI_Request_Int * req)
{
  int actual_size;
  int dest_node_id;
  int dest_port_id;
  gm_status_t temp;
  actual_size = (in_size <= out_size) ? in_size : out_size;
  dest_node_id = _USF_conn_list[dest].node_id;
  dest_node_flag = dest_node_id;
  dest_port_id = _USF_conn_list[dest].port_id;

  if (actual_size == 0) {
    temp = GM_SUCCESS;

    gm_handle_recv_post_callback (local_port, req, temp);
    return;
  }
  DB ("rank %d strat to send data\n", my_rank, 1)
    gm_directed_send_with_callback (local_port, out_buf,
                                    (gm_remote_ptr_t) (gm_up_t) in_buf,
                                    actual_size, GM_LOW_PRIORITY,
                                    dest_node_id, dest_port_id,
                                    gm_handle_recv_post_callback, req);
}

void
gm_handle_header (struct _USF_header *h_ptr)
{

  int i;                        /* Loop variable */
  int nump;                     /* # of procs in COMM_WORLD */

  int recv_size;                /* The size of received packet */
  int size;                     /* Used for ready mode receive */
  _USF_request_node *req1, *req_probe;  /* The request node used for lookup */
  _USF_sysreq_node *sysreq1;    /* The sysreq node used to insert a node */
  int isongcomm = 0;            /* Is there an ongoiing communication */
  int header_msg = 0;
  int source;
  struct _USF_header *header;
  MPI_Request_Int *req;
  _USF_request_node *node = _USF_request_hd;
  source = h_ptr->source;
  switch (h_ptr->type) {
  case _USF_RECVPOST:
    recv_post_counter++;

    /* These are removed from sysreq by MPI_(I)SEND calls */
    /* Look for a send request in request list */
    DB ("rank %d, Get a recv_post\n", my_rank, 1);
    pthread_mutex_lock (&_USF_mut_req);
    req1 =
      _USF_request_lookup (h_ptr->tag, h_ptr->source, _USF_SEND, h_ptr->comm);
    DB ("rank %d, isong %d\n", my_rank, isongcomm);
    if (req1 == NULL) {
      sysreq1 = (_USF_sysreq_node *) pool_take (_USF_sysreq_node_pool);
      if (sysreq1 == NULL)
        DB ("rank %d Memory error", my_rank, 1);
      pthread_mutex_lock (&_USF_mut_sysreq);
      _USF_sysreq_insert (h_ptr->tag, h_ptr->source, _USF_RECEIVE,
```

```
                        h_ptr->comm, h_ptr->size, h_ptr->in_buf, sysreq1);
    pthread_mutex_unlock (&_USF_mut_sysreq);
  } else {
    DB ("rank %d, handle recv_post\n", my_rank, 1);
    _USF_gm_handle_recv_post (h_ptr->source, h_ptr->dest, h_ptr->tag,
                              h_ptr->comm, h_ptr->size, h_ptr->in_buf,
                              req1->req->buf, req1->req->size, req1->req);

    /* update mpi_request */
    req1->req->done = 1;
    if (req1->req->posted == 0)
      ++send_post_account[h_ptr->source];

    /* Check if it was blocking !!! */
  }
  pthread_mutex_unlock (&_USF_mut_req);
  break;
case _USF_SENDPOST:
  send_post_counter++;

  /* These are removed from sysreq by MPI_(I)RECV calls */
  /* Send a recv post if unsent recv post in mpi req list */
  DB ("rank %d Get a send_post\n", my_rank, 1);
  if (h_ptr->send_seq_num <= recv_post_account[h_ptr->source])
    break;
  pthread_mutex_lock (&_USF_mut_req);

  /* Continue with the operation */
  req1 = _USF_request_lookup_p (h_ptr->tag, _USF_RECEIVE, h_ptr->comm, 0);
  if (req1 == NULL) {         /* put in the sys req queue */

    //Received a send-post but there is no recv req in system queue
    DB ("rank %d SENDPOST NULL, put in sysreq Q\n", my_rank, 1);
    sysreq1 = (_USF_sysreq_node *) pool_take (_USF_sysreq_node_pool);
    pthread_mutex_lock (&_USF_mut_sysreq);
    _USF_sysreq_insert (h_ptr->tag, h_ptr->source, _USF_SEND, h_ptr->comm,
                        h_ptr->size, h_ptr->in_buf, sysreq1);
    pthread_mutex_unlock (&_USF_mut_sysreq);

    /* check if there is a probe */
    req_probe =
      _USF_request_lookup_p (MPI_ANY_TAG, _USF_PROBE, MPI_COMM_WORLD, 0);
    if (req_probe != NULL) {

      pthread_mutex_lock (&_USF_mut_block);
      if (req_probe->req->blocking) {
        req_probe->req->end_of_transmission = 1;
        pthread_cond_signal (&_USF_cond_block);
      }

      else
        req_probe->req->end_of_transmission = 1;
      pthread_mutex_unlock (&_USF_mut_block);
      thread_recving = 0;
    }
  } else {

    /* will match a MPI_ANY_SOURCE nonsent recv */
    /* Send a recv_post */
    if (req1->req->peer != MPI_ANY_SOURCE) {
      pthread_mutex_unlock (&_USF_mut_req);
      DB ("node %d don't do anything for send header\n", my_rank, 1);
      break;
```

```
      }
      DB ("node %d send recv post for any source to node %d\n", my_rank,
          h_ptr->source);
      req = req1->req;
      i = _USF_send_header_pop ();
      if (i == -1) {
        printf ("running out of header\n");
        fflush (stdout);
        exit (1);
      }
      header = (struct _USF_header *) send_header_list[i];
      header->source = my_rank;
      header->dest = h_ptr->source;
      req->peer = h_ptr->source;
      header->tag = req->tag;
      header->comm = req->comm;
      header->type = _USF_RECVPOST;
      header->size = req->size;
      header->in_buf = req->buf;
      _USF_gm_send_post (header, i);

      req1->req->posted = 1;
      recv_post_account[req1->req->peer]++;
    }
    pthread_mutex_unlock (&_USF_mut_req);
    break;
  case _USF_ACK_GMSEND:
    pthread_mutex_lock (&_USF_mut_req);
    DB ("rank %d, get a ack post, done receiving!\n", my_rank, 1);
    req1 =
      _USF_request_lookup_m (h_ptr->tag, h_ptr->source, _USF_RECEIVE,
                             h_ptr->comm, 0);
    ack_post_counter++;
    if (req1 != NULL) {
      pthread_mutex_lock (&_USF_mut_block);
      ack_signal_counter++;
      req1->req->done = 1;

      /* Check if it was blocking !!! */
      if (req1->req->blocking) {
        DB ("rank %d, wake up main\n", my_rank, 1);
        req1->req->end_of_transmission = 1;
        pthread_cond_signal (&_USF_cond_block);
      }

      else
        req1->req->end_of_transmission = 1;

#ifdef SINGLE_THREAD
      thread_recving = 0;

#endif
      pthread_mutex_unlock (&_USF_mut_block);
    } else {
      DB ("rank %d can't find RECV request, error ACK header\n", my_rank, 1);
      printf
        ("rank %d bad header source %d, dest %d, type %d comm %d, tag %d, size %d, \n",
         my_rank, h_ptr->source, h_ptr->dest, h_ptr->type, h_ptr->comm,
         h_ptr->tag, h_ptr->size);
      fflush (stdout);

      while (node != NULL) {
        node = node->next;
```

```
      }
    }
    pthread_mutex_unlock (&_USF_mut_req);
    break;
  default:
    DB ("rank %d , error header\n", my_rank, 1);
    break;
  }
}
void
_USF_gm_serve_mpi_requests (int rank, int nump)
{
  int i;                        /* Loop variable */
  int args[7];                  /* The message header */
  _USF_request_node *curr;      /* The pointer used to traverse the list */
  MPI_Request_Int *req;         /* The pointer used to store MPI_Request_Int */
  _USF_sysreq_node *sysreq1;    /* The pointer used to retrieve sysreq */
  int send_size;                /* The number of bytes sent */
  int sysreq_flag;              /* Shows if there was a matching sysreq */
  struct _USF_header *header;
  pthread_mutex_lock (&_USF_mut_req);
  curr = _USF_request_hd;

  /* Traverse and serve all the requests */
  while (curr != NULL) {
    req = curr->req;
    DB ("rank %d s_mpi_req fst type %d\n", rank, req->type);

    /* continue if already completed */
    if ((req->done) || (req->posted) || (req->active == 0)) {
      DB ("rank %d found completed or posted or inactive\n", rank, 1);
      curr = curr->next;
      continue;
    }
    if (req->type == _USF_SEND) {       /* type send */
      send_request_counter++;
      pthread_mutex_lock (&_USF_mut_sysreq);

      /* Check for a matching sysreq recv post */
      sysreq1 =
        _USF_sysreq_lookup (req->tag, req->peer, _USF_RECEIVE, req->comm);
      DB ("rank %d serve mpi _USF_SEND request.\n", rank, 1);
      if (sysreq1 == NULL) {    /* send send_post */

        /* if it is a ready mode message send
         * immediately */
        DB ("rank %d Send a send_post\n", rank, 1);
        i = _USF_send_header_pop ();
        if (i == -1) {
          printf ("running out of header\n");
          fflush (stdout);
          exit (1);
        }
        header = (struct _USF_header *) send_header_list[i];
        header->source = my_rank;
        header->dest = req->peer;
        header->tag = req->tag;
        header->comm = req->comm;
        header->type = _USF_SENDPOST;
        header->size = req->size;
        header->in_buf = NULL;
        header->send_seq_num = ++send_post_account[req->peer];
        _USF_gm_send_post (header, i);
```

```
      req->posted = 1;
      DB ("rank %d Sent a send_post!!!\n", rank, 1);
    } else {                  /* if there is a request send the message */
      send_post_account[req->peer]++;
      DB ("rank %d serve recv_post in MPI_serv request.\n", rank, 1);
      _USF_gm_handle_recv_post (sysreq1->source, rank, sysreq1->tag,
                                sysreq1->comm, sysreq1->size,
                                sysreq1->in_buf, req->buf, req->size, req);
      req->done = 1;
      pool_return (_USF_sysreq_node_pool, (pool_item *)
                   _USF_sysreq_remove (req->tag, req->peer, _USF_RECEIVE,
                                       req->comm));
    }
    pthread_mutex_unlock (&_USF_mut_sysreq);
  } else if (req->type == _USF_RECEIVE) {     /* type receive */
    recv_request_counter++;
    DB ("rank %d serve _USF_RECEIVE request\n", rank, 1);
    pthread_mutex_lock (&_USF_mut_sysreq);

    /* Check for a matching sysreq send post */
    DB ("rank %d, req->peer %d\n", rank, req->peer);
    sysreq1 =
      _USF_sysreq_lookup (req->tag, req->peer, _USF_SEND, req->comm);

    /* If there is remove it */
    if (sysreq1 == NULL) {
      DB ("rank %d, NOT FOUND SYSREQ\n", rank, 1);
      sysreq_flag = 0;
    } else {
      DB ("rank %d, FOUND SYSREQ src %d\n", rank, sysreq1->source);
      sysreq_flag = 1;
      req->peer = sysreq1->source;
      pool_return (_USF_sysreq_node_pool, (pool_item *)
                   _USF_sysreq_remove (req->tag, req->peer, _USF_SEND,
                                       req->comm));
      DB ("rank %d, After _USF_SEND Remove %d\n", rank, req->peer);
    }
    pthread_mutex_unlock (&_USF_mut_sysreq);
    if ((req->peer != MPI_ANY_SOURCE) || sysreq_flag) {
      DB ("rank %d, GO into send recv_post\n", rank, 1);
      i = _USF_send_header_pop ();
      if (i == -1) {
        printf ("running out of header\n");
        fflush (stdout);
        exit (1);
      }
      header = (struct _USF_header *) send_header_list[i];
      header->source = my_rank;
      header->dest = req->peer;
      header->tag = req->tag;
      header->comm = req->comm;
      header->type = _USF_RECVPOST;
      header->size = req->size;
      header->in_buf = req->buf;
      _USF_gm_send_post (header, i);
      DB ("rank %d, sent recv_post\n", rank, 1);
      req->posted = 1;
      recv_post_account[req->peer]++;
    }
  }
  curr = curr->next;
}
pthread_mutex_unlock (&_USF_mut_req);
```

```
  DB ("rank %d, unlock mut_req before exiting serve_mpi_req\n", my_rank, 1);
  return;
}


// gm_thread : Thread run function
// This is the body of listening thread function. This function only polling
// (receiving) 6-int headers/POSTs
void *
gm_thread ()
{
  gm_recv_event_t *event;
  int sender_rank = 0;
  int fast_recv;
  struct _USF_header *ptr, pre_header;
  int temp_counter;

  thread_recving = 1;
  blocking_recv = 0;

  //Waiting for incomming message
  while (thread_recving) {
    debug_counter++;
    //handle local recv request
    while (request_pending) {
      _USF_gm_serve_mpi_requests (my_rank, total_nodes);
      pthread_mutex_lock (&_USF_mut_gm_pending);
      request_pending--;
      pthread_mutex_unlock (&_USF_mut_gm_pending);
    }

    event = gm_receive (local_port);

    switch (gm_ntoh_u8 (event->recv.type)) {
    case GM_NO_RECV_EVENT:
      break;
    case GM_FAST_RECV_EVENT:
    case GM_FAST_PEER_RECV_EVENT:
    case GM_FAST_HIGH_RECV_EVENT:
    case GM_FAST_HIGH_PEER_RECV_EVENT:
      DB ("node %d gets gm header\n", my_rank, 1);

      ptr = (struct _USF_header *) (gm_ntohp (event->recv.message));

      gm_handle_header (ptr);

      gm_provide_receive_buffer (local_port, gm_ntohp (event->recv.buffer),
                                 _USF_MAXDATASIZE, GM_LOW_PRIORITY);
      break;

    case GM_RECV_EVENT:
    case GM_PEER_RECV_EVENT:
    case GM_HIGH_RECV_EVENT:
    case GM_HIGH_PEER_RECV_EVENT:

      //Handle incoming mssage
      printf ("message error\n");
      fflush (stdout);
      break;
    default:
      gm_unknown (local_port, event);
    }
  }
```

```
}


//-------------------------------------------------------------------------
// _USF_GM_recv_all : Receive incomming messages
// This has the same functionality as _USF_TCP_recv_all. It receives data
// from GM and determine if this is the data that user looks for.
// If not, it queues the data and do receive again until it receive the
// correct data.
//-------------------------------------------------------------------------
void
_USF_GM_recv_all (int source, int size, void *buf, char *msg)
{
}
#endif
```

B.19 init.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>
#include <pthread.h>
#include "mpi.h"

pool_t *_USF_mpi_request_int_pool;
pool_t *_USF_request_node_pool;
pool_t *_USF_sysreq_node_pool;

int
MPI_Init (int *argc, char ***argv)
{
  _USF_mpi_request_int_pool =
    pool_fill (16, -1, 8, -1, sizeof (MPI_Request_Int), (pool_malloc) malloc,
               (pool_free) free, NULL, NULL);
  _USF_request_node_pool =
    pool_fill (16, -1, 8, -1, sizeof (_USF_request_node),
               (pool_malloc) malloc, (pool_free) free, NULL, NULL);
  _USF_sysreq_node_pool =
    pool_fill (16, -1, 8, -1, sizeof (_USF_sysreq_node), (pool_malloc) malloc,
               (pool_free) free, NULL, NULL);

#ifdef _USF_MPI_TCP_

  _USF_TCP_init (argc, argv);
  _USF_TCP_thread_init ();
#endif

#ifdef _USF_MPI_GM_

  _USF_GM_init (argc, argv);
  _USF_GM_thread_init ();
```

```
#endif

  return 0;
}
```

B.20 iprobe.c

```
#include<stdio.h>
#include<stdlib.h>
#include"mpi.h"

int
MPI_Iprobe (int source, int tag, MPI_Comm comm,
            int *flag, MPI_Status * status)
{
  int rank;                     /* The rank of the current processor. */
  _USF_sysreq_node *sreq;

  MPI_Comm_rank (comm, &rank);

#ifdef DEBUG
  printf ("rank %d iProbe start\n", rank);
  fflush (stdout);
#endif
  sreq = _USF_sysreq_lookup (tag, source, _USF_SEND, comm);
  if (sreq == NULL) {
    *flag = 0;
  } else {
    *flag = 1;
    status->count = sreq->size;
    status->MPI_SOURCE = sreq->source;
    status->MPI_TAG = sreq->tag;
  }
#ifdef DEBUG
  printf ("rank %d iProbe end\n", rank);
  fflush (stdout);
#endif

  return 0;
}
```

B.21 mrun.c

```
// The mrun program is the equivalent of mpirun, it will initiate the
// calls to the processes.
// Currently only the number of processes and the name of the runnable
// file is going to be implemented.

#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include"debug.h"
#include"mpi.h"

// The arguments that will be used when calling the main function of
// the processes:
// 1 - The rank of the processor.
// 2 - The number of processors.
```

```
// 3 - The address of the current processor.

int
main (int argc, char **argv)
{

  char *cmd_str;                // Commandline string.
  int tmp;                      // The temporary variable.
  int i, j;                     // Loop variable.
  int nump = 1;                 // the number of processors.
  char *nump_str;
  int pos_called_file = 1;      // the pos of file name in argv[], default is 1
  char *cur_num;                // Used as a parameter to itostr function.
  char *my_name;                // The hostname that runs the program.
  pid_t *children;              // The pids of the child processes.
  int err;                      // The error value.
  char cwdbuf[256];             // Buffer for current working directory


  cur_num = (char *) malloc (8 * sizeof (char));
  cmd_str = (char *) malloc (256 * sizeof (char));
  my_name = (char *) malloc (128 * sizeof (char));

  // get the num of processors
  for (i = 1; i < argc; i++) {
    if (strcmp (argv[i], "-np") == 0) {
      pos_called_file = i + 2;
      nump = atoi (argv[i + 1]);
      nump_str = argv[i + 1];
    } else
      break;
  }

  // get the host name
  if (gethostname (my_name, 30) != 0)
    return Error (ER_HOSTNM);

  // call nump copies of programs
  if (nump == 1) {              // run with only one processor
    if ((err = _USF_make_conn_list (my_name, 1)) != 0)
      return err;
    // Start the program on processors
    memset (cmd_str, '\0', 80);
    if (strcmp (my_name, _USF_conn_list[0].hostname) != 0) {
      strcat (cmd_str, "rsh ");
      strcat (cmd_str, _USF_conn_list[0].hostname);
    }
    if (argv[pos_called_file][0] == '/') {      // A complete filename
      strcat (cmd_str, " ");
      strcat (cmd_str, argv[pos_called_file]);  // get the called file name
    } else {                    // Need to concat this current directory
      strcat (cmd_str, " ");
      strcat (cmd_str, getcwd (cwdbuf, 256));
      strcat (cmd_str, "/");
      strcat (cmd_str, argv[pos_called_file]);  // get the called file name
    }
    strcat (cmd_str, " 0 1 ");  // rank, number of procs
    strcat (cmd_str, my_name);  // hostname

    // get the parameters for called program
    for (j = pos_called_file + 1; j < argc; j++) {
      strcat (cmd_str, " ");
      strcat (cmd_str, argv[j]);
```

```
    }

    nump = 1;
    children = (pid_t *) malloc (nump * sizeof (pid_t));
    children[0] = fork ();
    if (!children[0]) {         //The child process
      if (system (cmd_str) != 0) {
        return Error (ER_FILENAME);
      }
      exit (0);
    }
  } else if (nump > 1) {        // specify the number of processors
    if ((err = _USF_make_conn_list (my_name, nump)) != 0)
      return err;

    children = (pid_t *) malloc (nump * sizeof (pid_t));
    for (i = 0; i < nump; i++) {        // Start the program on processors
      memset (cmd_str, '\0', 80);
      if (strcmp (my_name, _USF_conn_list[i].hostname) != 0) {
        strcat (cmd_str, "rsh ");
        strcat (cmd_str, _USF_conn_list[i].hostname);
      }

      if (argv[pos_called_file][0] == '/') {    // A complete filename
        strcat (cmd_str, " ");
        strcat (cmd_str, argv[pos_called_file]);        // get the called file name
      } else {                  // Need to concat this current directory
        strcat (cmd_str, " ");
        strcat (cmd_str, getcwd (cwdbuf, 256));
        strcat (cmd_str, "/");
        strcat (cmd_str, argv[pos_called_file]);        // get the called file name
      }

      strcat (cmd_str, " ");
      sprintf (cur_num, "%d", i);       // rank
      strcat (cmd_str, cur_num);
      strcat (cmd_str, " ");
      strcat (cmd_str, nump_str);       // number of procs
      strcat (cmd_str, " ");
      strcat (cmd_str, my_name);        // hostname

      for (j = pos_called_file + 1; j < argc; j++) {
        strcat (cmd_str, " ");
        strcat (cmd_str, argv[j]);
      }

      children[i] = fork ();
      if (!children[i]) {       //The child process
        DBG (("here: %s\n", cmd_str));
        if (system (cmd_str) != 0) {
          return Error (ER_FILENAME);
        }
        exit (0);
      }
    }
  } else {                      // Error
    return Error (ER_INCORRECT);
  }
  for (i = 0; i < nump; i++) {  // wait for the children
    waitpid (children[i], NULL, 0);
  }
```

```
  return 0;
}
```

B.22 new_funcs.c

```
#include "mpi.h"
#include "sys/time.h"

int
MPI_Type_size (MPI_Datatype datatype, int *size)
{

  *size = _USF_mpi_sizeof (datatype);

  return (MPI_SUCCESS);
}



double
MPI_Wtime ()
{
  double t1;
  struct timeval t2;
  gettimeofday (&t2, NULL);
  t1 = t2.tv_sec + t2.tv_usec / 1000000.0;
  return t1;
}


int
MPI_Error_string (int errorcode, char *string, int *resultlen)
{
  strcpy (string, "not implemented");

}


int
MPI_Reduce_scatter (void *sendbuf, void *recvbuf, int *recvcnts,
                    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
{


}

int
MPI_Abort (MPI_Comm comm, int errorcode)
{

  return MPI_ERR_UNKNOWN;
}


int
MPI_Comm_group (MPI_Comm comm, MPI_Group * group)
{
  int i;
  for (i = 0; i < _USF_num_of_comm; i++) {
    if (_USF_comm_list[i].comm == comm) {
      *group = _USF_comm_list[i].group;
```

```
      return 0;
    }
  }
  return -1;
}

int
MPI_Group_incl (MPI_Group old_group,    // in
                int new_group_size,     // in
                int ranks_in_old_group[],       // in
                MPI_Group * new_group   // out
  )
{
  int i, j;
  // More code needs to be added for groups more than 16
  if (_USF_group_max == _USF_num_of_group)
    return -1;
  *new_group = _USF_group_ref;
  _USF_group_list[_USF_num_of_group].group_no_old = old_group;
  _USF_group_list[_USF_num_of_group].group_no_local = _USF_group_ref;
  _USF_group_list[_USF_num_of_group].group_size = new_group_size;

  _USF_get_group_num (old_group, &j);


  for (i = 0; i < new_group_size; i++) {
    // set the rank in old group
    _USF_group_list[_USF_num_of_group].rank_old[i] = ranks_in_old_group[i];
    // set the rank in global group 0
    _USF_group_list[_USF_num_of_group].rank_global[i] =
      _USF_group_list[j].rank_global[ranks_in_old_group[i]];
    // set the rank in MPI_COMM_WORLD


  }

  _USF_group_ref++;             // increase the group reference number
  _USF_num_of_group++;          //
  return 0;
}

// find coressponding position in group_list
int
_USF_get_group_num (MPI_Group group, int *group_num)
{
  int i;
  for (i = 0; i < _USF_num_of_group; i++) {
    if (_USF_group_list[i].group_no_local == group) {
      *group_num = i;
      return 0;
    }
  }

  return 1;
}


int
MPI_Comm_create (MPI_Comm old_comm,     // in
                 MPI_Group new_group,   // in
                 MPI_Comm * new_comm    // out
  )
{
```

```
  int i, j, k, rank;
  // More code needs to be added for communicators more than 16
  if (_USF_comm_max == _USF_num_of_comm)
    return -1;
  *new_comm = _USF_comm_ref;

  MPI_Comm_rank (old_comm, &rank);

  _USF_get_group_num (new_group, &j);
  _USF_get_comm_num (old_comm, &k);

  _USF_comm_list[_USF_num_of_comm].group = new_group;
  _USF_comm_list[_USF_num_of_comm].comm = _USF_comm_ref;
  _USF_comm_list[_USF_num_of_comm].old_comm = old_comm;
  _USF_comm_list[_USF_num_of_comm].nump = _USF_group_list[j].group_size;

  _USF_comm_list[_USF_num_of_comm].rank = -1;
  // -1 means not belonging to this communicator

  for (i = 0; i < _USF_comm_list[_USF_num_of_comm].nump; i++) {
    // rank is the processor's rank in old communicator
    if (rank == _USF_group_list[j].rank_old[i])
      _USF_comm_list[_USF_num_of_comm].rank = i;

  }

  _USF_comm_ref++;
  _USF_num_of_comm++;

  return 0;
}

int
_USF_get_comm_num (MPI_Comm comm, int *comm_num)
{
  int i;
  for (i = 0; i < _USF_num_of_comm; i++) {
    if (_USF_comm_list[i].comm == comm) {
      *comm_num = i;
      return 0;
    }
  }
  *comm_num = -1;
  return 1;
}

int
MPI_Group_translate_ranks (MPI_Group group_a, int n, int *ranks_a,
                           MPI_Group group_b, int *ranks_b)
{
  return 0;
}


int
MPI_Comm_free (MPI_Comm * commp)
{
  return 0;
}


int
MPI_Comm_split (MPI_Comm comm, int color, int key, MPI_Comm * comm_out)
```

```
{
  int i, j, k, m, i1, i2;
  int output = 0;
  int comm_num;
  int size, rank;
  int *buf, *workingarray;
  int local[2];
  int tempColor;
  MPI_Comm tempComm;
  int *group_rank = NULL;
  MPI_Group old_group, new_group;

  MPI_Comm_size (comm, &size);
  MPI_Comm_rank (comm, &rank);

  _USF_get_comm_num (comm, &comm_num);

  MPI_Comm_group (comm, &old_group);

  buf = (int *) malloc (size * 2 * sizeof (int));
  workingarray = (int *) malloc (size * sizeof (int));

  if (buf == NULL) {
    printf ("Memory Error\n");
    return;
  }

  for (i = 0; i < size; i++) {
    workingarray[i] = 0;
  }

  local[0] = color;
  local[1] = key;

  group_rank = (int *) malloc (size * sizeof (int));

  MPI_Allgather (local, 2, MPI_INT, buf, 2, MPI_INT, comm);



  for (i = 0; i < size; i++) {
    if (workingarray[i] == 1)
      continue;
    tempColor = buf[2 * i];
    k = 0;
    for (j = i; j < size; j++) {
      if (tempColor == buf[j * 2] && workingarray[j] == 0) {
        group_rank[k] = j;
        k++;
        workingarray[j] = 1;
        if (j == rank)
          output = 1;
      }
    }

    for (i1 = 0; i1 < k - 1; i1++) {
      for (i2 = i1; i2 < k - 1; i2++)
        if (buf[group_rank[i2] * 2 + 1] > buf[group_rank[i2 + 1] * 2 + 1]) {
          m = group_rank[i2 + 1];
          group_rank[i2 + 1] = group_rank[i2];
          group_rank[i2] = m;
        }
    }
```

```
    MPI_Group_incl (old_group, k, group_rank, &new_group);
    if (output) {
      MPI_Comm_create (comm, new_group, comm_out);
      output = 0;
    } else
      MPI_Comm_create (comm, new_group, &tempComm);
  }

  if (color == MPI_UNDEFINED) {
    *comm_out = MPI_COMM_NULL;
  }

  return MPI_SUCCESS;

}



// check the communicator

int
_USF_check_comm (MPI_Comm comm, int *target)
{
  int rank, i;

  if (*target == MPI_ANY_SOURCE)
    return 0;

  MPI_Comm_rank (comm, &rank);
  if (rank == -1)
    return 1;
  else {
    _USF_get_comm_num (comm, &i);

    _USF_get_group_num (_USF_comm_list[i].group, &i);

    *target = _USF_group_list[i].rank_global[*target];

    return 0;
  }


}

int
MPI_Group_rank (MPI_Group group, int *rank)
{
  int group_num, global_rank, i;

  *rank = -1;
  if (_USF_get_comm_num (group, &group_num))
    return 1;
  MPI_Comm_rank (MPI_COMM_WORLD, &global_rank);

  for (i = 0; i < _USF_group_list[group_num].group_size; i++)
    if (_USF_group_list[group_num].rank_global[i] == global_rank)
      *rank = i;

  if (*rank == -1)
    return 1;
  return 0;
```

```
}


int
MPI_Group_size (MPI_Group group, int *size)
{
  int group_num;

  *size = -1;
  if (_USF_get_comm_num (group, &group_num))
    return 1;

  *size = _USF_group_list[group_num].group_size;

}
```

B.23 p2p_funcs.c

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<unistd.h>
#include"mpi.h"
#include "debug.h"

/* Be careful about inserting to head or tail in request list! */

#ifdef _USF_MPI_GM_             // addded for GM
extern pthread_mutex_t _USF_mut_gm_pending;
extern int request_pending;
extern int thread_recving;

int recv_call = 0;
int isend_call = 0;

unsigned int
address_hash (char *addr, int size)
{
  register unsigned int key;
  //unsigned int key;
  key = (unsigned int) addr;
  return ((key >> 14) * 2654435761 + size) % MAX_MEM_REG_ENTRY;
}


dma_mem_reg_table *mem_reg_table;

int
buf_not_reg (void *p, int size)
{
  int pos;
  dma_mem_reg_table *next = NULL;

  pos = address_hash (p, size);
  next = mem_reg_table + pos;

  if (mem_reg_table[pos].p == NULL) {
    mem_reg_table[pos].p = p;
    mem_reg_table[pos].size = size;
    mem_reg_table[pos].duplicate_link = NULL;
    return 1;
```

```
  } else if (mem_reg_table[pos].p == p && mem_reg_table[pos].size == size) {
    return 0;
  } else {
    while (next->duplicate_link != NULL) {
      if (next->duplicate_link->p == p & next->duplicate_link->size == size) {
        return 0;
      } else
        next = next->duplicate_link;
    }

    next->duplicate_link =
      (dma_mem_reg_table *) malloc (sizeof (dma_mem_reg_table));
    if (next->duplicate_link == NULL) {
      printf ("allocate memory error\n");
      fflush (stdout);
      exit (1);
    }
    next = next->duplicate_link;
    next->p = p;
    next->size = size;
    next->duplicate_link = NULL;
    return 1;
  }

  return 1;

}

#endif // end of new additions

inline void
_USF_SR_CONSTRUCTOR (int if_active, int if_blocking, int comm_type,
                     int peer_type, MPI_Request_Int ** req, void *buf,
                     _USF_request_node ** node, int count,
                     MPI_Datatype datatype, int tag, MPI_Comm comm)
{
  int rank;

  MPI_Comm_rank (comm, &rank);
  (*node) = (_USF_request_node *) pool_take (_USF_request_node_pool);
  (*node)->next = NULL;
  (*node)->back = NULL;
  (*req) = (MPI_Request_Int *) pool_take (_USF_mpi_request_int_pool);
  (*req)->active = if_active;
  (*req)->done = 0;
  (*req)->end_of_transmission = 0;
  (*req)->type = comm_type;
  (*req)->blocking = if_blocking;
  (*req)->posted = 0;
  (*req)->buf = buf;
  (*req)->size = count * _USF_mpi_sizeof (datatype);
#ifdef _USF_MPI_GM_
  if (buf != NULL && (*req)->size != 0 && buf_not_reg (buf, (*req)->size)) {

#ifdef DEBUG
    printf ("rank %d in constructor function to register memory\n", rank);
    fflush (stdout);
#endif

    gm_register_memory (local_port, buf, (*req)->size);
  }
#endif
  (*req)->tag = tag;
```

```
  (*req)->peer = peer_type;
  (*req)->nchunk = 0;
  (*req)->comm = comm;
  (*req)->node = (*node);
  (*node)->req = (*req);
}

inline void
_USF_SR_MULTIPLE_DEC (MPI_Request_Int * req, int type)
{
#ifdef _USF_MPI_TCP_
  if (type == _USF_SEND) {
    if (req->size > _USF_MAXDATASIZE)
      req->multiple = 1;
    else
      req->multiple = 0;
  } else {
    req->multiple = 0;
  }
#endif
#ifdef _USF_MPI_GM_
  req->multiple = 0;
#endif

}
inline void
_USF_SR_ADD_TOQ (_USF_request_node * node)
{
  pthread_mutex_lock (&_USF_mut_req);
  _USF_request_instl (node);
  pthread_mutex_unlock (&_USF_mut_req);
}
inline void
_USF_SR_NOTIFY (void)
{
  int rank, nump;
#ifndef SINGLE_THREAD
  // multiple threaded mode, try to notify communication thread
#ifdef _USF_MPI_TCP_
  int pipei = 0;
  write (_USF_s_pipe[1], &pipei, sizeof (int));
#else
  pthread_mutex_lock (&_USF_mut_gm_pending);
  request_pending++;
  pthread_mutex_unlock (&_USF_mut_gm_pending);
#endif

#else
  // single threaded mode
#ifdef _USF_MPI_TCP_
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &nump);
  _USF_serve_mpi_requests (rank, nump);
#endif

#ifdef _USF_MPI_GM_
  request_pending++;
  thread_recving = 1;
  gm_thread ();
#endif

#endif
```

```
}

inline void
_USF_SR_BLOCK (_USF_request_node * node)
{

  int nump;                     /* # of procs in COMM_WORLD */
  int rank;                     /* the rank in COMM_WORLD */


  MPI_Comm_size (MPI_COMM_WORLD, &nump);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);



#ifndef SINGLE_THREAD
  pthread_mutex_lock (&_USF_mut_block);
  if (node->req->end_of_transmission == 0)
    pthread_cond_wait (&_USF_cond_block, &_USF_mut_block);
  pthread_mutex_unlock (&_USF_mut_block);
#else

#ifdef _USF_MPI_TCP_
  int i;                        /* Loop variable */
  fd_set set;                   /* the set that is used for select */
  int max;                      /* the maximum file descriptor */
  int finished = 0;             /* The loop conditional */
  int nu;


  while (!node->req->end_of_transmission) {


    max = _USF_TCP_prepare_set (&set);
    if ((nu = pselect (max + 1, &set, NULL, NULL, NULL, NULL)) == -1)
      Error (ERF_SELECT);
    DBG (("rank %d, woken up from select! sel %d \n", rank, nu));
    for (i = 0; i < nump; i++) {
      if (i == rank)
        continue;
      if (FD_ISSET (_USF_conn_list[i].sd, &set)) {
        DBG (("rank %d handling message %d start\n", rank, i));
        _USF_handle_msg (i);
        DBG (("rank %d handling message %d end\n", rank, i));
      }
    }
  }
#endif

#ifdef _USF_MPI_GM_
  while (!node->req->end_of_transmission) {
    thread_recving = 1;
    gm_thread ();
  }
#endif

#endif

  pthread_mutex_lock (&_USF_mut_req);
  _USF_request_remove (node);
  pthread_mutex_unlock (&_USF_mut_req);

}
```

```
int
MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest,
          int tag, MPI_Comm comm)
{
  MPI_Request_Int *req;
  _USF_request_node *node;
  int i;
  _USF_MPI_struct_type *stype;
  int rank;

  if (datatype < MPI_Primitive_Type) {
    if (_USF_check_comm (comm, &dest)) {
      return 1;
    }
    _USF_SR_CONSTRUCTOR (1, 1, _USF_SEND, dest, &req, buf, &node, count,
                         datatype, tag, comm);
    _USF_SR_MULTIPLE_DEC (req, _USF_SEND);
    _USF_SR_ADD_TOQ (node);
    _USF_SR_NOTIFY ();
    _USF_SR_BLOCK (node);
  } else {
    if (_USF_DDType_Get_Type (datatype, &stype) == MPI_SUCCESS)
      MPI_Comm_rank (comm, &rank);
    for (i = 0; i < stype->count; i++) {
      MPI_Send ((void *) (buf + stype->disp[i]), stype->block_lengths[i],
                stype->typelist[i], dest, tag, comm);
    }
  }
  return 0;
}


int
MPI_Rsend (void *buf, int count, MPI_Datatype datatype, int dest,
           int tag, MPI_Comm comm)
{
  MPI_Request_Int *req;
  _USF_request_node *node;
  _USF_MPI_struct_type *stype;
  int rank;
  int i;

  if (datatype < MPI_Primitive_Type) {
    if (_USF_check_comm (comm, &dest)) {
      return 1;
    }
    _USF_SR_CONSTRUCTOR (1, 1, _USF_SEND, dest, &req, buf, &node, count,
                         datatype, tag, comm);
    _USF_SR_MULTIPLE_DEC (req, _USF_SEND);
    _USF_SR_ADD_TOQ (node);
    _USF_SR_NOTIFY ();
    _USF_SR_BLOCK (node);
  } else {
    if (_USF_DDType_Get_Type (datatype, &stype) == MPI_SUCCESS)
      MPI_Comm_rank (comm, &rank);
    for (i = 0; i < stype->count; i++) {
      MPI_Rsend ((void *) (buf + stype->disp[i]), stype->block_lengths[i],
                 stype->typelist[i], dest, tag, comm);
    }
  }
  return 0;
```

```
}

int
MPI_Ssend (void *buf, int count, MPI_Datatype datatype, int dest,
           int tag, MPI_Comm comm)
{
  MPI_Request_Int *req;
  _USF_request_node *node;
  _USF_MPI_struct_type *stype;
  int rank;
  int i;

  if (datatype < MPI_Primitive_Type) {
    if (_USF_check_comm (comm, &dest)) {
      return 1;
    }
    _USF_SR_CONSTRUCTOR (1, 1, _USF_SEND, dest, &req, buf, &node, count,
                         datatype, tag, comm);
    _USF_SR_MULTIPLE_DEC (req, _USF_SEND);
    _USF_SR_ADD_TOQ (node);
    _USF_SR_NOTIFY ();
    _USF_SR_BLOCK (node);
  } else {
    if (_USF_DDType_Get_Type (datatype, &stype) == MPI_SUCCESS)
      MPI_Comm_rank (comm, &rank);
    for (i = 0; i < stype->count; i++) {
      MPI_Ssend ((void *) (buf + stype->disp[i]), stype->block_lengths[i],
                 stype->typelist[i], dest, tag, comm);
    }
  }
  return 0;
}

int
MPI_Isend (void *buf, int count, MPI_Datatype datatype, int dest,
           int tag, MPI_Comm comm, MPI_Request * request)
{
  _USF_request_node *node;
  _USF_MPI_struct_type *stype;
  int rank;
  int i;


  if (datatype < MPI_Primitive_Type) {
    if (_USF_check_comm (comm, &dest)) {
      return 1;
    }
    _USF_SR_CONSTRUCTOR (1, 0, _USF_SEND, dest, request, buf, &node, count,
                         datatype, tag, comm);
    _USF_SR_MULTIPLE_DEC ((*request), _USF_SEND);
    _USF_SR_ADD_TOQ (node);
    _USF_SR_NOTIFY ();
  } else {
    if (_USF_DDType_Get_Type (datatype, &stype) == MPI_SUCCESS)
      MPI_Comm_rank (comm, &rank);
    for (i = 0; i < stype->count; i++) {
      MPI_Isend ((void *) (buf + stype->disp[i]), stype->block_lengths[i],
                 stype->typelist[i], dest, tag, comm, request);
    }
  }
  return 0;
}
```

```
int
MPI_Irsend (void *buf, int count, MPI_Datatype datatype, int dest,
            int tag, MPI_Comm comm, MPI_Request * request)
{
  _USF_request_node *node;
  _USF_MPI_struct_type *stype;
  int rank;
  int i;

  if (datatype < MPI_Primitive_Type) {
    if (_USF_check_comm (comm, &dest)) {
      return 1;
    }
    _USF_SR_CONSTRUCTOR (1, 0, _USF_SEND, dest, request, buf, &node, count,
                         datatype, tag, comm);
    _USF_SR_MULTIPLE_DEC ((*request), _USF_SEND);
    _USF_SR_ADD_TOQ (node);
    _USF_SR_NOTIFY ();
  } else {
    if (_USF_DDType_Get_Type (datatype, &stype) == MPI_SUCCESS)
      MPI_Comm_rank (comm, &rank);
    for (i = 0; i < stype->count; i++) {
      MPI_Irsend ((void *) (buf + stype->disp[i]), stype->block_lengths[i],
                  stype->typelist[i], dest, tag, comm, request);
    }
  }
  return 0;
}

int
MPI_Issend (void *buf, int count, MPI_Datatype datatype, int dest,
            int tag, MPI_Comm comm, MPI_Request * request)
{
  _USF_request_node *node;
  _USF_MPI_struct_type *stype;
  int rank;
  int i;

  if (datatype < MPI_Primitive_Type) {
    if (_USF_check_comm (comm, &dest)) {
      return 1;
    }
    _USF_SR_CONSTRUCTOR (1, 0, _USF_SEND, dest, request, buf, &node, count,
                         datatype, tag, comm);
    _USF_SR_MULTIPLE_DEC ((*request), _USF_SEND);
    _USF_SR_ADD_TOQ (node);
    _USF_SR_NOTIFY ();
  } else {
    if (_USF_DDType_Get_Type (datatype, &stype) == MPI_SUCCESS)
      MPI_Comm_rank (comm, &rank);
    for (i = 0; i < stype->count; i++) {
      MPI_Issend ((void *) (buf + stype->disp[i]), stype->block_lengths[i],
                  stype->typelist[i], dest, tag, comm, request);
    }
  }
  return 0;
}

int
MPI_Send_init (void *buf, int count, MPI_Datatype datatype, int dest,
               int tag, MPI_Comm comm, MPI_Request * request)
{
  _USF_request_node *node;
```

```
  if (_USF_check_comm (comm, &dest)) {
    return 1;
  }
  _USF_SR_CONSTRUCTOR (0, 0, _USF_SEND, dest, request, buf, &node, count,
                       datatype, tag, comm);
  _USF_SR_MULTIPLE_DEC ((*request), _USF_SEND);
  _USF_SR_ADD_TOQ (node);
  return 0;
}

int
MPI_Rsend_init (void *buf, int count, MPI_Datatype datatype, int dest,
                int tag, MPI_Comm comm, MPI_Request * request)
{
  _USF_request_node *node;

  if (_USF_check_comm (comm, &dest)) {
    return 1;
  }
  _USF_SR_CONSTRUCTOR (0, 0, _USF_SEND, dest, request, buf, &node, count,
                       datatype, tag, comm);
  _USF_SR_MULTIPLE_DEC ((*request), _USF_SEND);
  _USF_SR_ADD_TOQ (node);
  return 0;
}

int
MPI_Ssend_init (void *buf, int count, MPI_Datatype datatype, int dest,
                int tag, MPI_Comm comm, MPI_Request * request)
{
  _USF_request_node *node;

  if (_USF_check_comm (comm, &dest)) {
    return 1;
  }
  _USF_SR_CONSTRUCTOR (0, 0, _USF_SEND, dest, request, buf, &node, count,
                       datatype, tag, comm);
  _USF_SR_MULTIPLE_DEC ((*request), _USF_SEND);
  _USF_SR_ADD_TOQ (node);
  return 0;
}

int
MPI_Recv_init (void *buf, int count, MPI_Datatype datatype, int source,
               int tag, MPI_Comm comm, MPI_Request * request)
{
  _USF_request_node *node;

  if (_USF_check_comm (comm, &source)) {
    return 1;
  }
  _USF_SR_CONSTRUCTOR (0, 0, _USF_RECEIVE, source, request, buf, &node,
                       count, datatype, tag, comm);
  _USF_SR_MULTIPLE_DEC ((*request), _USF_RECEIVE);
  _USF_SR_ADD_TOQ (node);
  return 0;
}

int
MPI_Irecv (void *buf, int count, MPI_Datatype datatype, int source,
           int tag, MPI_Comm comm, MPI_Request * request)
{
```

```
  _USF_request_node *node;
  int i;
  _USF_MPI_struct_type *stype;

  if (datatype < MPI_Primitive_Type) {
    if (_USF_check_comm (comm, &source)) {
      return 1;
    }
    _USF_SR_CONSTRUCTOR (1, 0, _USF_RECEIVE, source, request, buf, &node,
                         count, datatype, tag, comm);
    _USF_SR_MULTIPLE_DEC ((*request), _USF_RECEIVE);
    _USF_SR_ADD_TOQ (node);
    _USF_SR_NOTIFY ();
  } else {
    _USF_DDType_Get_Type (datatype, &stype);
    for (i = 0; i < stype->count; i++) {
      MPI_Irecv ((void *) buf + stype->disp[i], stype->block_lengths[i],
                 stype->typelist[i], source, tag, comm, request);
    }
  }
  return 0;
}


int
MPI_Recv (void *buf, int count, MPI_Datatype datatype, int source,
          int tag, MPI_Comm comm, MPI_Status * status)
{
  MPI_Request_Int *req;
  _USF_request_node *node;
  int i;

  int size;

  _USF_MPI_struct_type *stype;
  if (datatype < MPI_Primitive_Type) {
    if (_USF_check_comm (comm, &source)) {
      return 1;
    }
    _USF_SR_CONSTRUCTOR (1, 1, _USF_RECEIVE, source, &req, buf, &node,
                         count, datatype, tag, comm);
    _USF_SR_MULTIPLE_DEC (req, _USF_RECEIVE);
    _USF_SR_ADD_TOQ (node);
    _USF_SR_NOTIFY ();
    _USF_SR_BLOCK (node);
  } else {
    _USF_DDType_Get_Type (datatype, &stype);
    for (i = 0; i < stype->count; i++) {
      MPI_Recv ((void *) buf + stype->disp[i], stype->block_lengths[i],
                stype->typelist[i], source, tag, comm, status);
    }
  }
  return 0;
}
```

B.24 pool.c

```
/*
 * pool.c
 *
 * Implementation of the pool interface for object allocation can caching
```

```
 *
 * Alex Fedosov
 * 2001-10-23 Gregory D. Benson
 *   name change: Pool to pool_t
 *   removed locking
 *   added user-supplied init and destroy callbacks
 */

#include <assert.h>
#include <limits.h>

#include "debug.h"
#include "pool.h"

pool_t *
pool_fill (int initial_size, int max_size, int alloc_incr,
           int free_threshold, unsigned int item_size,
           pool_malloc u_malloc, pool_free u_free,
           pool_func u_init, pool_func u_destroy)
{
  int i;
  pool_t *p;
  pool_item *current = NULL, *prev = NULL;

  DBG (("[pool_fill] initial: %d max: %d incr: %d free: %d\n",
        initial_size, max_size, alloc_incr, free_threshold));

  /* check parameters */
  if (initial_size <= 0 || max_size == 0 || max_size < -1 ||
      alloc_incr < -1 || free_threshold < -1)
    return NULL;

  /* create a pool */
  p = (pool_t *) u_malloc (sizeof (pool_t));
  assert (p != NULL);

  p->malloc = u_malloc;
  p->free = u_free;
  p->init = u_init;
  p->destroy = u_destroy;

  /* what happens when a negative number is cast to unsigned? */
  p->capacity = (max_size < 0) ? INT_MAX : max_size;
  p->free_limit = (free_threshold < 0) ? INT_MAX : free_threshold;
  p->alloc_incr = (alloc_incr < 0) ? initial_size : alloc_incr;
  p->item_size = item_size;

  /* allocate the initial elements */
  prev = NULL;
  for (i = 0; i < initial_size; i++) {
    current = (pool_item *) u_malloc (item_size);
    assert (current != NULL);
    if (u_init != NULL) {
      u_init (current);
    }
    current->next = prev;
    prev = current;
  }
  p->head = current;

  DBG (("[pool_fill] successfully added %d items\n", i));

  /* update the size and number of available elements */
```

```
  p->size = initial_size;
  p->available = initial_size;

  DBG (("[pool_fill] size: %d available: %d capacity: %d free limit %d alloc incr: %d\n \
head: 0x%X\n", p->size, p->available, p->capacity,
        p->free_limit, p->alloc_incr, (int) p->head));

  return p;
}

pool_item *
pool_take (pool_t * p)
{
  int i;
  pool_item *e = NULL, *current = NULL, *prev = NULL;

  /* if pool is empty, add more elements */
  if (pool_empty (p)) {
    DBG (("[pool_take] pool exhausted\n"));
    DBG (("[pool_take] size: %d capacity: %d\n", p->size, p->capacity));

    /* see if we are below max capacity */
    if (p->size >= p->capacity) {
      DBG (("[pool_take] maximum capacity reached\n"));
      return NULL;
    }

    DBG (("[pool_take] re-filling the pool\n"));

    /* see if we are allowed to increase size */
    if (p->alloc_incr <= 0) {
      DBG (("[pool_take] not allowed to increase size\n"));
      return NULL;
    }

    prev = p->head;
    for (i = 0; i < p->alloc_incr; i++) {
      current = (pool_item *) p->malloc (p->item_size);
      if (p->init != NULL) {
        p->init (current);
      }
      current->next = prev;
      prev = current;
    }
    p->head = current;

    /* update size and availability */
    p->size += p->alloc_incr;
    p->available = p->alloc_incr;

    DBG (("[pool_take] added %d items, new size is %d\n", i, p->size));
  }

  /* now just take the first element and update the head pointer */
  e = p->head;
  p->head = p->head->next;

  /* update number of available elements */
  p->available--;

  DBG (("[pool_take] obtained available item, %d now available\n",
        p->available));
```

```
  /* don't know why, just felt like it. */
  e->next = NULL;

  return e;
}

void
pool_return (pool_t * p, pool_item * e)
{
  int i, extras;
  pool_item *current, *prev;

  /* add element back to the list */
  e->next = p->head;
  p->head = e;

  /* update number of available elements */
  p->available++;

  DBG (("[pool_return] returned an item to the pool, %d now available\n",
        p->available));

  /* see if anything needs deallocating */
  if ((extras = (p->available - p->free_limit)) > 0) {
    DBG (("[pool_return] pool has %d extra items\n", extras));

    /* need to deallocate that many elements */
    current = p->head;
    for (i = 0; i < extras; i++) {
      prev = current;
      current = current->next;
      if (p->destroy != NULL) {
        p->destroy (prev);
      }
      p->free (prev);
    }

    /* update head pointer */
    p->head = current;

    /* update pool size and availability */
    p->size -= extras;
    p->available -= extras;

    DBG (("[pool_return] deallocated %d items, new size is \
%d\n", i, p->size));
  }
}

void
pool_drain (pool_t * p)
{
  int i;
  pool_item *current, *prev;
  pool_free u_free;

  /* see if pool is already empty */
  if (!pool_empty (p)) {
    DBG (("[pool_drain] pool not empty, has %d items\n", p->available));

    /* go through the list and free all elements */
    current = p->head;
    for (i = 0; i < p->available; i++) {
```

```
      prev = current;
      current = current->next;
      if (p->destroy != NULL) {
        p->destroy (prev);
      }
      p->free (prev);
    }

    DBG (("[pool_drain] pool drained, discarded %d items\n", i));
  }

  /* pool is now (or already) empty, free its memory */
  u_free = p->free;
  u_free (p);

  DBG (("[pool_drain] no more pool.\n"));
}

int
pool_empty (pool_t * p)
{
  return (p->head == NULL);
}
```

B.25 probe.c

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include"mpi.h"
#include"debug.h"

extern int thread_recving;

int
MPI_Probe (int source, int tag, MPI_Comm comm, MPI_Status * status)
{
  int flag, rank, nump;         /* Flag value used for Iprobe */
  MPI_Request_Int *req;         /* The pointer used to init the MPI Request */
  _USF_request_node *node;      /* The pointer used to store
                                 * the MPI Request in.*/

  node = (_USF_request_node *) pool_take (_USF_request_node_pool);
  req = (MPI_Request_Int *) pool_take (_USF_mpi_request_int_pool);
  req->active = 1;
  req->done = 0;
  req->end_of_transmission = 0;
  req->type = _USF_PROBE;
  req->blocking = 1;
  req->posted = 0;
  req->tag = MPI_ANY_TAG;
  req->peer = source;
  req->comm = comm;
  req->multiple = 0;
  req->nchunk = 0;
  req->node = node;
  node->req = req;

  MPI_Comm_size (MPI_COMM_WORLD, &nump);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
```

```
  pthread_mutex_lock (&_USF_mut_req);
  _USF_request_inshd (node);
  pthread_mutex_unlock (&_USF_mut_req);

#ifdef DEBUG
  printf ("rank %d Probe start\n", _USF_comm_list[0].rank);
  fflush (stdout);
#endif
  flag = 0;
  while (!flag) {
    MPI_Iprobe (source, tag, comm, &flag, status);
    if (!flag) {
#ifndef SINGLE_THREAD
      pthread_mutex_lock (&_USF_mut_block);
      if (req->end_of_transmission == 0)
        pthread_cond_wait (&_USF_cond_block, &_USF_mut_block);
      pthread_mutex_unlock (&_USF_mut_block);
#else

#ifdef _USF_MPI_TCP_
      int i;                    /* Loop variable */

      fd_set set;               /* the set that is used for select */
      int max;                  /* the maximum file descriptor */
      int finished = 0;         /* The loop conditional */
      int nu;




      while (!(req->end_of_transmission)) {

        max = _USF_TCP_prepare_set (&set);
        if ((nu = pselect (max + 1, &set, NULL, NULL, NULL, NULL)) == -1)
          Error (ERF_SELECT);
        DBG (("rank %d, woken up from select! sel %d \n", rank, nu));
        for (i = 0; i < nump; i++) {
          if (i == rank)
            continue;
          if (FD_ISSET (_USF_conn_list[i].sd, &set)) {
            DBG (("rank %d handling message %d start\n", rank, i));
            _USF_handle_msg (i);
            DBG (("rank %d handling message %d end\n", rank, i));
          }
        }
      }
#endif

#ifdef _USF_MPI_GM_
      while (!(req->end_of_transmission)) {
        thread_recving = 1;
        gm_thread ();
      }
#endif

#endif
    }

  }
  pthread_mutex_lock (&_USF_mut_req);
  _USF_request_remove (node);
  pthread_mutex_unlock (&_USF_mut_req);
```

```
#ifdef DEBUG
  printf ("rank %d Probe end\n", _USF_comm_list[0].rank);
  fflush (stdout);
#endif
  return 0;
}
```

B.26 reduce.c

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<unistd.h>
#include"errno.h"
#include"mpi.h"

#define _USF_SOLVER_MAX_EQ(type1)                 \
      for(i=0;i<count;i++) {                      \
              if (p2.type1[i] > p1.type1[i])      \
                      p1.type1[i] = p2.type1[i];  \
      }
#define _USF_SOLVER_MIN_EQ(type1)                 \
      for(i=0;i<count;i++) {                      \
              if (p2.type1[i] < p1.type1[i])      \
                      p1.type1[i] = p2.type1[i];  \
      }
#define _USF_SOLVER_SUM_EQ(type1)                 \
      for(i=0;i<count;i++) {                      \
              p1.type1[i] += p2.type1[i];         \
      }
#define _USF_SOLVER_PROD_EQ(type1)                \
      for(i=0;i<count;i++) {                      \
              p1.type1[i] *= p2.type1[i];         \
      }
#define _USF_SOLVER_LAND_EQ(type1)                          \
      for(i=0;i<count;i++) {                                \
              p1.type1[i] = (p1.type1[i] && p2.type1[i]);   \
      }
#define _USF_SOLVER_BAND_EQ(type1)                         \
      for(i=0;i<count;i++) {                               \
              p1.type1[i] = (p1.type1[i] & p2.type1[i]);   \
      }
#define _USF_SOLVER_LOR_EQ(type1)                           \
      for(i=0;i<count;i++) {                                \
              p1.type1[i] = (p1.type1[i] || p2.type1[i]);   \
      }
#define _USF_SOLVER_BOR_EQ(type1)                          \
      for(i=0;i<count;i++) {                               \
              p1.type1[i] = (p1.type1[i] | p2.type1[i]);   \
      }
#define _USF_SOLVER_LXOR_EQ(type1)                            \
      for(i=0;i<count;i++) {                                  \
              p1.type1[i] = (p1.type1[i] && p2.type1[i]) ||   \
                         ((!p1.type1[i]) && (!p2.type1[i]));  \
       }
#define _USF_SOLVER_BXOR_EQ(type1)                 \
      for(i=0;i<count;i++) {                       \
              p1.type1[i] = (p1.type1[i] ^ p2.type1[i]);   \
      }
#define _USF_SOLVER_MINLOC_EQ(type1)                 \
      for(i=0;i<count;i++) {                      \
```

```
                if (p2.type1[i] > p1.type1[i])      \
                        p1.type1[i] = p2.type1[i];  \
                        }
#define _USF_SOLVER_MAXLOC_EQ(type1)                  \
        for(i=0;i<count;i++) {                      \
                if (p2.type1[i] > p1.type1[i])      \
                        p1.type1[i] = p2.type1[i];  \
                        }


#define _USF_SOLVER(type1, type2, equation)   \
        p1.type1 = type2 hostbuf;                   \
        p2.type1 = type2 peerbuf;                   \
        equation(type1);                            \



typedef union _USF_reduce_union
{

  short *shortp;
  int *intp;
  long *longp;
  unsigned short *ushortp;
  unsigned *usignedp;
  unsigned long *ulongp;
  float *floatp;
  double *doublep;
  long double *ldoublep;
  long long *llp;
}
_USF_reduce_union;

#define _USF_SOLVER_GENERAL(funname, solver_eq)                             \
int funname (void *hostbuf, void *peerbuf, int count,                       \
                    MPI_Datatype datatype)                                  \
{                                                                           \
        int i;                                                              \
        _USF_reduce_union p1, p2;                                           \
        if(datatype > MPI_LONG_DOUBLE_INT) {                                \
                printf("Unexpected Datatype!\n");                           \
                exit(1);                                                    \
        }                                                                   \
        switch(datatype) {                                                  \
        case MPI_SHORT:                                                     \
                _USF_SOLVER(shortp, (short *), solver_eq);                  \
                break;                                                      \
        case MPI_INT:                                                       \
                _USF_SOLVER(intp, (int *), solver_eq);                      \
                break;                                                      \
        case MPI_LONG:                                                      \
                _USF_SOLVER(longp, (long *), solver_eq);                    \
                break;                                                      \
        case MPI_UNSIGNED_SHORT:                                            \
                _USF_SOLVER(ushortp, (unsigned short*), solver_eq);         \
                break;                                                      \
        case MPI_UNSIGNED:                                                  \
                _USF_SOLVER(usignedp, (unsigned *), solver_eq);             \
                break;                                                      \
        case MPI_UNSIGNED_LONG:                                             \
                _USF_SOLVER(ulongp, (unsigned long*), solver_eq);           \
                break;                                                      \
        case MPI_FLOAT:                                                     \
```

```
                _USF_BINARY_GEN(_USF_SOLVER(floatp, (float *), solver_eq));      \
                break;                                                           \
        case MPI_DOUBLE:                                                         \
                _USF_BINARY_GEN(_USF_SOLVER(doublep, (double *), solver_eq));    \
                break;                                                           \
        case MPI_LONG_DOUBLE:                                                    \
                _USF_BINARY_GEN(_USF_SOLVER(ldoublep, (long double*), solver_eq));\
                break;                                                           \
        case MPI_LONG_LONG:                                                      \
                _USF_SOLVER(llp, (long long*), solver_eq);                       \
                break;                                                           \
        default:                                                                 \
                printf("This operation isn't defined for this datatype!\n");     \
                exit(1);                                                         \
        }                                                                        \
        return 0;                                                                \
}

#define _USF_BINARY_GEN(x) x
_USF_SOLVER_GENERAL (_USF_solve_max, _USF_SOLVER_MAX_EQ);
_USF_SOLVER_GENERAL (_USF_solve_min, _USF_SOLVER_MIN_EQ);
_USF_SOLVER_GENERAL (_USF_solve_sum, _USF_SOLVER_SUM_EQ);
_USF_SOLVER_GENERAL (_USF_solve_prod, _USF_SOLVER_PROD_EQ);
_USF_SOLVER_GENERAL (_USF_solve_land, _USF_SOLVER_LAND_EQ);
_USF_SOLVER_GENERAL (_USF_solve_lor, _USF_SOLVER_LOR_EQ);
_USF_SOLVER_GENERAL (_USF_solve_lxor, _USF_SOLVER_LXOR_EQ);

#undef _USF_BINARY_GEN(x)
#define _USF_BINARY_GEN(x)
_USF_SOLVER_GENERAL (_USF_solve_bxor, _USF_SOLVER_BXOR_EQ);
_USF_SOLVER_GENERAL (_USF_solve_band, _USF_SOLVER_BAND_EQ);
_USF_SOLVER_GENERAL (_USF_solve_bor, _USF_SOLVER_BOR_EQ);

typedef union _USF_reduce_double_union
{

  _USF_FLOAT_INT *ufip;
  _USF_DOUBLE_INT *udip;
  _USF_LONG_INT *ulip;
  _USF_SHORT_INT *usip;
  _USF_2INT *uiip;
  _USF_LONG_DOUBLE_INT *uldip;
}
_USF_reduce_double_union;


int
_USF_solve_2loc (void *hostbuf, void *peerbuf, int count,
                 MPI_Datatype datatype, int min)
{
  int i;

  _USF_reduce_double_union p1, p2;

  if (datatype > MPI_LONG_DOUBLE_INT) {
    printf ("Unexpected Datatype!\n");
    exit (1);
  }
  switch (datatype) {
  case MPI_FLOAT_INT:
    p1.ufip = (_USF_FLOAT_INT *) hostbuf;
    p2.ufip = (_USF_FLOAT_INT *) peerbuf;
    for (i = 0; i < count; i++) {
```

```
    if (min) {
      if (p2.ufip[i].fst < p1.ufip[i].fst) {
        p1.ufip[i].fst = p2.ufip[i].fst;
        p1.ufip[i].sec = p2.ufip[i].sec;
      }
    } else {
      if (p2.ufip[i].fst > p1.ufip[i].fst) {
        p1.ufip[i].fst = p2.ufip[i].fst;
        p1.ufip[i].sec = p2.ufip[i].sec;
      }
    }
  }
  break;
case MPI_DOUBLE_INT:
  p1.udip = (_USF_DOUBLE_INT *) hostbuf;
  p2.udip = (_USF_DOUBLE_INT *) peerbuf;
  for (i = 0; i < count; i++) {
    if (min) {
      if (p2.udip[i].fst < p1.udip[i].fst) {
        p1.udip[i].fst = p2.udip[i].fst;
        p1.udip[i].sec = p2.udip[i].sec;
      }
    } else {
      if (p2.udip[i].fst > p1.udip[i].fst) {
        p1.udip[i].fst = p2.udip[i].fst;
        p1.udip[i].sec = p2.udip[i].sec;
      }
    }
  }
  break;
case MPI_LONG_INT:
  p1.ulip = (_USF_LONG_INT *) hostbuf;
  p2.ulip = (_USF_LONG_INT *) peerbuf;
  for (i = 0; i < count; i++) {
    if (min) {
      if (p2.ulip[i].fst < p1.ulip[i].fst) {
        p1.ulip[i].fst = p2.ulip[i].fst;
        p1.ulip[i].sec = p2.ulip[i].sec;
      }
    } else {
      if (p2.ulip[i].fst > p1.ulip[i].fst) {
        p1.ulip[i].fst = p2.ulip[i].fst;
        p1.ulip[i].sec = p2.ulip[i].sec;
      }
    }
  }
  break;
case MPI_SHORT_INT:
  p1.usip = (_USF_SHORT_INT *) hostbuf;
  p2.usip = (_USF_SHORT_INT *) peerbuf;
  for (i = 0; i < count; i++) {
    if (min) {
      if (p2.usip[i].fst < p1.usip[i].fst) {
        p1.usip[i].fst = p2.usip[i].fst;
        p1.usip[i].sec = p2.usip[i].sec;
      }
    } else {
      if (p2.usip[i].fst > p1.usip[i].fst) {
        p1.usip[i].fst = p2.usip[i].fst;
        p1.usip[i].sec = p2.usip[i].sec;
      }
    }
  }
```

```
    break;
  case MPI_2INT:
    p1.uiip = (_USF_2INT *) hostbuf;
    p2.uiip = (_USF_2INT *) peerbuf;
    for (i = 0; i < count; i++) {
      if (min) {
        if (p2.uiip[i].fst < p1.uiip[i].fst) {
          p1.uiip[i].fst = p2.uiip[i].fst;
          p1.uiip[i].sec = p2.uiip[i].sec;
        }
      } else {
        if (p2.uiip[i].fst > p1.uiip[i].fst) {
          p1.uiip[i].fst = p2.uiip[i].fst;
          p1.uiip[i].sec = p2.uiip[i].sec;
        }
      }
    }
    break;
  case MPI_LONG_DOUBLE_INT:
    p1.uldip = (_USF_LONG_DOUBLE_INT *) hostbuf;
    p2.uldip = (_USF_LONG_DOUBLE_INT *) peerbuf;
    for (i = 0; i < count; i++) {
      if (min) {
        if (p2.uldip[i].fst < p1.uldip[i].fst) {
          p1.uldip[i].fst = p2.uldip[i].fst;
          p1.uldip[i].sec = p2.uldip[i].sec;
        }
      } else {
        if (p2.uldip[i].fst > p1.uldip[i].fst) {
          p1.uldip[i].fst = p2.uldip[i].fst;
          p1.uldip[i].sec = p2.uldip[i].sec;
        }
      }
    }
    break;
  default:
    printf ("This operation isn't defined for this datatype!\n");
    exit (1);


  }

  return 0;
}

// tag -15 is used for MPI_REDUCE
int
MPI_Reduce (void *sendbuf, void *recvbuf, int count,
            MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
{
  int nump, rank;
  int bm, nb, cb;
  int partner;
  MPI_Status status;
  void *buf1, *buf2;
  MPI_User_function *userop;


  MPI_Comm_size (comm, &nump);
  MPI_Comm_rank (comm, &rank);

  if (rank == -1) {
    printf ("Wrong communicator in calling MPI_Reduce\n");
```

```
  fflush (stdout);
  return 1;
}

nb = _USF_num_bits (nump - 1);
bm = _USF_make_bm (nb);

if (rank == 0)
  rank = root;
else if (rank == root)
  rank = 0;
for (cb = 1; cb <= bm; cb <<= 1) {
  partner = rank ^ cb;

  if (rank < partner) {
    if (partner == 0)
      partner = root;
    else if (partner == root)
      partner = 0;
    if (partner >= nump)
      continue;
    MPI_Recv (recvbuf, count, datatype, partner, -15, comm, &status);
    if ((rank == 0) && (cb == bm)) {
      buf1 = recvbuf;
      buf2 = sendbuf;
    } else {
      buf1 = sendbuf;
      buf2 = recvbuf;
    }
    switch (op) {
    case MPI_MAX:
      _USF_solve_max (buf1, buf2, count, datatype);
      break;
    case MPI_MIN:
      _USF_solve_min (buf1, buf2, count, datatype);
      break;
    case MPI_SUM:
      _USF_solve_sum (buf1, buf2, count, datatype);
      break;
    case MPI_PROD:
      _USF_solve_prod (buf1, buf2, count, datatype);
      break;
    case MPI_LAND:
      _USF_solve_land (buf1, buf2, count, datatype);
      break;
    case MPI_BAND:
      _USF_solve_band (buf1, buf2, count, datatype);
      break;
    case MPI_LOR:
      _USF_solve_lor (buf1, buf2, count, datatype);
      break;
    case MPI_BOR:
      _USF_solve_bor (buf1, buf2, count, datatype);
      break;
    case MPI_LXOR:
      _USF_solve_lxor (buf1, buf2, count, datatype);
      break;
    case MPI_BXOR:
      _USF_solve_bxor (buf1, buf2, count, datatype);
      break;
    case MPI_MINLOC:
      _USF_solve_2loc (buf1, buf2, count, datatype, 1);
      break;
```

```
      case MPI_MAXLOC:
        _USF_solve_2loc (buf1, buf2, count, datatype, 0);
        break;
      default:
        if (op >= 120) {
          //This is a user defined opreation
          userop = Get_User_function (op);
          if (userop == NULL) {
            printf ("This operation isn't defined!\n");
            exit (1);
          } else {
            userop (buf2, buf1, &count, &datatype);
          }
        } else {
          printf ("This operation isn't defined!\n");
          exit (1);
        }
      }
    } else {
      if (partner == 0)
        partner = root;
      else if (partner == root)
        partner = 0;
      MPI_Send (sendbuf, count, datatype, partner, -15, comm);
      break;
    }
  }

  return 0;
}
```

B.27 request.c

```
#include<stdio.h>
#include "mpi.h"

_USF_request_node *_USF_request_hd, *_USF_request_tl;


//Looking for local send _USF_SEND request
_USF_request_node *
_USF_request_lookup (int tag, int peer, int type, int comm)
{
  _USF_request_node *node = _USF_request_hd;

  while (node != NULL) {
#ifdef DEBUG
    printf (" ---> t %d pe %d ty %d c %d\n", tag, peer, type, comm);
    printf (" ---> t %d pe %d ty %d c %d\n", node->req->tag, node->req->peer,
            node->req->type, node->req->comm);
    fflush (stdout);
#endif
    if (((tag == node->req->tag) ||
         ((tag == MPI_ANY_TAG) && (tag >= -1))) &&
        (peer == node->req->peer) && (type == node->req->type)
        && (comm == node->req->comm) && (node->req->done == 0)
        && (node->req->active))
      return node;
    node = node->next;
  }
  return node;
```

```
}

//Looking for multiple local request
_USF_request_node *
_USF_request_lookup_m (int tag, int peer, int type, int comm, int multiple)
{
  _USF_request_node *node = _USF_request_hd;

  while (node != NULL) {

#ifdef DEBUG
    printf ("m---> t %d pe %d ty %d c %d m %d\n", tag, peer, type, comm,
            multiple);
    printf ("m---> t %d pe %d ty %d c %d m %d a %d d %d\n", node->req->tag,
            node->req->peer, node->req->type, node->req->comm,
            node->req->multiple, (node->req->active), node->req->done);
    fflush (stdout);
#endif

    if (type == _USF_SEND) {    // Modified by Qing Huang

      if (((tag == node->req->tag) || (tag == MPI_ANY_TAG)) &&
          (peer == node->req->peer) && (type == node->req->type)
          && (comm == node->req->comm) && (node->req->done == 0)
          && (multiple == node->req->multiple) && (node->req->active))
        return node;

    } else if (type == _USF_RECEIVE) {
      if (((tag == node->req->tag) || (node->req->tag == MPI_ANY_TAG))
          && ((peer == node->req->peer)
              || (node->req->peer == MPI_ANY_SOURCE))
          // Modified by Qing Hunag
          && (type == node->req->type)
          && (comm == node->req->comm) && (node->req->done == 0)
          && (multiple == node->req->multiple) && (node->req->active))
        return node;
    }
    node = node->next;
  }
  return node;
}


//Looking for local send _USF_RECEIVE or _USF_PROBE request
_USF_request_node *
_USF_request_lookup_p (int tag, int type, int comm, int posted)
{
  _USF_request_node *node = _USF_request_hd;

  while (node != NULL) {
#ifdef DEBUG
    printf ("p---> t %d ty %d c %d p %d \n", tag, type, comm, posted);
    printf ("p---> t %d ty %d c %d p %d\n", node->req->tag, node->req->type,
            node->req->comm, node->req->posted);
    fflush (stdout);
#endif
    if (((tag == node->req->tag) ||
         ((node->req->tag == MPI_ANY_TAG) && (node->req->tag >= -1))
         || ((tag == MPI_ANY_TAG) && (tag >= -1))) &&
        (type == node->req->type) && (comm == node->req->comm) &&
        (posted == node->req->posted) && (node->req->done == 0)
        && (node->req->multiple == 0) && (node->req->active))
      //may not need multiple here
```

```
      return node;
    node = node->next;
  }
  return node;
}

void
_USF_request_inshd (_USF_request_node * value)
{
  value->next = NULL;
  value->back = NULL;
  if (_USF_request_hd == NULL) {
    _USF_request_hd = value;
    _USF_request_tl = value;
  } else {
    _USF_request_hd->back = value;
    value->next = _USF_request_hd;
    _USF_request_hd = value;
  }
  return;
}

void
_USF_request_instl (_USF_request_node * value)
{
  value->next = NULL;
  value->back = NULL;
  if (_USF_request_hd == NULL) {
    _USF_request_hd = value;
    _USF_request_tl = value;
  } else {
    _USF_request_tl->next = value;
    value->back = _USF_request_tl;
    _USF_request_tl = value;
  }
  return;
}

void
_USF_request_remove (_USF_request_node * elem)
{
  if (_USF_request_hd == NULL || elem == NULL) {
    // Shouldn't happen in a correct program.
    printf ("Cannot be removed, request queue is empty.\n");
    fflush (stdout);
    return;
  }
  if (_USF_request_hd->next == NULL) {  /* One element */
    if (_USF_request_hd != elem) {
      printf
        ("Request is wrong, the node doesn't belong to the reques queue.\n");
      fflush (stdout);
      return;                   /* Shouldn't happen in a correct program. */
    }
    _USF_request_hd = NULL;
    _USF_request_tl = NULL;
  } else if (_USF_request_hd == elem) {
    elem->next->back = NULL;
    _USF_request_hd = elem->next;
  } else if (_USF_request_tl == elem) {
    elem->back->next = NULL;
    _USF_request_tl = elem->back;
  } else {
```

```
    elem->next->back = elem->back;
    elem->back->next = elem->next;
  }
  if (elem != NULL) {           /* Free the element */
    if (elem->req != NULL)
      pool_return (_USF_mpi_request_int_pool, (pool_item *) elem->req);
    pool_return (_USF_request_node_pool, (pool_item *) elem);
  }
  return;
}

void
_USF_request_free ()
{
  int i;
  _USF_request_node *curr = _USF_request_hd;
  _USF_request_node *elem;


  _USF_request_hd = NULL;
  _USF_request_tl = NULL;
  while (curr != NULL) {
    elem = curr;
    curr = curr->next;
    if (elem != NULL) {         /* Free the element */
      if (elem->req != NULL)
        pool_return (_USF_mpi_request_int_pool, (pool_item *) elem->req);
      pool_return (_USF_request_node_pool, (pool_item *) elem);
    }
  }
  return;
}
```

B.28 scatter.c

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<unistd.h>
#include<pthread.h>
#include"errno.h"
#include"mpi.h"

#define LP printf

// tag -13 is used for gather.
int
MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype,
             void *recvbuf, int recvcount, MPI_Datatype recvtype,
             int root, MPI_Comm comm)
{
  int nump, rank, i, j, s;
  int dene[5];
  MPI_Request *req;
  MPI_Status *status;

  MPI_Comm_size (comm, &nump);
  MPI_Comm_rank (comm, &rank);

  if (rank == -1) {
    printf ("Wrong communicator in calling MPI_Scatter\n");
```

```
    fflush (stdout);
    return 1;
  }

  s = _USF_mpi_sizeof (recvtype);

  req = (MPI_Request *) malloc ((nump - 1) * sizeof (MPI_Request));
  status = (MPI_Status *) malloc ((nump - 1) * sizeof (MPI_Status));

  if (rank == root) {
    for (i = 0, j = 0; i < nump; i++, j++) {
      if (i == rank) {
        j--;
        continue;
      }
      MPI_Isend ((sendbuf + (i * sendcount * s)), sendcount,
                 sendtype, i, -13, comm, &(req[j]));
    }

    memcpy (recvbuf, (sendbuf + (root * sendcount * s)), sendcount * s);
    MPI_Waitall ((nump - 1), req, status);
  } else {
    MPI_Recv (recvbuf, recvcount, recvtype, root, -13, comm, status);
  }
  return 0;
}

// tag -18 is used for gather.
int
MPI_Scatterv (void *sendbuf, int *sendcount, int *displs,
              MPI_Datatype sendtype, void *recvbuf, int recvcount,
              MPI_Datatype recvtype, int root, MPI_Comm comm)
{
  int nump, rank, i, j, s;
  int dene[5];
  MPI_Request *req;
  MPI_Status *status;

  MPI_Comm_size (comm, &nump);
  MPI_Comm_rank (comm, &rank);

  if (rank == -1) {
    printf ("Wrong communicator in calling MPI_Scatter\n");
    fflush (stdout);
    return 1;
  }

  s = _USF_mpi_sizeof (recvtype);

  req = (MPI_Request *) malloc ((nump - 1) * sizeof (MPI_Request));
  status = (MPI_Status *) malloc ((nump - 1) * sizeof (MPI_Status));

  if (rank == root) {
    for (i = 0, j = 0; i < nump; i++, j++) {
      if (i == rank) {
        j--;
        continue;
      }
      MPI_Isend (sendbuf + displs[i], sendcount[i],
                 sendtype, i, -18, comm, &(req[j]));
    }

    memcpy (recvbuf, sendbuf + displs[root], sendcount[root] * s);
```

```
    MPI_Waitall ((nump - 1), req, status);
  } else {
    MPI_Recv (recvbuf, recvcount, recvtype, root, -18, comm, status);
  }
  return 0;
}
```

B.29 sendrecv.c

```
#include<stdio.h>
#include<stdlib.h>
#include"mpi.h"

int
MPI_Sendrecv (void *sendbuf, int sendcount, MPI_Datatype sendtype,
              int dest, int sendtag, void *recvbuf, int recvcount,
              MPI_Datatype recvtype, int source, int recvtag,
              MPI_Comm comm, MPI_Status * status)
{
  MPI_Request reqs[2];
  MPI_Status st;

  MPI_Isend (sendbuf, sendcount, sendtype, dest, sendtag, comm, &reqs[0]);
  MPI_Irecv (recvbuf, recvcount, recvtype, source, recvtag, comm, &reqs[1]);
  MPI_Wait (&reqs[0], &st);
  MPI_Wait (&reqs[1], status);
  return 0;
}
```

B.30 sendrecv_replace.c

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include"mpi.h"

int
MPI_Sendrecv_replace (void *buf, int count, MPI_Datatype datatype,
                      int dest, int sendtag, int source, int recvtag,
                      MPI_Comm comm, MPI_Status * status)
{
  MPI_Request reqs[2];
  MPI_Status st;
  void *tmpbuf;
  int size;

  size = _USF_mpi_sizeof (datatype) * count;
  tmpbuf = malloc (size);
  memcpy (tmpbuf, buf, size);
  MPI_Isend (tmpbuf, count, datatype, dest, sendtag, comm, &reqs[0]);
  MPI_Irecv (buf, count, datatype, source, recvtag, comm, &reqs[1]);
  MPI_Wait (&reqs[0], &st);
  MPI_Wait (&reqs[1], status);
  free (tmpbuf);
  return 0;
}
```

B.31 start.c

```
#include<stdio.h>
#include<stdlib.h>
#include"mpi.h"

#ifdef _USF_MPI_GM_
extern pthread_mutex_t _USF_mut_gm_pending;
extern int request_pending;
extern int thread_recving;
#endif

int
MPI_Start (MPI_Request * request)
{
  int pipei = 0;                /* The pipe var. to communicate with comm_thread. */
  int rank, nump;

  pthread_mutex_lock (&_USF_mut_req);
  if ((*request)->active != 0) {
    printf ("Error request should have been equal to 0\n");
    pthread_mutex_unlock (&_USF_mut_req);
    exit (0);
  } else {
    (*request)->active = 2;
  }
  pthread_mutex_unlock (&_USF_mut_req);

#ifndef SINGLE_THREAD

#ifdef _USF_MPI_TCP_
  write (_USF_s_pipe[1], &pipei, sizeof (int));
#endif

#ifdef _USF_MPI_GM_
  pthread_mutex_lock (&_USF_mut_gm_pending);
  request_pending++;
  pthread_mutex_unlock (&_USF_mut_gm_pending);
#endif

#else

#ifdef _USF_MPI_TCP_
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &nump);
  _USF_serve_mpi_requests (rank, nump);
#endif

#ifdef _USF_MPI_GM_
  request_pending++;
  thread_recving = 1;
  gm_thread ();
#endif

#endif

  return 0;
}
```

B.32 startall.c

```
#include<stdio.h>
#include<stdlib.h>
#include"mpi.h"

#ifdef _USF_MPI_GM_
extern pthread_mutex_t _USF_mut_gm_pending;
extern int request_pending;
extern int thread_recving;
#endif

int
MPI_Startall (int count, MPI_Request * array_of_requests)
{
  int i;
  int pipei = 0;                /* The pipe var. to communicate with comm_thread. */
  int rank, nump;

  pthread_mutex_lock (&_USF_mut_req);
  for (i = 0; i < count; i++) {
    if ((array_of_requests[i])->active != 0) {
      printf ("Error request should have been equal to 0\n");
      pthread_mutex_unlock (&_USF_mut_req);
      exit (0);
    } else {
      (array_of_requests[i])->active = 2;
    }
  }
  pthread_mutex_unlock (&_USF_mut_req);

#ifndef SINGLE_THREAD

#ifdef _USF_MPI_TCP_
  write (_USF_s_pipe[1], &pipei, sizeof (int));
#endif

#ifdef _USF_MPI_GM_
  pthread_mutex_lock (&_USF_mut_gm_pending);
  request_pending++;
  pthread_mutex_unlock (&_USF_mut_gm_pending);
#endif

#else

#ifdef _USF_MPI_TCP_
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &nump);
  _USF_serve_mpi_requests (rank, nump);
#endif

#ifdef _USF_MPI_GM_
  request_pending++;
  thread_recving = 1;
  gm_thread ();
#endif

#endif


  return 0;
}
```

B.33 sysreq.c

```
#include<stdio.h>
#include"mpi.h"

_USF_sysreq_node *_USF_sysreq_hashtable[_USF_MAXPROCESSORS];

_USF_sysreq_node *
_USF_sysreq_lookup (int tag, int source, int type, int comm)
{
  int h = 0;
  int i = 0;
  int nump = 0;
  _USF_sysreq_node *node = NULL;

  MPI_Comm_size (comm, &nump);  // Modified by Qing Huang

  if (type == _USF_SEND) {      // Search for Send request

    if (source == MPI_ANY_SOURCE) {
      for (i = 0; i < nump; i++) {
        h = i % _USF_MAXPROCESSORS;
        for (node = _USF_sysreq_hashtable[h]; node != NULL; node = node->next) {
#ifdef DEBUG
          printf (" ---> s %d sr %d ty %d c %d\n", tag, i, type, comm);
          printf (" ---> s %d sr %d ty %d c %d\n", node->tag, node->source,
                  node->type, node->comm);
#endif
          if (((tag == node->tag) ||
               ((tag == MPI_ANY_TAG) && (tag >= -1)))
              && (i == node->source)
              && (type == node->type)
              && (comm == node->comm)) {
#ifdef DEBUG
            printf (" ---> Found Send Sys node and return\n");
            fflush (stdout);
#endif
            return node;
          }
        }
      }
#ifdef DEBUG
      printf (" ---> Not Found Send Sys node and return\n");
      fflush (stdout);
#endif
      return node;
    }
    h = source % _USF_MAXPROCESSORS;
    for (node = _USF_sysreq_hashtable[h]; node != NULL; node = node->next) {
      if (((tag == node->tag) || (tag == MPI_ANY_TAG))
  // Modified by Qing Huang
          && (source == node->source)
          && (type == node->type) && (comm == node->comm)) {
        return node;
      }
    }
  } else if (type == _USF_RECEIVE) {    // Search for Recieve request

    h = source % _USF_MAXPROCESSORS;
    for (node = _USF_sysreq_hashtable[h]; node != NULL; node = node->next) {
      if (((tag == node->tag) || (node->tag == MPI_ANY_TAG))
  // Modified by Qing Huang
          && (source == node->source)
```

```
          && (type == node->type) && (comm == node->comm)) {
        return node;
      }
    }

  }


  return node;
}


_USF_sysreq_node *
_USF_sysreq_remove (int tag, int source, int type, int comm)
{
  int h = 0;
  int i = 0;
  int nump = 0;
  _USF_sysreq_node *node = NULL, *prev = NULL;

  MPI_Comm_size (comm, &nump);

  if (type == _USF_SEND) {
#ifdef DEBUG

    printf (" --->%d Start to remove type _USF_SEND\n", nump);
    fflush (stdout);
#endif

    if (source == MPI_ANY_SOURCE) {
      for (i = 0; i < nump; i++) {
        h = i % _USF_MAXPROCESSORS;
        for (prev = node = _USF_sysreq_hashtable[h]; node != NULL;
             prev = node, node = node->next) {
#ifdef DEBUG
          printf (" ---> r %d sr %d ty %d c %d\n", tag, i, type, comm);
          printf (" ---> r %d sr %d ty %d c %d\n", node->tag, node->source,
                  node->type, node->comm);
#endif
          if (((tag == node->tag) || ((tag == MPI_ANY_TAG) && (tag >= -1)))
      // Modified by Qing Huang
              /*may need || (node->tag == -1)) MPI_ANY_TAG */
              && (i == node->source)    // Modified by Qing Huang
              && (type == node->type)
              && (comm == node->comm)) {
#ifdef DEBUG
            printf (" ---> Find Send Sys node and will remove it\n");
            fflush (stdout);
#endif
            if (node == prev) {
              _USF_sysreq_hashtable[h] = node->next;
            } else {
              prev->next = node->next;
            }
#ifdef DEBUG
            printf (" ---> Found Send Sys node and removed\n");
            fflush (stdout);
#endif
            return node;
          }
        }
      }
#ifdef DEBUG
```

```
      printf (" ---> Not Found Send Sys node and not removed\n");
      fflush (stdout);
#endif
      return node;
    }
    h = source % _USF_MAXPROCESSORS;
    for (prev = node = _USF_sysreq_hashtable[h];
         node != NULL; prev = node, node = node->next) {
      if (((tag == node->tag) || (tag == MPI_ANY_TAG))
  // Modified by Qing Huang
          /*may need || (node->tag == -1)) MPI_ANY_TAG */
          && ((source == node->source) || (source == MPI_ANY_SOURCE))
  // Modified by Qing Huang
          && (type == node->type) && (comm == node->comm)) {
        if (node == prev) {
          _USF_sysreq_hashtable[h] = node->next;
        } else {
          prev->next = node->next;
        }
#ifdef DEBUG
        printf (" ---> Found Send Sys node and removed\n");
        fflush (stdout);
#endif
        break;
      }
    }
  }

  else if (type == _USF_RECEIVE) {

#ifdef DEBUG
    printf (" ---> Start to remove type _USF_RECEIVE\n");
    fflush (stdout);
#endif

    h = source % _USF_MAXPROCESSORS;
    for (prev = node = _USF_sysreq_hashtable[h]; node != NULL;
         prev = node, node = node->next) {
      if (((tag == node->tag) || (node->tag == MPI_ANY_TAG) || tag == MPI_ANY_TAG)
  // Modified by Qing Huang
          /*may need || (node->tag == -1)) MPI_ANY_TAG */
          && (source == node->source)   // Modified by Qing Huang
          && (type == node->type) && (comm == node->comm)) {
        if (node == prev) {
          _USF_sysreq_hashtable[h] = node->next;
        } else {
          prev->next = node->next;
        }
#ifdef DEBUG
        printf (" ---> Found Recv Sys node and removed\n");
        fflush (stdout);
#endif
        break;
      }
    }

  }


  return node;
}

#ifdef _USF_MPI_TCP_
```

```
void
_USF_sysreq_insert (int tag, int source, int type, int comm, int size,
                    _USF_sysreq_node * value)
{
  int h;
  _USF_sysreq_node *node = NULL, *prev = NULL;

  h = source % _USF_MAXPROCESSORS;
  value->next = NULL;
  value->tag = tag;
  value->source = source;
  value->type = type;
  value->comm = comm;
  value->size = size;

  if (_USF_sysreq_hashtable[h] == NULL) {
    _USF_sysreq_hashtable[h] = value;
    return;
  }
  for (node = _USF_sysreq_hashtable[h];
       node != NULL; prev = node, node = node->next);
  prev->next = value;
  return;
}
#endif


#ifdef _USF_MPI_GM_
void
_USF_sysreq_insert (int tag, int source, int type, int comm, int size,
                    void *in_buf, _USF_sysreq_node * value)
{
  int h;
  _USF_sysreq_node *node = NULL, *prev = NULL;

  h = source % _USF_MAXPROCESSORS;
  value->next = NULL;
  value->tag = tag;
  value->source = source;
  value->type = type;
  value->comm = comm;
  value->size = size;
  value->in_buf = in_buf;
  if (_USF_sysreq_hashtable[h] == NULL) {
    _USF_sysreq_hashtable[h] = value;
    return;
  }
  for (node = _USF_sysreq_hashtable[h];
       node != NULL; prev = node, node = node->next);
  prev->next = value;
  return;
}
#endif

void
_USF_sysreq_free ()
{
  int i;
  _USF_sysreq_node *curr;

  for (i = 0; i < _USF_MAXPROCESSORS; i++) {
    while (_USF_sysreq_hashtable[i] != NULL) {
      curr = _USF_sysreq_hashtable[i];
```

```
      _USF_sysreq_hashtable[i] = _USF_sysreq_hashtable[i]->next;
      if (curr != NULL)
        free (curr);
    }
  }
  return;
}
```

B.34 test.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include"mpi.h"

int
MPI_Test (MPI_Request * request, int *flag, MPI_Status * status)
{
  pthread_mutex_lock (&_USF_mut_req);
  status->count = 0;
  *flag = 0;
  if (*request == MPI_REQUEST_NULL) {
    pthread_mutex_unlock (&_USF_mut_req);
    return 0;
  }
  if (!(*request)->active) {
    pthread_mutex_unlock (&_USF_mut_req);
    return 0;
  }
  if ((*request)->done) {
    *flag = 1;
    status->count = (*request)->size;
    status->MPI_SOURCE = (*request)->peer;
    status->MPI_TAG = (*request)->tag;
    if ((*request)->active == 2) {
      (*request)->active == 0;
    } else {
      _USF_request_remove ((*request)->node);
      *request = MPI_REQUEST_NULL;
    }
    pthread_mutex_unlock (&_USF_mut_req);
    return 0;
  }
  return 0;
}
```

B.35 testall.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include"mpi.h"

int
MPI_Testall (int count, MPI_Request * array_of_requests, int *flag,
             MPI_Status * array_of_statuses)
{
  int i;
```

```
  int tmp;

  *flag = 0;
  for (i = 0; i < count; i++) {
    MPI_Test (&array_of_requests[i], &tmp, &array_of_statuses[i]);
    if (i == 0) {
      *flag = tmp;
    } else {
      if (!(*flag && tmp))
        *flag = 0;
    }
  }
  return 0;
}
```

B.36 testany.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<time.h>
#include"mpi.h"

int
MPI_Testany (int count, MPI_Request * array_of_requests, int *index,
             int *flag, MPI_Status * status)
{
  int i;

  *flag = 0;
  for (i = 0; i < count; i++) {
    pthread_mutex_lock (&_USF_mut_req);
    status->count = 0;
    if (array_of_requests[i] == MPI_REQUEST_NULL) {
      pthread_mutex_unlock (&_USF_mut_req);
      continue;
    }
    if (!(array_of_requests[i]->active)) {
      pthread_mutex_unlock (&_USF_mut_req);
      continue;
    }
    if (array_of_requests[i]->done) {
      *flag = 1;
      *index = i;
      status->count = array_of_requests[i]->size;
      status->MPI_SOURCE = array_of_requests[i]->peer;
      status->MPI_TAG = array_of_requests[i]->tag;
      _USF_request_remove (array_of_requests[i]->node);
      array_of_requests[i] = MPI_REQUEST_NULL;
      pthread_mutex_unlock (&_USF_mut_req);
      return 0;
    }
    pthread_mutex_unlock (&_USF_mut_req);
  }
  return 0;
}
```

B.37 user_func.c

```
// MPI user defined reduce function implementation
// Created by Chaney 2002
// Uniersity of San Francisco

#include <string.h>
#include <stdlib.h>
#include "mpi.h"


typedef struct StructUserFunc
{
  int idx;
  int commute;
  MPI_User_function *ptr2userfunc;
  struct StructUserFunc *next;
}
USF_User_Func_Node;

USF_User_Func_Node *head_userfunc = NULL;
//Set start number of user define op
int User_Op_Index = 120;

//Create user define operation
int
MPI_Op_create (MPI_User_function * userfunc, int commute, MPI_Op * op)
{
  USF_User_Func_Node *funcnode;

  funcnode = (USF_User_Func_Node *) malloc (sizeof (USF_User_Func_Node));
  funcnode->idx = User_Op_Index;
  *op = User_Op_Index;
  User_Op_Index++;
  funcnode->commute = commute;
  funcnode->ptr2userfunc = userfunc;
  funcnode->next = head_userfunc;
  head_userfunc = funcnode;
}

//Create user define operation
int
MPI_op_free (MPI_Op * op)
{
  USF_User_Func_Node *ptr, *pre;

  ptr = head_userfunc;
  pre = ptr;
  while (ptr != NULL) {
    if (ptr->idx == *op) {
      *op = MPI_OP_NULL;
      if (pre != ptr) {
        pre->next = ptr->next;
      } else {
        head_userfunc = ptr->next;
      }
      free (ptr);
      return MPI_SUCCESS;
    }
    pre = ptr;
    ptr = ptr->next;
  }
  return -1;
```

```
}

//Get user define operation (function pointer)
MPI_User_function *
Get_User_function (MPI_Op op)
{
  USF_User_Func_Node *ptr;

  ptr = head_userfunc;
  while (ptr != NULL) {
    if (ptr->idx == op) {
      return ptr->ptr2userfunc;
    }
    ptr = ptr->next;
  }
  return NULL;
}
```

B.38 util.c

```
//  Error message function
//  Get the size of MPI datatype
//
//  make connection list and set initial values
//


#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<errno.h>
#include<string.h>
#include"debug.h"
#include"mpi.h"
#include "gm.h"

_USF_conn_struct *_USF_conn_list;

#define DEBUG

int
Error (int wh)
{
  char *str;
  int rank;

  str = (char *) malloc (200 * sizeof (char));

  MPI_Comm_rank (MPI_COMM_WORLD, &rank);


  switch (wh) {
  case ER_INCORRECT:
    strcat (str, "The correct usage:\n");
    strcat (str, "mrun <execname> or mrun -np <#proc> <execname>\n");
    break;
  case ER_FILENAME:
    strcat (str, "Filename missing, or not executable.\n");
    break;
  case ER_NUMP:
    strcat (str, "The number of procs should be greater than zero.\n");
```

```
    break;
  case ER_NEMACS:
    strcat (str, "There aren't enough number of processors.\n");
    break;
  case ER_HOSTNM:
    strcat (str, "The host name unavailable.\n");
    break;
  case ER_MACFILE:
    strcat (str, "The machines file is missing.\n");
    break;
  case ERF_SOCKETINIT:
    strcat (str, "Unable to initialize socket.\n");
    break;
  case ERF_BINDINIT:
    strcat (str, "Unable to bind socket.\n");
    break;
  case ERF_ACCEPT:
    strcat (str, "Unable to accept new connection.\n");
    break;
  case ERF_RECV:
    strcat (str, "Unable to receive.\n");
    break;
  case ERF_SEND:
    strcat (str, "Unable to send.\n");
    break;
  case ERF_GETHE:
    strcat (str, "Unable to get address by hostname.\n");
    break;
  case ERF_CONNECT:
    strcat (str, "Unable to connect.\n");
    break;
  case ERF_SETSOCKOPT:
    strcat (str, "Setsockopt error.\n");
    break;
  case ERF_LISTEN:
    strcat (str, "Unable to listen.\n");
    break;
  case ERF_SELECT:
    strcat (str, "Select error.\n");
    break;
  case ERF_PTHREAD:
    strcat (str, "Pthread creation error.\n");
    break;
  default:
    strcat (str, "Unexpected error.\n");
    break;
  }
  printf ("Error num--> %d : %s", wh, str);
  system ("hostname");
  fflush (stdout);
  free (str);
  exit (0);                     /* may change to only exit if fatal error */
  return wh;
}

int
_USF_mpi_sizeof (MPI_Datatype ty)
{

  switch (ty) {
  case MPI_CHAR:
    return sizeof (char);
    break;
```

```
case MPI_SHORT:
  return sizeof (short);
  break;
case MPI_INT:
  return sizeof (int);
  break;
case MPI_LONG:
  return sizeof (long);
  break;
case MPI_UNSIGNED_CHAR:
  return sizeof (unsigned char);
  break;
case MPI_UNSIGNED_SHORT:
  return sizeof (unsigned short);
  break;
case MPI_UNSIGNED:
  return sizeof (unsigned);
  break;
case MPI_UNSIGNED_LONG:
  return sizeof (unsigned long);
  break;
case MPI_FLOAT:
  return sizeof (float);
  break;
case MPI_DOUBLE:
  return sizeof (double);
  break;
case MPI_LONG_DOUBLE:
  return sizeof (long double);
  break;
case MPI_BYTE:
  return sizeof (char);
  break;
case MPI_LONG_LONG:
  return sizeof (long long);
case MPI_PACKED:
  return sizeof (char);
  break;
case MPI_LB:
  return 1;
  break;
case MPI_UB:
  return 1;
  break;
case MPI_FLOAT_INT:
  return sizeof (float) + sizeof (int);
  break;
case MPI_DOUBLE_INT:
  return sizeof (double) + sizeof (int);
  break;
case MPI_LONG_INT:
  return sizeof (long) + sizeof (int);
  break;
case MPI_SHORT_INT:
  return sizeof (short) + sizeof (int);
  break;
case MPI_2INT:
  return sizeof (int) + sizeof (int);
  break;
case MPI_LONG_DOUBLE_INT:
  return sizeof (long double) + sizeof (int);
  break;
```

```
  default:
    return -1;
  }
}

int
_USF_make_conn_list (char *my_name, int nump)
{
  FILE *macs;                   // The machines file that mpi will use.
  char *mac_nm;                 // The name of a machine.
  char *cur_dir;                // The current directory.
  int mac_size;                 // The number of processors read.
  char *colonplace;             // The pointer to : in mac_nm str.
  int psize;                    // The number of procs on current ip.
  int i;                        // The loop variable.
  gm_status_t my_status;

  struct gm_port *my_port;

  cur_dir = (char *) malloc (256 * sizeof (char));
  mac_size = 0;
#ifdef _USF_MPI_TCP_
  int default_port = _USF_SERVPORT;
#endif

#ifdef _USF_MPI_GM_
  int default_port = 4;
#endif

  strcpy (cur_dir, getenv ("HOME"));
  strcat (cur_dir, "/machines.kcg");
  macs = fopen (cur_dir, "r");
  if (macs == NULL) {
    printf ("Tryed alternate location, still couldn't find machines file\n");
    return Error (ER_MACFILE);
  }
  _USF_conn_list = (_USF_conn_struct *)
    malloc (nump * sizeof (_USF_conn_struct));
  mac_nm = (char *) malloc (80 * sizeof (char));
  for (; ((fscanf (macs, "%s", mac_nm)) != EOF) && (mac_size < nump);) {
    if ((colonplace = strchr (mac_nm, (int) ':')) == NULL) {
      int i;
      _USF_conn_list[mac_size].hostname = mac_nm;
      for (i = mac_size; i > 0; i--)
        if (!strcmp
            (_USF_conn_list[i - 1].hostname,
             _USF_conn_list[mac_size].hostname))
          break;
      if (i != 0) {
#ifdef _USF_MPI_TCP_
        _USF_conn_list[mac_size].port = _USF_conn_list[i - 1].port + 1;
#endif
#ifdef _USF_MPI_GM_
        _USF_conn_list[mac_size].port_id = _USF_conn_list[i - 1].port_id + 1;
#endif
      } else {
#ifdef _USF_MPI_TCP_
        _USF_conn_list[mac_size].port = default_port;
#endif
#ifdef _USF_MPI_GM_
        _USF_conn_list[mac_size].port_id = default_port;
#endif
      }
```

```
      DBG (("%s, %d \n", _USF_conn_list[mac_size].hostname,
            _USF_conn_list[mac_size].port));
      mac_size++;
    } else {
      psize = atoi (colonplace + 1);
      for (i = 0; (i < psize) && (mac_size < nump); i++) {
        *colonplace = '\0';
        _USF_conn_list[mac_size].hostname = mac_nm;
        _USF_conn_list[mac_size].port = _USF_SERVPORT + i;
        DBG (("%s, %d\n", _USF_conn_list[mac_size].hostname,
              _USF_conn_list[mac_size].port));
        mac_size++;
      }
    }
    mac_nm = (char *) malloc (80 * sizeof (char));
  }
  free (mac_nm);
  fclose (macs);
  DBG (("--> util.c out of make\n"));


  for (i = 0; i < nump; i++) {
    _USF_conn_list[i].next_msg = _USF_HEADER_TYPE;
  }
  return 0;
}
```

B.39 wait.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include"mpi.h"
#include "gm.h"
#include "debug.h"

extern int debug_counter;
extern int thread_recving;

int
MPI_Wait (MPI_Request * request, MPI_Status * status)
{

  int nump;                     /* # of procs in COMM_WORLD */
  int rank;                     /* the rank in COMM_WORLD */

  MPI_Comm_size (MPI_COMM_WORLD, &nump);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);

  pthread_mutex_lock (&_USF_mut_req);
  status->count = 0;
  if (*request == MPI_REQUEST_NULL) {
    pthread_mutex_unlock (&_USF_mut_req);
    return 1;
  }
  if (!(*request)->active) {
    pthread_mutex_unlock (&_USF_mut_req);
    return 1;
  }

  pthread_mutex_unlock (&_USF_mut_req);
```

```
// multiple threaded mode
#ifndef SINGLE_THREAD

  pthread_mutex_lock (&_USF_mut_block);
#ifdef DEBUG
  printf ("rank %d !!Wait!! cond_bl start\n", _USF_comm_list[0].rank);
  fflush (stdout);
#endif
  if ((*request)->end_of_transmission == 0) {
#ifdef DEBUG
    printf ("rank %d waiting ############## %p debug_counter %d\n", rank,
            (*request), debug_counter);
    fflush (stdout);
    printf ("rank %d dest %d type %d tag %d comm %d size %d req %p \n", rank,
            (*request)->peer, (*request)->type, (*request)->tag,
            (*request)->comm, (*request)->size, (*request));
    fflush (stdout);
#endif
    (*request)->blocking = 1;
    pthread_cond_wait (&_USF_cond_block, &_USF_mut_block);

  }
#ifdef DEBUG
  printf ("rank %d !!Wait!! cond_bl end\n", _USF_comm_list[0].rank);
  fflush (stdout);
#endif
  pthread_mutex_unlock (&_USF_mut_block);

// single threaded mode
#else

#ifdef _USF_MPI_TCP_
  int i;                        /* Loop variable */

  fd_set set;                   /* the set that is used for select */
  int max;                      /* the maximum file descriptor */
  int finished = 0;             /* The loop conditional */
  int nu;




  while (!((*request)->end_of_transmission)) {

    max = _USF_TCP_prepare_set (&set);
    if ((nu = pselect (max + 1, &set, NULL, NULL, NULL, NULL)) == -1)
      Error (ERF_SELECT);
    DBG (("rank %d, woken up from select! sel %d \n", rank, nu));
    for (i = 0; i < nump; i++) {
      if (i == rank)
        continue;
      if (FD_ISSET (_USF_conn_list[i].sd, &set)) {
        DBG (("rank %d handling message %d start\n", rank, i));
        _USF_handle_msg (i);
        DBG (("rank %d handling message %d end\n", rank, i));
      }
    }
  }
#endif

#ifdef _USF_MPI_GM_
```

```
  while (!((*request)->end_of_transmission)) {
    thread_recving = 1;
    gm_thread ();
  }
#endif

#endif




  /* The message has arrived */
  pthread_mutex_lock (&_USF_mut_req);
  status->count = (*request)->size;
  status->MPI_SOURCE = (*request)->peer;
  status->MPI_TAG = (*request)->tag;
  if ((*request)->active == 2) {
    (*request)->active = 0;     // modified by Qing
  } else {
    _USF_request_remove ((*request)->node);
    *request = MPI_REQUEST_NULL;
  }

  pthread_mutex_unlock (&_USF_mut_req);
  return 0;
}
```

B.40 waitall.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include"mpi.h"

int
MPI_Waitall (int count, MPI_Request * array_of_requests,
             MPI_Status * array_of_statuses)
{
  int i;

  for (i = 0; i < count; i++) {
    MPI_Wait (&array_of_requests[i], &array_of_statuses[i]);
  }
  return 0;
}
```

B.41 waitany.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<time.h>
#include"mpi.h"

int
MPI_Waitany (int count, MPI_Request * array_of_requests, int *index,
             MPI_Status * status)
{
```

```
  const struct timespec t1 = { 0, 500000 };
  struct timespec t2;
  int i;

  while (1) {
    for (i = 0; i < count; i++) {
      pthread_mutex_lock (&_USF_mut_req);
      status->count = 0;
      if (array_of_requests[i] == MPI_REQUEST_NULL) {
        pthread_mutex_unlock (&_USF_mut_req);
        continue;
      }
      if (!(array_of_requests[i]->active)) {
        pthread_mutex_unlock (&_USF_mut_req);
        continue;
      }
      if (array_of_requests[i]->done) {
        *index = i;
        status->count = array_of_requests[i]->size;
        status->MPI_SOURCE = array_of_requests[i]->peer;
        status->MPI_TAG = array_of_requests[i]->tag;
        _USF_request_remove (array_of_requests[i]->node);
        array_of_requests[i] = MPI_REQUEST_NULL;
        pthread_mutex_unlock (&_USF_mut_req);
        return 0;
      }
      pthread_mutex_unlock (&_USF_mut_req);
    }
    nanosleep (&t1, &t2);
  }
  return 0;
}
```