DISTRIBUTED PARALLEL COMPUTATION USING STANDARD ML

By

VAISHALI CHATTOPADHYAY

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

DECEMBER 2007

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of VAISHALI CHATTOPADHYAY find it satisfactory and recommend that it be accepted.

Chair

Acknowledgments

I would like to thank my advisor Dr. Carl Hauser, for his invaluable support, encouragement and guidance. But for him, this thesis would not have been possible.

I wish to thank the School of EECS for supporting my graduate study at WSU. Thanks is due to Ms. Ruby Young for answering all my questions with immense patience and guiding me through the maze of administrative protocol.

I would like to thank Dr. David Bakken and Dr. Murali Medidi to be on my committee.

I would like to thank Dr. Sumanth JV for his encouragement and support. I would also like to acknowledge Research Computing Facility (RCF) in University of Nebraska, Lincoln for letting me run my experiments.

My friends (who're far too many to name here) have been a great source of encouragement and help both in and out of school. Thank you all for your support.

DISTRIBUTED PARALLEL COMPUTATION USING STANDARD ML

Abstract

by Vaishali Chattopadhyay, M.S.

Washington State University
December 2007

Chair: Carl Hauser

This work describes the design and implementation of SMPI, the first native implementation of a library of functions that support parallel programming in SML. The intent of the proposed work is to provide the basic routines of MPI in SML to facilitate programmers to use SML for parallel programming. We find that the functional constructs available in SML aid in writing well structured, concise and robust code. We also implemented the same algorithms in Python and C in order to compare the performance of SML against it. This was necessary since the existing Python implementations are wrappers around MPICH. We chose to create a lightweight C implementation in order to perform a fair comparison of SML with C since MPICH, despite being implemented in C, incurs a significant overhead due to its high portability.

Contents

List of Figures

In the fond memory of my grandmother

Late Sabita Devi

Chapter 1

Introduction

Large computational applications are complex in structure and have significant execution time. However they can often be broken down into independent tasks that can be done in parallel. Efforts are being expended in defining methodologies and designing techniques which allow large complex applications to be constructed more reliably and run more efficiently. Parallel programming significantly reduces the elapsed computation time for programs while functional programming introduces structure into them making them easy to write, debug and re-use. Combining these two ideas, this thesis presents a native implementation of a message passing interface for parallel programming in a functional language and compares it to other native implementations in an imperative and an interpreted language.

For parallel programming, an efficient communication mechanism between the processes is a must. Message Passing is one of the paradigms used to enable communication between processors. The Message Passing Interface (MPI [41] [35]) is a library of routines which provides this mechanism to perform point-to-point and collective communication between processors. Point-to-point communication implies the communication between any two processors participating in the computation. Collective operations on the other hand imply the communication between all the processors participating in the computation. Simple send and receive would form a part of the point-to-point communication routines, while broadcast, reduce and synchronization routines like barrier would form the collective operations.

This thesis describes a library of routines containing the basic communication primitives of MPI using a functional language (Standard ML, [31]). This allows programmers to use parallel programming constructs in a functional language. In this thesis we first discuss the design and implementation of the message passing interface using the advanced programming language, SML, and then compare the performance of SMPI implementation with other native implementations in an imperative and interpreted language. Since most implementations available are only wrappers to MPICH [20], traditional implementation of MPI using a conventional language C, we had to implement a native implementation of MPI in Python, an interpreted language and to avoid MPICH's overhead performance cost a lightweight implementation in C also had to be implemented.

1.1 Motivation

In recent decades much attention has turned to parallel programming due to its ability to speed up computations. MPI allows processes to communicate in a parallel programming environment. Thus parallel programmers are now turning their focus to the development of MPI for efficiently passing messages between the processes. MPI has been implemented by several researchers successfully in conventional languages like C, C++, FORTRAN and Java. These implementations support the development of parallel programs written in these languages. Wrappers to these implementations have been written for other languages like Caml, Java, OCaml, Python etc. which will be discussed in Chapter 2.

Significant advances in hardware have assisted in speeding up computations, thus drawing the attention of programmers towards writing well-structured and easy to understand programs rather than emphasizing on the performance alone. Edoardo Biagioni [12] suggests that structured implementations of of the Transmission Control Protocol using an extension of the Standard ML (SML) language can be made as efficient as comparable implementations in other languages. Ensemble [48] a library of protocols used to build distributed application has been written entirely in a functional language Objective Caml [1], a dialect of ML. To quote Philip Wadler of Bell labs, ”Ensemble beats the performance of its predecessor, Horus, by a wide margin, even though Horus is written in C” [48]. He states that the performance improvement was achieved only due to improved design rather than through long hours of hand-coding the entire system in C. The common conception until recent times that functional languages have poorer performance than conventional languages like C was disproved by Philip Wadler in [49].

Previous efforts to develop MPI using functional style has been in providing wrapper functionality to some traditional MPI implementations in conventional languages like C, e.g. ScaMPI [8] which was a Caml [1] interface to MPI and OCamlMPI [46] OCaml interface to MPI. The wrappers developed have limited functionality as a result of the limitations of the language used for the native implementation over which the wrapper is built. Sava Mintchev [33] suggests that functional style of programming can speed up MPI collective operations and he provides functional specifications for the improved operations. However he states that due to the absence of implementation of MPI in functional language these specifications were translated into a imperative programming language. Thus we see that lack of native implementation of MPI in functional language has hindered the development of parallel programs in functional languages.

In this thesis we address this problem by providing an implementation of MPI using a functional language and compare its performance with our native C and Python implementations.

1.2 Why SML?

We chose Standard ML (SML [31]) as the basis for the implementation of MPI using a functional language because:

- SML module system makes the different parts of the program virtually independent and easily modifiable
- Higher order functions of SML allow functions to be passed as arguments which makes programming flexible
- SML has compile time type checking which facilitates writing error free code
- SML allows the use of some of the features of imperative style of programming
- SML has a sophisticated exception handling mechanism facilitating debugging

1.3 Main Goals

A native implementation of MPI in a functional language would encourage parallel programmers to develop programs in that language. Functional languages with features such as higher order functions, strong typing, modularity, polymorphism and clear syntax allow it to have certain advantages over imperative languages. Thus this thesis implements the message passing interface in a functional language and studies its performance against other native implementation and also studies how these features affect the MPI implementation and aid in developing well structured programs. SMPI is the first native implementation of MPI in SML and provides a library containing basic communication primitives to ease parallel programming in SML. The aim of SMPI library is to aid programmers in writing well structured parallel programs and assess the suitability of SML language mechanisms for use in parallel programming.

1.4 Organization of the thesis

The first chapter has given a general introduction and has provided the motivation and goals of this thesis. The second chapter describes the related and background work related to this thesis. It introduces parallel programming concepts and gives an overview of the message passing interface. It also introduces functional programming and describes its advantages for parallel programming. A few existing MPI implementations are discussed. Chapter 3 describes SMPI in detail from the application programmerś perspective. Chapter 4 describes the message passing algorithms used in the implementation of SMPI. Chapter 5 describes how functional style affects the structure of the program. Chapter 6 discusses the experimental setup and results of comparing SML implementation with MPICH and our native implementations in Python and C. Finally the seventh chapter forms the conclusion of the thesis providing a summary and a brief discussion about the future work.

Chapter 2

Background and related work

This chapter provides an overview of parallel programming, message passing interface and a brief discussion of functional language programming. It gives an overview of the history of the MPI standard and describes the bindings available in it. This chapter also discusses the previous implementation efforts of message passing interface in languages such as C, Caml, Python etc.

2.1 Parallel Programming

Parallel programming implies that a set of processors work cooperatively to solve a large computational problem by breaking it up into smaller tasks and executing them simultaneously. It enables computation of larger quantities of data within shorter execution time as compared to traditional sequential style of programming where each task is performed in an ordered manner. A large number of uniprocessors together provide a great potential for parallelism thereby increasing computation power. Babaoglu et al. in [10] suggest that there are some technical issues, which include heterogeneity, high-latency communication, fault tolerance and dynamic load balancing, that need to be addressed to efficiently exploit the parallelism inherent in a distributed system. An application must manage all these concerns in addition to computing a result. This necessitates efficient cooperation between processors for parallel programming and adds to the complexity of the programming task.

Fig. 2.1 shows us a pictorial representation of the processes executing in different processors communicating via a communication network.

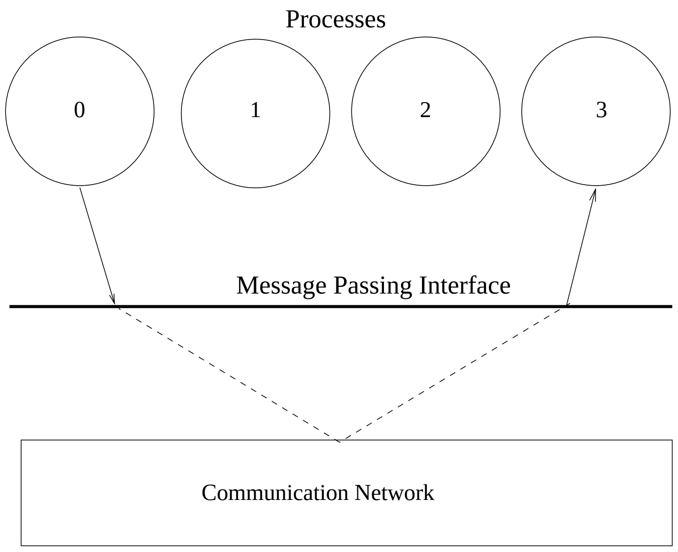

Figure 2.1: Parallel execution

Parallel programming can be implemented in a shared memory system or a distributed memory system. In a shared memory system as depicted in Fig. 2.2 all the processors have direct access to a common memory store through which they communicate. On the other hand in a distributed memory system the nodes are interconnected by a network where each node is a processor with its own local memory. Fig. 2.3 illustrates a distributed memory system. Even though shared memory systems outperform and are easier to program than distributed memory systems, the flexibility, scalability and low cost provided by the latter makes them more prevalent, [14]. Communication between the processes can be achieved through:

Distributed Shared Memory (DSM) model

> DSM allows sharing of data between processors that do not share physical memory. This is achieved by having a common memory segment which can be accessed by all the processors and which is updated regularly. Synchronization between processes is achieved via locks and semaphores.

Message Passing Model

> Message Passing model achieves communication between the nodes by exchanging messages across the network.

Non-Uniform Memory Access (NUMA) model

> In the NUMA model the memory of other processors can also be accessed along with its local memory. The time required to access these different memory modules may differ.

Among these three communication mechanism the message passing model is most widely used because of its scalability, security and cheap resource requirements, [14]. To improve performance in a distributed environment high performance switches and fast access mechanisms can be used. More details on the various models and parallel programming can be found in [14, 17, 42].

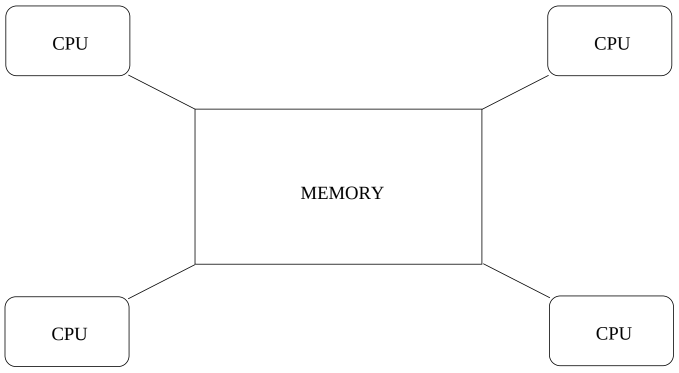

Figure 2.2: Shared Memory System

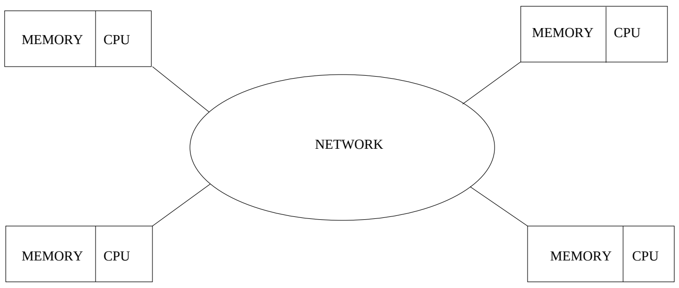

Figure 2.3: Distributed Memory System

2.1.1 Message Passing Interface

As programmers are turning toward parallel computing the need to ease parallel programming is becoming more prominent. Performance of a parallel program depends on how the computation is broken into smaller tasks and how efficiently they are made to communicate, [18]. Message passing enables this communication between processors in a distributed memory environment by transmitting data over an interconnected network. A message passing library is a collection of communication primitives that parallel processes use to communicate and synchronize with other parallel processes. Primitives include communication functions such as send, receive, reduce and broadcast, and synchronization primitives such as barrier, [19, 42].

A standard library of function calls that can be used to implement a message passing program is the Message Passing Interface (MPI), [21]. It provides an abstraction of how the underlying hardware is organized. Programmers can thus write parallel programs containing MPI subroutines and function calls that will work on any machine on which the MPI library is installed. These message passing libraries relieve the developers from the cumbersome task of network programming, and allow them to concentrate on program development.

2.1.2 Development of the MPI standard

A workshop on "Standards for Message Passing in Distributed Memory Environment" was conducted by the Center for Research in Parallel Computation in 1992, in which a MPI Forum consisting of eighty members from forty organizations discussed and defined an open and portable message passing standard. No efforts had been put in defining a standard until then, [19]. This had led to various vendors implementing their own message passing libraries and distributing them leading to non-portable programs. The initial specification for the MPI-1 standard was released in August 1994, [4]. This standard was developed by incorporating the most useful features of then existing implementations of MPI, [25]. The features that were included in the standard were point-to-point communication, collective operations, process groups, communication contexts, process topologies, bindings for Fortran 77 and C, environmental management and inquiry and profiling interface. These features enabled efficient computing in multiprocessor environments by defining a structure in which the various processors could communicate and in a collaborative manner perform computations. The features that were not included in the standard were explicit shared-memory operations, complex operations requiring more operating system support, program construction tools and debugging facilities, and explicit support for threads and task management. For more details on these features refer to [4].

The standard facilitated the development of parallel programs with efficient communication. The goals of the MPI design were portability and efficiency. The MPI standard provides bindings only for conventional languages like C and Fortran. The functionality provided includes point-to-point communication along with collective communication (broadcast, reduce, barrier). The standard has been changing ever since and several later versions have been released, [6]. Presently MPI-2 is also available. It includes library functions for dynamic process management, input/output routines, one-sided operations and C++ bindings. Dynamic process management provides a mechanism for newly created processes to communicate with existing MPI applications and for two existing MPI applications to communicate. The input output routines allow efficient partitioning and collecting of data between the various processors. The one sided operations were added to avoid corresponding routines to be present in every copy of the MPI program running on the various processors. Finally C++ bindings were introduced in the MPI-2 standard to facilitate creation of object oriented parallel applications. Detailed description of the features in MPI-2 is provided in [6].

2.1.3 Language bindings for MPI

A language binding consists of one or more constructs in a programming language that provides access to a defined service. MPI Language bindings for C, C++ and Fortran allow MPI services in these languages.

The form of a language binding varies with the programming language. In procedural, imperative languages like C and Fortran the binding is commonly defined by a library of procedures. However, in a binding for an object-oriented language, such as C++, classes, types and templates are used. A good language binding should preserve the semantic and conceptual model of the service and should not introduce overhead, [15]. It should allow the application developer to use the bindings easily and efficiently. If the execution cost of a language binding exceeds the benefits of it as structuring mechanism then it may be rejected.

2.1.4 MPI Implementations

Several implementations of the MPI standard are available. One of the most widely used implementations is MPICH (developed at Argonne National Laboratory and Mississippi State University), [16,20,22]. The other implementations are LAM/MPI developed at the Ohio Supercomputer Center, [3], CHIMP implementation from Edinburgh Parallel Computing Center, [9], OOMPI implementation from Open Systems Laboratory at Indiana University, [28] and Unify from Mississippi State University, [13]. A brief overview of some of them are given below:

C bindings for Point-to-Point operations

int **MPI_Send**(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
int **MPI_Recv**(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

C bindings for Collective operations

int **MPI_Barrier**(MPI_Comm comm)
int **MPI_Bcast**(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
int **MPI_Reduce**(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

C bindings for Groups, Contexts, and Communicators

int **MPI_Group_size**(MPI_Group group, int *size)
int **MPI_Group_rank**(MPI_Group group, int *rank)

Figure 2.4: Some of the bindings in MPI for C

2.1.4.1 MPICH

MPICH is an open-source C implementation of MPI-1 developed at the Argonne National Laboratory and Mississippi State University by Gropp and Lusk in 1992, [16, 20, 22]. This used the C bindings provided in the MPI standard and was developed with the aim of providing a reference implementation of the MPI standard. Being the first complete implementation of the MPI standard and being freely available it is most widely used.

In an MPICH program, a global communicator MPI_COMM_WORLD consisting of the details about the environment in which MPICH application is being run is created. It identifies all the processors that are participating in the computation along with the unique id that is assigned to it during initialization. This communicator is passed as an argument to all the MPI routines. MPICH allows definition of groups of processors. Communication within a group is classified as intra communication while communication across groups is defined as inter communication. MPICH supports both inter and intra group communication. The various types of communications are point to point communication and collective communication. Point-to-point communication include the basic send and receive primitives. The prototype for these function calls is similar to the ones provided in Fig. 2.4. The function calls have a long parameter list specifying the communicator, buffers to hold the messages to be sent or received, datatype of the message, source and destination of the message and the tags and status of the communication. The buffers for the message are type casted to *void ** and then sent or received. This makes this routines *type unsafe*. The collective operations also have similar prototype definition as illustrated in Fig. 2.4.

2.1.4.2 LAM/MPI

LAM/MPI developed at the Ohio Supercomputer Center [3],provides C, C++ and Fortran 77 bindings for all MPI-1 functions, types and constants. It also supports some of the features specified in MPI-2 like dynamic process creation, MPI input output and one sided communications. The core feature of LAM/MPI is the System Service Interface (SSI) which provides a component framework for the LAM run-time environment and the MPI communication layer. SSI allows the libraries required by the MPI programs to be added during runtime.

LAM/MPI implements point-to-point communication [44], collective operations [43] and checkpoint/restart support [39, 40] for MPI. The point-to-point communication is also known as the Request Progression Interface(RPI).

2.1.4.3 OOMPI

OOMPI [29, 45] is an object-oriented interface to the MPI message passing library standard. It provides an MPI C++ class library that incorporates in it the C++/object oriented abstractions for message passing. It is developed as a thin layer that runs over the C bindings provided in the MPI-1 standard. Even though C++ bindings were later provided in the MPI-2 standard, OOMPI does not use those bindings as it was implemented before the MPI-2 was released officially.

In OOMPI data exists in the form of objects and communication of objects between the processes needs to take place. The C bindings provided in MPI-1 do not deal with objects. Thus OOMPI had to built an interface in it to communicate objects. The base types supported by OOMPI are char, short, int, long, unsigned char, unsigned short, unsigned, unsigned long, float and double.

2.1.4.4 Python Implementations

PyPar [34] and pyMPI [30] are the two current libraries that perform message passing in Python. However, they are both wrappers around MPICH and are primarily responsible for tasks such as serializing Python objects and sending them as a C char array. To the best of our knowledge, our MPI Python implementation is the only pure Python implementation and can work on any platform where a basic socket library is available.

2.2 Imperative and Functional programming

This section introduces two different approaches to programming:

- Imperative programming- A programming style that specifies an explicit sequences of steps to follow to produce a result.
- Functional programming- A programming style that is based on defining relationships between values in terms of functions.

2.2.1 Imperative programming

Imperative programming specifies a sequence of steps to produce a result. Most of the languages developed have been imperative in style as most computers use the von Neuman architecture, which is also imperative. FORTRAN, Basic, Pascal, C, C++, Java are some of the high level imperative languages. High level imperative languages allow five basic types of statements: assignment, looping, conditional branching, unconditional branching and procedure calls.

Let us consider an example of a simple factorial program using a high level imperative language C:

Imperative programming is a style of programming with side effects. The use of global variables causes different parts of the program to change due to changes in other parts of the program. Thus even though imperative languages allow decomposition of large problems into modules, these modules are not truly independent. They may have side effects on each other. In most imperative programming languages memory allocation and deallocation for data items has to be implemented manually by the programmer. This increases the scope of error and sometimes causes memory leaks. Most imperative programming type checking is not very strict most of the errors are not caught until runtime. Compiler cannot detect programming error as it would in a type safe language. In other to provide callback

```
int factorial(num:int){         /*function name         */
     int f;                     /*variable declaration  */
     for (f=1;num > 0;num--)    /*looping statement     */
           f *= num;            /*assignment statement  */
     return f;                  /*return statement      */
}
int main()
{
     ⋮
     factorial(10);             /*procedure call        */
     ⋮
}
```

Figure 2.5: Factorial function in C

functions that library functions call with certain parameters to obtain effects desired by the programmer, imperative languages like C we need to use function pointers to pass function addresses around. However the usefulness of these function pointers is limited as we cannot dynamically alter the behavior of the function.

2.2.2 Functional programming

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions. In contrast to imperative programming, functional programming emphasizes the evaluation of functional expressions, rather than execution of commands and eliminates several constructs often considered essential to imperative languages. For example, in strict functional programming, there is no explicit memory allocation and no explicit variable assignment. However, these operations occur automatically when a function is invoked thus removing any side effects of function evaluation. By disallowing side effects in functions, the language provides referential transparency which ensures that the result of a function will be the same for a given set of parameters no matter where, or when, it is evaluated. A powerful mechanism available in functional languages are higher-order functions which can take other functions as arguments, and/or return functions as results. These higher-order functions enable powerful abstractions and operations to be constructed. Functional languages like SML also have first class functions with closures meaning that when a function is passed as a parameter the function pointers also have a set of values stored with them. Thus the same address for the function pointer can be used with different sets of values unlike function pointers in imperative languages.

Let us consider an example of a simple factorial program using a functional language SML:

```
fun  factorial(0)  = 1 |
     factorial(n)  = n * factorial(n-1)
```

Figure 2.6: Factorial function in SML

We see that it uses a recursive call to the function *factorial*. We do not use any variable and the return value of the function is the result. Unlike the imperative program this does not have any side effects since it does not use any variables.

2.2.3 Standard ML (SML)

ML (standing for ”Meta-Language”) [36] [31] is a general-purpose mostly functional programming language developed by Robin Milner and others in the early 1980s at Edinburgh University. ML is an impure functional language, because it permits imperative programming, unlike pure functional programming languages such as Haskell, [23]. Features of ML include automatic memory management through garbage collection, a static type-safe polymorphic type system, type inference, algebraic data types, pattern matching, and a sophisticated module system with functions on modules (functors).

Type inference is a technique which allows the compiler to determine from the code the type of each variable and symbol used in the program, without having to explicitly declare them. This allows for a compact, yet easily readable code. Algebraic data types allow to define new types as data structures, and combine them in a hierarchical fashion. Pattern matching is the capacity for a function to deconstruct algebraic data types, into its different subtypes, in order to apply a particular computation for each subtype. Today there are several languages in the ML family; among them the most popular are SML (Standard ML) [32], F# [2] and Caml [1].

2.2.4 Concurrent ML (CML)

Concurrent programming is the task of writing programs consisting of multiple independent threads of control called processes. These processes are executed in parallel on a single processor. Concurrent ML (CML) [38] is an extension to SML that facilitates concurrent programming. CML is completely written in SML and is implemented on top of SML/NJ.

The basic modes of communication and synchronization in CML are shared memory access or explicit message passing. The messages are passed between the threads through a “typed” communication channel. By “typed” we mean that the thread can receive information of only single type through a channel. However union types can be used to allow multiple types of messages.

Unlike parallel programming which achieves parallel execution in a multiprocessor environment, concurrent programming achieves parallel execution in a uniprocessor environment.

2.3 Efforts in development of MPI for functional languages

Wrappers to traditional implementation of MPI in conventional language like C are available in Caml such as OCamlMPI and ScaMPI. A brief overview of the wrappers implemented in functional languages is provided in the following section.

2.3.1 OCamlMPI and ScaMPI

OCamlMPI [46] provides Caml bindings for a large subset of MPI functions. OCamlMPI was implemented for the Starfish project at Technion - Israel Institute of Technology. Starfish which is written in OCaml is a fault-tolerant system for running parallel MPI programs on clusters of workstations/PCs. OCamlMPI was implemented specifically for the Starfish architecture. It supports point-to-point and collective operations. Even though OCaml has higher order functions OCamlMPI does not make use of them as it is a wrapper around C and C does not support this feature. The MPI_Collective Operations are restricted to a predefined set as in MPICH.

Another MPI interface for Caml is ScaMPI. ScaMPI (Simple Caml to MPI interface) [8] is a library allowing Caml programs to make calls to MPI-1 communication routines. It is not a complete implementation of the MPI and provides a few calls for basic communication primitives. These primitives include send, receive, scatter and gather. ScaMPI maps function calls from Caml to C and supports three datatypes: integer, float and strings. Caml being a functional language has automatic storage. During type conversion from Caml to C, memory has to be allocated for the data items. The function calls in Caml are identical to the ones available for C. This limits the functionality of ScaMPI. It is restricted from taking advantage of the constructs available in Caml.

Some of the function prototype definitions for ScaMPI are:

external ssend_int: int → pid → tag → unit = “mpi_ssend_int”

external ssend_float: int → pid → tag → unit = “mpi_ssend_float”

external ssend_string: int → pid → tag → unit = “mpi_ssend_string”

These function calls take the same arguments as MPICH. An overhead is present to translate these calls into C calls.

We observe that there is a lack of native implementations of MPI in a functional language and there is none available in SML. This has restricted the development of parallel programs in SML and programmers have not been able to take advantage of the functional style in parallel programming. This has provided us with the motivation for development of SMPI, a native implementation of MPI in SML which will be discussed in detail in the following chapters.

Chapter 3

Native Implementation of MPI in SML

3.1 Introduction

Traditionally, conventional languages have been used for developing MPI as they offer compatibility with existing systems along with high performance. However, they have some inherent defects at the most basic level making the implementations ”fat and weak”, [11]. Such languages do not provide strict type checking and automatic storage management for dynamic data, thus making them unsafe. They also, to some extent, lack modularity, [12]. SMPI, a message passing interface developed using an advanced programming language, addresses these shortcomings by incorporating the functional style. It provides a well structured implementation of MPI using SML of New Jersey.

3.2 SMPI architecture

In this section we briefly explain the SMPI architecture (see Fig. 3.1)

The SMPI architecture is divided into four distinct layers. These layers are:

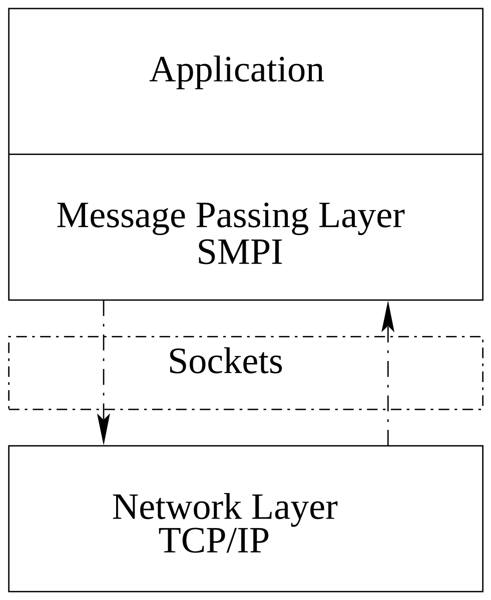

Figure 3.1: SMPI architecture

Application layer

The *application layer* consist of programming code and sets of rules to solve problems. SMPI supports the Single Program Multiple Data (SPMD) model of parallel computing, wherein a group of processes cooperate by executing identical program images on local data values. The SMPI application program makes calls to SMPI communication routines to communicate between processes.

Message Passing Layer

The *message passing layer* provides the communication mechanism required by processes to pass messages between them. In this architecture the SMPI library acts as the Message Passing Layer. It consists of functions that facilitate the transfer of messages between the processes.

Socket Layer

The *socket layer* provides the application program interface for the Network layer. Reliable communication is ensured by the use of TCP/IP sockets as it is connection-oriented service.

Network Layer

The *network layer* forms the underlying communication channel for transporting the messages from one processor to another. The Ethernet channel is used for this purpose.

3.3 SMPI

SMPI is a library for parallel programming in SML. It allows users to develop parallel applications while reaping the benefits of a functional language. SMPI provides functions for performing communication between different processes running on different machines.

3.3.1 Environment description

SMPI application programs are executed over a set of processors which are specified during startup. The processor that initiates the execution of the application program is called the *root processor*. When a program begins execution, TCP connections between the processors are set up and these connections last throughout the execution of the program. Each processor is connected to every other processor taking part in the computation. The identities of the processors participating in the computation are provided as a argument when starting the application program.

Fig.3.2 illustrates a four node environment for a parallel program using SMPI. Each node is connected to each of the other three nodes by a TCP connection. As the root processor, $N1$ starts a copy of the program in all other processors. The results of collective operations are returned to the root processor.

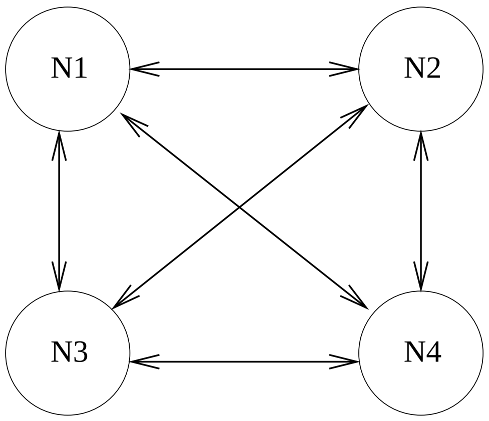

Figure 3.2: SMPI Environment

3.3.2 Communication handle

Every SMPI function's argument list includes a communication handle. The communication handle represents the set of processors participating in the computation. At each node the communication handle contains information needed to communicate with all the other participants. A communication handle is a list of machine names, rank i.e an unique identifier that is assigned to the machine and array of sockets used to communicate. The processors are assigned unique identifiers and this information is stored in an handle of the type ”comm”.

```
type comm ={nodes:string list, rank:int,
sockets:SOCK.clientSocket option ref array}
```

Here

- *rank* is the unique identifier assigned to the machine
- *nodes* is the list of all the nodes involved and
- *sockets* is the list of sockets used for communicating

The nodes and the unique identifier rank are assigned during the initialization process. The socket handles are populated after all the connections are made. For details refer to chapter 4.

3.3.3 Communication Primitives

Like other MPI architectures SMPI provides both point-to-point communication and collective operations.

3.3.3.1 Point-to-Point communication

Point-point communication is achieved through simple MPI.Send and MPI.Recv functions.

val Send : string * int * comm → unit

val Recv: int * int * comm → string

The parameters for the MPI.Send function are

- buffer to be sent (string)
- destination identifier
- the communication handle

The parameters for the MPI.Recv are the:

- length of the buffer to be received
- source identifier
- the communication handle

The return type of this function is the value being received.

3.3.3.2 Collective operations

SMPI collective operations are performed by calling the point-to-point communication functions i.e. MPI.Send and MPI.Recv. It is just a set of calls to the point-to-point communication primitives grouped together. The various collective operations have been modeled after those used in MPICH, [7]. The algorithms used are tree based algorithms, which are discussed in detail in chapter 4.

val Barrier : comm → unit

val Bcast : string * int * comm → string

The parameter for Barrier is the communicator handle. This blocks the program running in all the processors until all have reached the point where this Barrier call is made. Broadcasting values to all the processors is achieved by the Bcast function call. Bcast takes the communication handle,the value to be broadcasted along with its length as parameters and returns the value it receives.

3.3.3.3 Array Operations

SMPI supports all the point to point and collective operations mentioned above for arrays. Additionally, the collective operation Reduce is also supported for an array. The current implementation of SMPI only supports Real arrays. We defined the type MPI.REAL64 which is a tuple consisting of functions used to manipulate Real64Array and Real64ArraySlice.

val SendArr : ’a * int * comm * dt → unit

val RecvArr : ’a * int * comm * dt → int

val BcastArr : ’a * comm * dt → ’c

val ReduceArr : ’a * (’b * ’b → ’c) * comm * dt → ’b

The parameters for the MPI.SendArr function are

- ’a ArraySlice to be sent
- destination identifier
- the communication handle
- datatype of the ArraySlice (eg. MPI.REAL64)

The parameters for the MPI.RecvArr are the:

- ’a ArraySlice to be populated with the received array

- source identifier
- the communication handle
- datatype of the ArraySlice (eg. MPI.REAL64)

The parameters for the MPI.BcastArr function are

- ’a ArraySlice to be sent
- the communication handle
- datatype of the ArraySlice (eg. MPI.REAL64)

The parameters for the MPI.ReduceArr are the:

- ’a ArraySlice to be reduced
- custom function defining the reduce operation
- the communication handle
- datatype of the ArraySlice (eg. MPI.REAL64)

In order to efficiently send and receive Real arrays we had to modify the SML Basis library and SML runtime library to support the required operations on Real arrays. To the Socket structure in the SML Basis Library we added the following calls:

val sendRealArr : $'af, activestream$ sock * Real64ArraySlice.slice $\rightarrow$ int

val recvRealArr : $'af, activestream$ sock * Real64ArraySlice.slice $\rightarrow$ int

We modified both the signature and the implementation for the Socket structure. The implementation of the Socket structure in the SML Basis library uses the Unsafe C Interface [37] to call the required functions in the SML runtime library. The SML runtime library is written in C and is dynamically called by the SML basis library. In the SML runtime library we have added functions to efficiently send and receive the bytes of a SML Real array. Modifications to the SML Basis Library and the SML runtime library are available as a patch file in our source distribution.

Our Reduce algorithm is a tree based algorithm and is discussed in detail in chapter 4. We added commonly used Reduce operations MAX_REAL64, MIN_REAL64, MUL_REAL64, ADD_REAL64 and SUB_REAL64 to SMPI library. In order to improve the performance of these we implemented them in the SML runtime library and called them using the Unsafe C Interface. In addition to these the user is allowed to define their own custom reduce operation and pass them to the Reduce method.

3.3.4 Structure of a SMPI program

Every SMPI program must essentially have the following structure.

```
structure Test =struct
    fun main(pgmName, argv) =
    let
        val MCW = MPI.Init(argv)
            ⋮
    in
    ⋮
    end
    val _ = SMLofNJ.exportFn(”test”,main)
end(*test*)
```

We provide a script mpirun used to run the heap image created by the above sml program. More details in chapter 4.

3.4 Implementation issues

SML does not inherently have any constructs that supports parallelism. SMPI adds multiprocessor support to SML by providing a set of routines that can be included in the form of a library. Since there are no bindings for SML in the MPI standard the design of SMPI is modeled after MPICH, a C implementation of MPI, [7].

In large computation nearly all the data that is passed around are reals. However the Socket structure in the SML Basis Library supports only sending and receiving of Word8VectorSlice and Word8ArraySlice. We added the ability to send and receive Real64ArraySlice by modifying the SML Basis Library and SML runtime library.

The SML Basis library does not use the MSG_WAITALL flag for the receive call in the runtime Socket implementation. This ensures that we do not have to loop until we receive the required number of bytes. So we modified the runtime library to use the MSG_WAITALL flag. In our modifications to runtime we also loop around the send call to ensure that all the required bytes are sent.

Modifying the Basis library and runtime required boot strapping the compiler. For more details refer B.

3.5 SMPI Library

The SMPI library provides a set of structures that can be included in SML programs to incorporate MPI features into SML. The structures defined in this library are listed below:

Structure	Description
MPI	Main structure containing all the user calls
SOCK	Socket utility
MYTIMER	Used for timing calls

The MPI structure contains the function calls described in section 3.3 along with several other constructs to setup the SMPI environment, perform the initialization process and perform the termination process by closing all the connections. It also contains the functions required to perform the point-to-point and collective operations. The SOCK structure defines the underlying reliable communication interface using sockets. The MYTIMER structure defines the calls to calculate elapsed time for function calls. The interface for all these structures along with the description is provided in the the appendix.

Chapter 4

Message Passing Algorithms

This chapter describes the implementation details for various MPI primitives and explains the algorithms used for communication operations. The collective operations described here are: broadcast, reduce and barrier. Broadcast and reduce use a tree based algorithm which is similar to the one used in MPICH. Our C, SML and Python implementations are based on the following algorithms.

4.1 MPI Primitives

4.1.1 Initialization and Finalization

The initialization routine $MPI\_Init()$ is responsible for creating sockets between all pairs of participating processes. Since a single socket is to be created between any pair of processes, this can be achieved by ensuring that every process connects to processes with rank greater than itself. This is illustrated in Algorithm 1. The sending and receiving of a 1-byte message in Algorithm 1 is to ensure that a race condition does not occur. Figure 4.1 illustrates the relative timing of the communication steps involved when initializing a group of six processes. Solid arrow indicate matching $connect()$ and $accept()$ calls (the arrow points to the $accept()$ call). The shaded areas correspond to the periods of time when a process is blocked on a $recv()$ call waiting for a 1-byte synchronization message. The synchronization message is essential in order to ensure that the $accept()$ call in each process receives $connect()$ requests from processes with monotonically increasing ranks. The synchronization message is depicted by a dashed arrow from the previous rank process. If not for this synchronization message, it might be possible that the solid arrow from process 1 to process 2 arrives at process 2 before the arrow from process 1 as illustrated in Figure 4.2.

Algorithm 1 MPI Initialization

```
$size \leftarrow Size(MCW)$
$myrank \leftarrow Rank(MCW)$
**for** $r = 0$ to $myrank$ **do**
    SockVec[r] = accept()
**end for**
**if** $myrank > 0$ **then**
    **Receive** 1 byte message on socket $SockVec\,[myrank - 1]$
**end if**
**for** $r = myrank + 1$ to $size$ **do**
    **Connect** to node $r$
**end for**
**if** $myrank < size - 1$ **then**
    **Send** 1 byte message on socket $SockVec\,[myrank + 1]$
**end if**
```

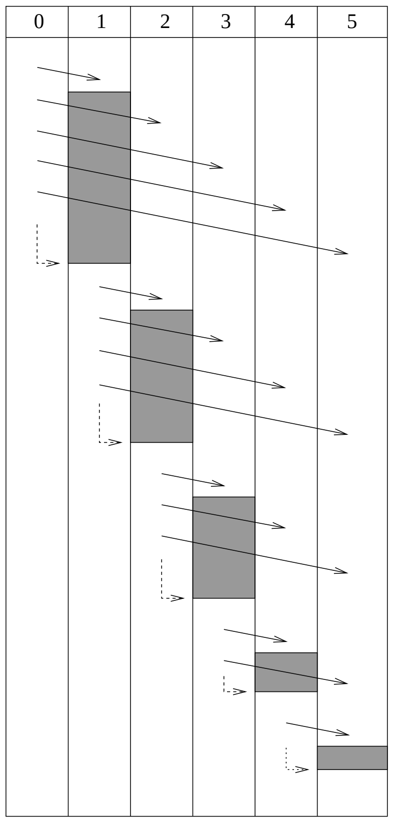

Figure 4.1: Initialization Algorithm

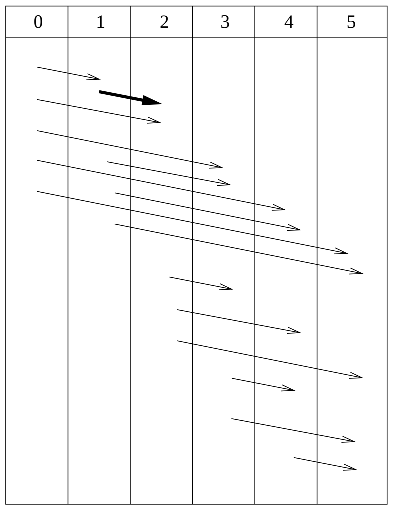

Figure 4.2: Race condition in improperly implemented initialization.

MPI_Finalize() is responsible for cleaning shutting down all the open sockets. In order to do this, for each socket, either process can close the socket.

4.1.2 Send and Receive

Our MPI_Send() and MPI_Recv() functions utilize the send() and recv() socket calls respectively. Additionally, they have to ensure that the required number of bytes have been sent and received and check for any error conditions. MPI_Send() additionally has to loop until the required number of bytes has been sent. MPI_Recv() does not need to do this since we utilize the MSG_WAITALL socket option (discussed in more detail in Chapter 6).

4.1.3 Broadcast Algorithm

Broadcast function when called broadcasts a message from the root process to all other processes participating in the computation. The algorithm used for broadcasting values to all the processors as mentioned above is a power-of-two based algorithm. It is also known as the broadcast tree algorithm. The root processor sends a data item to all the processes in the communicator handle MPI_COMM_WORLD. This is a very efficient way to send information as messages are sent in parallel. Since a tree algorithm is used, the number of communication phases required is proportional to the logarithm of the number of processes. If a sequential algorithm is employed, the number of communication phases required will be linearly proportional to the number of processes.

The broadcast is initiated by the root processor. At each time unit the number of processors receiving the information doubles since we use the commonly used binary tree based broadcast [47] as shown in Algorithm 2. Data transmission among processes that have already received the entire array (represented by $igot = 1$) is overlapped. Figure 4.3 illustrates the operation of the broadcast algorithm when there are 32 processes. The horizontal lines in each of the columns depicts when the process has finished receiving the array being broadcast. Algorithm 2 works even if the number of processors is not an exact power of two. Variable b keeps track of which level in the broadcast tree we are in since $log_2\ b$ represents the current step (assuming steps are counted from zero).

If the same communication was to be achieved by transmitting data from the root processor to all the other processors serially then the time taken to achieve this would be exponentially greater. The tree based algorithm allows for parallel transmission which reduces the execution time of this function as illustrated in Fig.

4.4.

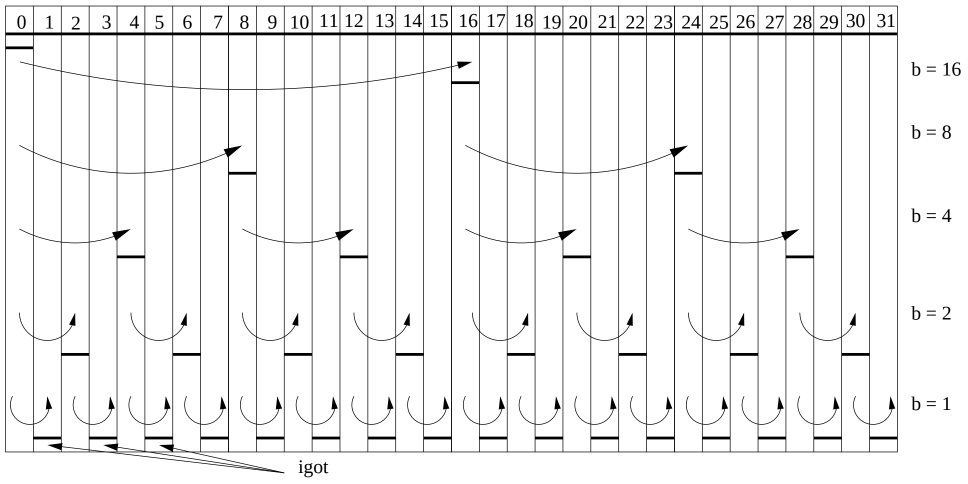

Figure 4.3: Broadcast Tree Algorithm

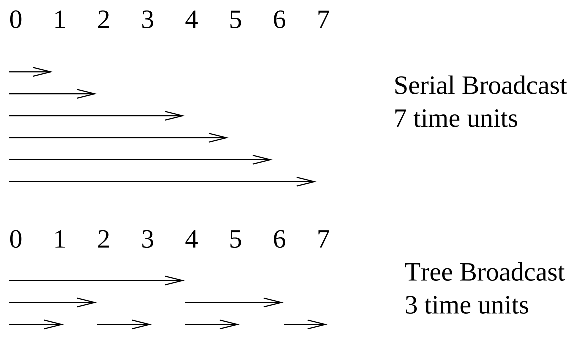

Figure 4.4: Efficiency for Tree Algorithm is Higher than Sequential Transmission

4.1.4 Reduce

The reduce operation reduces values on all processes to a single value. In the context of an array, the reduction is performed on a per-element basis resulting in a

Algorithm 2 Broadcast from root to all other processes

```
$size \leftarrow Size(MCW)$
$myrank \leftarrow Rank(MCW)$
$igot \leftarrow 0$
$b \leftarrow 1$
**if** size = 1 **then**
  return
**end if**
**while** $b < size$ **do**
  $b = b**2$
**end while**
**if** myrank = 0 **then**
  $igot \leftarrow 1$
**end if**
**while** $b > 1$ **do**
  $b \leftarrow b/2$
  **if** $igot = 1$ **then**
    **if** $myrank + b < size$ **then**
      Use MPI_Send() to send entire array to rank $myrank + b$
    **end if**
  **else if** $myrank\%b = 0$ **then**
    Use MPI_Recv() to receive entire array from rank $myrank - b$
    $igot \leftarrow 0$
  **end if**
**end while**
```

reduced array with the same length of the source arrays. The implementation uses a simple tree algorithm as illustrated in Algorithm 3.

The Reduce algorithm works by having all processes at the lowest level of the tree send their arrays to their parent processes. The parent processes reduce the received arrays with their own array using either a pre-defined function or a user-defined function and pass on the results to their parents and so on until the root process receives the fully reduced array.

Figure 4.5 illustrates this process in a system with 15 processes. The reduce operation is performed in a bottom-up fashion. The number of communication phases required is proportional to the logarithm of the number of processes. At each phase, nodes at a certain level communicate with nodes at the higher level. The ordering of the phases is illustrated by the labels of the arrows in the figure. The reduce operation is performed on all the non-leaf nodes (depicted with a + sign in the node). At the end of the algorithm, the root process contains the fully reduced array.

4.1.5 Barrier

The Barrier function synchronizes all the processes. When this function is called, the processes are blocked until all the processors reach this point. This is achieved by the following algorithm.

The algorithm synchronizes the processes by performing two circular passes [47]. In each pass, every processor waits to receive a token from the previous process and after receiving the token sends it to the next process. Process 0 performs the same steps in reverse order to avoid a deadlock. The first pass ensures that all processes arrive at the same point in the code. The second pass allows all the

Algorithm 3 Reduce array *buf* on root-process

```
$size \leftarrow Size(MCW)$
$myrank \leftarrow Rank(MCW)$
**if** size = 1 **then**
    return
**end if**
$myparent \leftarrow \left\lceil \frac{myrank}{2} \right\rceil - 1$
$child1 \leftarrow 2 * myrank + 1$
$child2 \leftarrow child1 + 1$
**if** $child1 \geq size$ **then** {If I am a leaf node}
    Use MPI_Send() to send entire array to rank *myparent*.
**else** {If I am an internal node}
    Define *tmpbuf* and *resbuf* to be arrays of length equal to that of *buf*.
    Let $op$ be the reduce operation to be performed.
    **if** $child1 < size$ **then** {If child1 exists}
        Use MPI_Recv() to receive the array sent from process *child1* into *resbuf*.
        $resbuf \leftarrow resbuf$ op $buf$
    **end if**
    **if** $child2 < size$ **then** {If child2 exists}
        Use MPI_Recv() to receive the array sent from process *child2* into *tmpbuf*.
        $resbuf \leftarrow resbuf$ op $tmpbuf$.
    **end if**
    **if** $myrank > 0$ **then** {If I am a non-root process}
        Use MPI_Send() to send the array *resbuf* to process with rank *myparent*.
    **end if**
**end if**
```

Algorithm 4 Barrier Synchronization Algorithm

```
$size \leftarrow Size(MCW)$
$myrank \leftarrow Rank(MCW)$
$next = (myrank + 1)$ MOD $size$
$prev = (myrank + size + 1)$ MOD $size$
**if** $myrank == 0$ **then**
    Send a 1-byte message to rank $next$.
    Receive a 1-byte message to rank $prev$.
    Send a 1-byte message to rank $next$.
    Receive a 1-byte message to rank $prev$.
**else**
    Receive a 1-byte message to rank $prev$.
    Send a 1-byte message to rank $next$.
    Receive a 1-byte message to rank $prev$.
    Send a 1-byte message to rank $next$.
**end if**
```

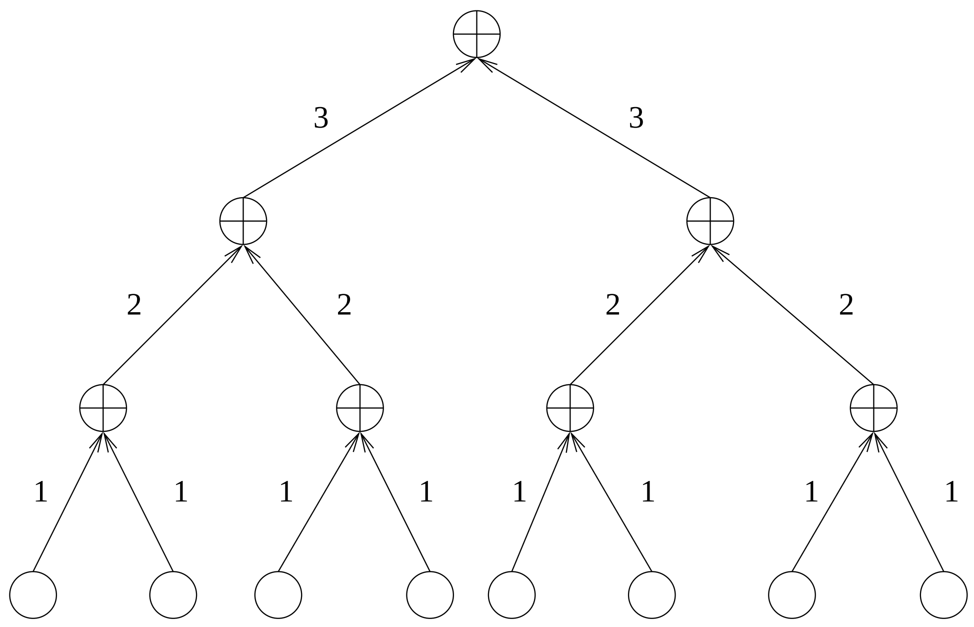

Figure 4.5: Reduce Tree Algorithm

processes to continue execution.

Figure 4.6 illustrates the relative communication timing when MPI_Init() is invoked for a system with 4 processes. The horizontal dashed lines depict when a process has completed its barrier and can proceed with execution. Figure 4.7 illustrates the effect of a delay in reaching the barrier in process 3. Despite the delay, all the processes exit the barrier at nearly the same time. Figure 4.8 illustrates the effect of a delay if a single pass barrier was used and a similar delay in process 3 existed. In this case, process 1 exits the barrier much before the other processes.

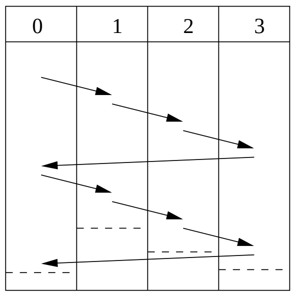

Figure 4.6: 2-Pass Barrier

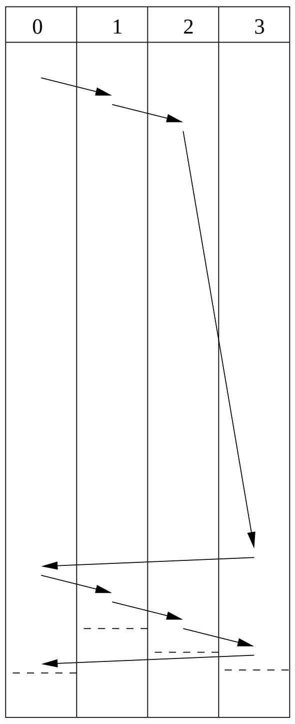

Figure 4.7: 2-Pass Barrier with delay in process 3

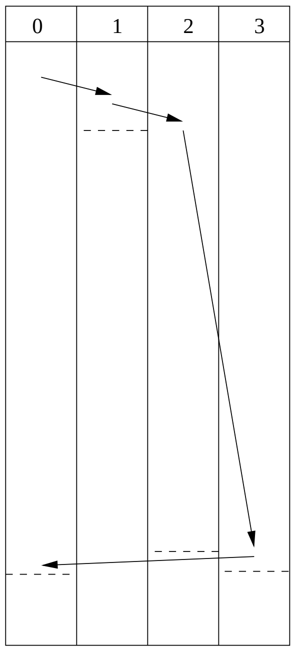

Figure 4.8: 1-Pass Barrier with delay in process 3

Chapter 5

Implementation Details

This chapter discusses the features of SML that can be utilized in user programs and have been used in the SMPI implementation to make it efficient, concise and robust.

5.1 SML constructs

5.1.1 SML Module system

SMPI takes advantage of the advanced module system provided by SML and provides a clear logical separation. The structures MPI, SOCK, TIMER defined in SMPI are logically separate and are developed and tested independently. The structures contain functions which define the various layers of SMPI. The MPI structure provides the application programming interface while the SOCK structure provides the underlying reliable communication interface. The structuring mechanism used in SMPI makes the implementation easy to understand.

5.1.2 Type Safe

Since SML is a type safe languages most of the errors can be detected at compilation time. Due to this a SML program rarely crashes unless a severe fault like running out of memory occurs, [5].

5.1.3 Higher-order Functions

SML features include higher order functions. This allows functions to be passed as arguments, stored in data structures and returned as results of function calls. This provides the ability to pass custom functions at runtime. C provides a similar concept through function pointers. However type of the function needs to be specified in C which is not required in SML. In our SMPI implementation MPI.ReduceArr are higher order functions since they accept user defined functions at runtime. Reduce functions apply the user defined function to the array contained at each node to successively reduce the array until the fully reduced array is received at the root node.

5.1.4 Automatic tuple expansion

Tuples can be expanded automatically into their components in SML. For example in SMPI the function *Rank* is defined to obtain the rank when the communication handle which is defined as a tuple containing the rank, the list of nodes and an array of sockets is passed to it. The function Rank:

```
fun Rank({rank=r,nodes,sockets}) = r
```

In the calling program the function *Rank*:

```
MPI.Rank(comm)
```

The above example illustrates how automatic tuple expansion can make functions concise and easy to read.

5.1.5 Automatic Garbage Collection

SML has an automatic garbage collector in which data that is no longer referenced is automatically deallocated. We do not need to free or allocate memory explicitly like in C. This makes the code simpler, cleaner and more reliable.

5.1.6 Error Handling

SMPI uses the exception handling mechanism provided by SML. During runtime the exception handling mechanism which is similar to the ones available in C++ and Java throws exceptions whenever an error or a faulty state is reached. C does not support exception handling. The exception handling used in SMPI enhances its functionality.

```
fun createServerSocket() =
let
        val serverSocket = INetSock.TCP.socket()
        val _ = Socket.Ctl.setREUSEADDR(serverSocket,true)
        val _ = Socket.bind(serverSocket, INetSock.any PORT)
        val _ = Socket.listen(serverSocket, 5)
in
        serverSocket
end
handle _ => raise "Fail: Could not create server socket"
```

5.1.7 Tail Recursion

Recursive functions are used widely in functional languages. These functions consume stack and can fail if the recursion goes on for too long. However by using tail recursion [24], where we do not maintain the return state in the call stack we can improve efficiency. In the SMPI implementation we have used tail recursion in all the places where we iterate.

5.1.8 Interfacing with C

Using Unsafe.CInterface structure C functions can be registered into SML. In SMPI, we modified the Socket structure provided by the SML Basis library to include routines that enable efficient sending and receiving of Real arrays. We implemented these routines as part of the SML runtime in C and included them in the SML basis library using the Unsafe.CInterface. Our MYTIMER module also uses the Unsafe C Interface to call the getTimeOfDay function provided by the C standard library.

Chapter 6

Experimental Results

6.1 Experimental Setup

We compared the performance of C, SML and Python implementations of MPI_Send(), MPI_Bcast(), MPI_Reduce() and MPI_Barrier(). Despite MPICH [16] already being a complete C implementation of the MPI-1 standard [4], we decided to implement these primitives in our own C implementation, since it would be unfair to compare the performance against MPICH, since MPICH is a highly portable implementation and in order to work with numerous architectures such as SMP systems, Myrinet and InfiniBand etc., it has numerous layers of abstractions, which has a noticeable impact on performance. MPI primitives can be classified as Point-to-Point or Collective operations. MPI_Send() is representative of Point-to-Point operations. These operations are between two processes and the performance is linear function of the message length. MPI_Bcast() is a collective operation since it can involve two or more processes. Since most collective operations use binary tree based algorithms, the performance is a linear function of the message size and a logarithmic function of the number of participating processes. MPI_Barrier() is a also a collective operation. However, unlike other collective operations, its performance is a linear function of the number of participating processes. Since we use a double circular shift algorithm in the MPI_Barrier() implementation, every process only sends two bytes to its neighbor and received two bytes from its neighbor.

We evaluated the performance of our implementations on a moderately heterogeneous cluster. The cluster comprised of 13 nodes. 10 nodes were Athlon MP 2200+s, 1 was an Athlon MP 1600+ and 2 were an Athlon MP 1900+s. The nodes were connected via a 100Mbps ethernet network. All nodes had 2 GB of RAM. The same network was also used for the shared network filesystem (NFS) between the nodes. The operating system on all the nodes was Fedora Core 6 Linux. We used python-2.4.4-1 and smlnj-110.59 to run our Python and SML implementations respectively.

In order to be consistent, we used the gettimeofday() function in all our implementations, since all the implementations eventually use the gettimeofday() present in glibc (standard C library). In our SML implementation, we did this by using the unsafe C interface. In Python, gettimeofday() is already available as part of the standard library.

In our C and Python implementations, we used the MSG_WAITALL flag. This flag ensured that an entire message is received with a single call to the recv() socket function. However, in SMLNJ, this flag has not been implemented. In order to remain consistent with the C and Python implementations, we modified the source code for the SML runtime library to support the MSG_WAITALL flag.

All the data points in all the plots were chosen as the minimum of 20 trials in order to improve accuracy.

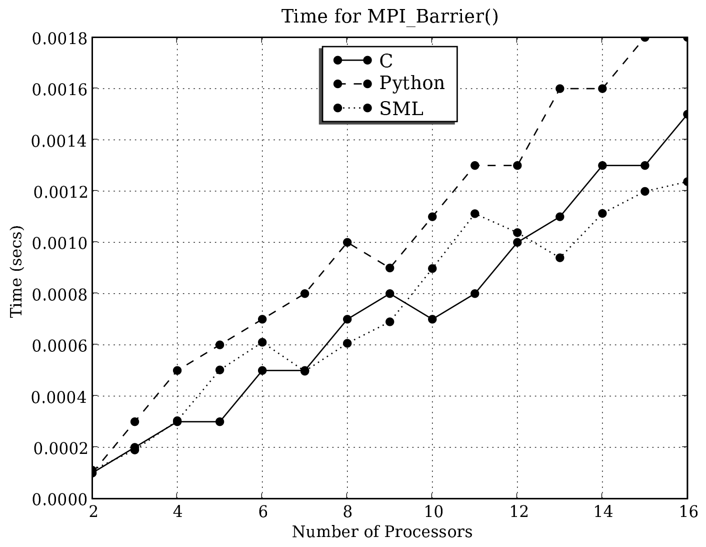

Figure 6.1: Comparison of Barrier performance.

6.2 Barrier

Figure 6.1 depicts the execution time for the MPI_Barrier() in the C, Python and SML implementations. Since, our Barrier call sends only four bytes of data among the nodes, the time for the barrier call significantly depends on performance of the language. Since, Python is purely interpreted, it performs slightly worse than our C implementation. On the other hand, the performance of SML very closely follows the performance of our C implementation. It can also be observed that for all the implementations, the time is linearly proportional to the number of processors.

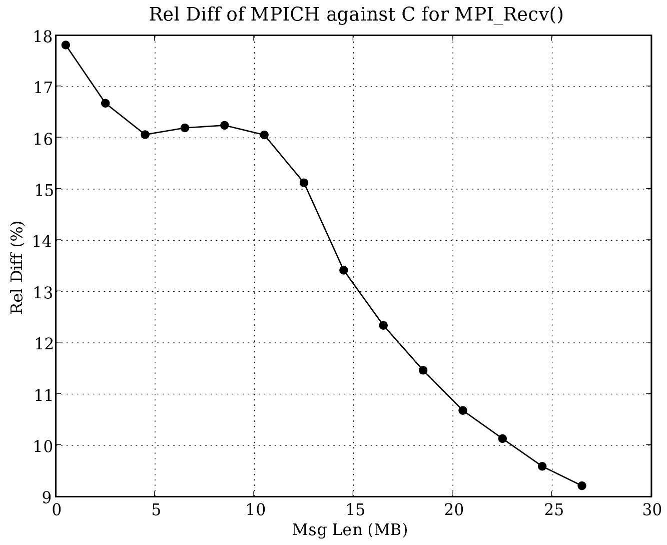

Figure 6.2: Comparison of MPICH and simple C MPI_Recv() implementation.

6.3 Point to Point Primitive

Figures 6.2 and 6.3 illustrate the performance of our C implementation against that of MPICH. The relative difference in wall time was determined by equation 6.1. A positive relative difference indicates that MPICH is performing worse than our C implementation.

$$\text{Relative Difference} = \frac{t_{\text{x}} - t_{\text{c}}}{t_{\text{c}}}, \text{ where x is either MPICH, SML or Python.} \quad (6.1)$$

It can be observed that for small message sizes, MPICH performs much worse than our C implementation. This is due to a constant cost involved in setting up MPICH. However, as the message size increases, the performance gap reduces. MPICH has a very large code base since it is a complete implementation of the

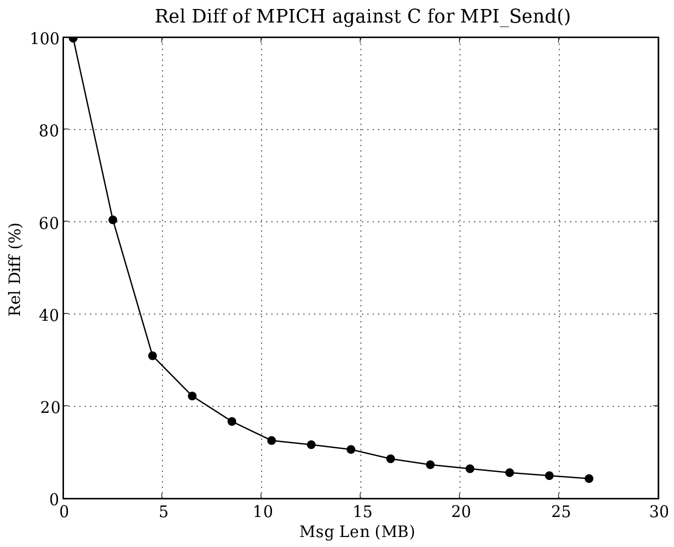

Figure 6.3: Comparison of MPICH and simple C MPI_Send() implementation.

MPI-1 standard. Further, MPICH has also been designed to work over multiple architectures. Hence, it is understandable for it to have a larger overhead. We as well as others [26, 27] have observed that message passing using native TCP sockets performs much better than MPICH. MPICH has been designed to work with heterogeneous workstations. It is even capable of working in a mixed endianness environment. Further, MPICH is capable of working with various network hardware. It achieves this by using an ADI (abstract device interface), which abstracts the underlying physical communication layer. Typically, MPICH's MPI calls are mapped onto MPID (MPI Device) calls and the MPID layer make calls to the ADI corresponding to the required physical communication layer. Due to the above reasons, we decided to compare our Python and SML implementations against our C implementation for the rest of this work.

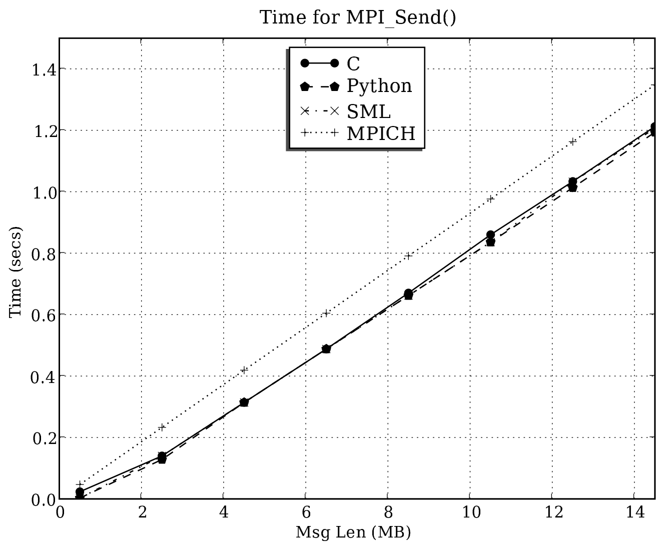

Figure 6.4: Wall Time for MPI_Send().

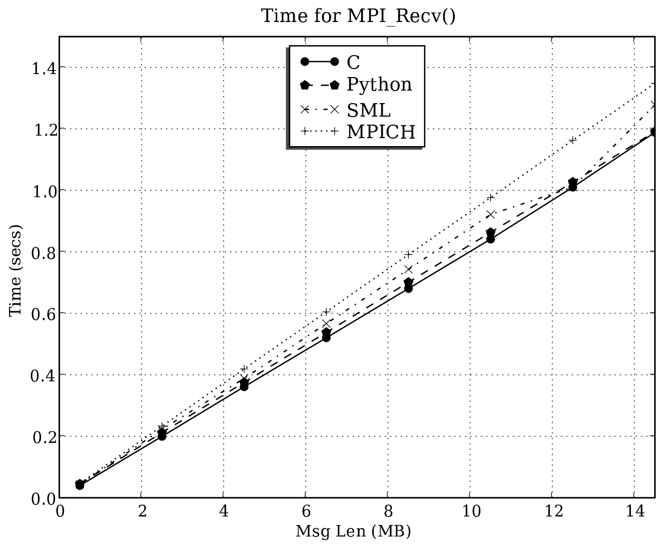

Figure 6.5: Wall Time for MPI_Recv().

Figures 6.4 and 6.5 plot the wall time of MPI_Send() and MPI_Recv() for all four implementations. It can be observed that the wall time is a linear function of the message size. The performance of MPICH is noticeably worse than the other implementations. In order to obtain a better comparison, we plotted the relative difference in wall time with respect to C. The relative difference was computed using equation 6.1. A positive relative difference indicates how much worse the implementation is compared to our C implementation.

Figures 6.6 and 6.7 plot the relative difference in wall time for Python, SML and MPICH against C. MPI_Recv() internally uses the recv() socket call. A non-blocking recv() call has to initially wait for data to be present in the operating system buffer. However, a send() socket call can immediately begin sending data. Due to this, we see slight irregularities in Figure 6.7.

6.4 Collective Primitive

In order to test the performance of collective operations, we implemented two of the most common MPI collective operations - MPI_Bcast() and MPI_Reduce(). MPI_Bcast() broadcasts the buffer from a root node to other nodes in the group of processes, while MPI_Reduce() performs a collective reduce operation and the fully reduced array is received at the root node.

6.4.1 Broadcast

Figures 6.8, 6.9 and 6.10 illustrate the wall time for MPI_Bcast in our C, Python and SML implementations respectively, when the number of processors is increased for a fixed message size. Since our Python, C and SML implementations use a binary

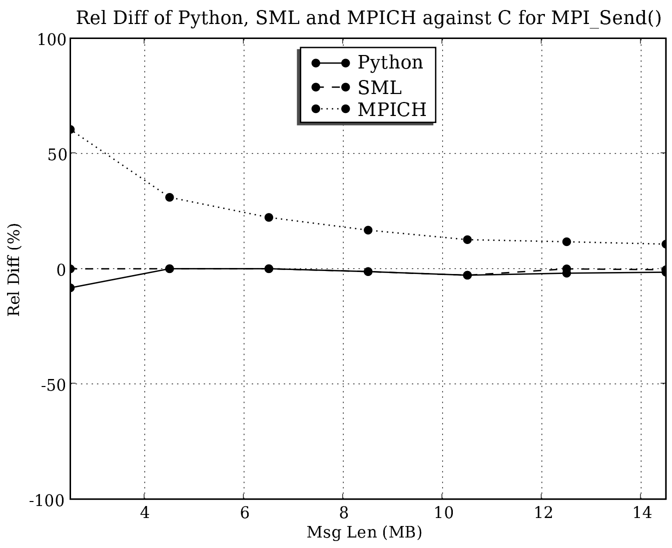

Figure 6.6: Relative difference of Python, SML and MPICH against C for MPI_Send().

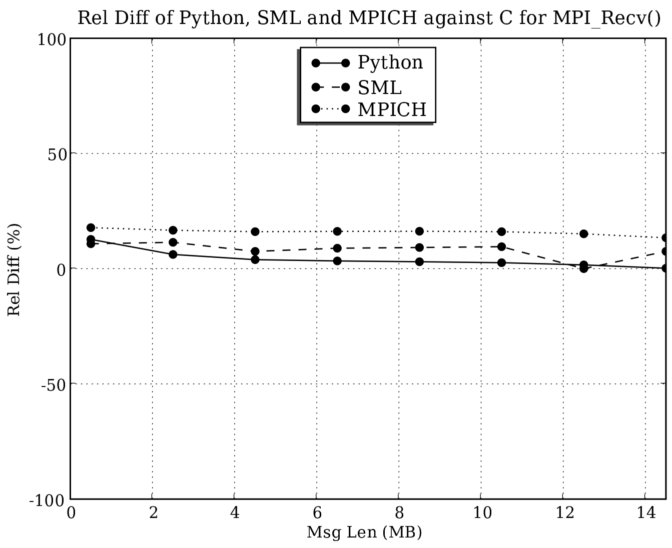

Figure 6.7: Relative difference of Python, SML and MPICH against C for MPI_Recv().

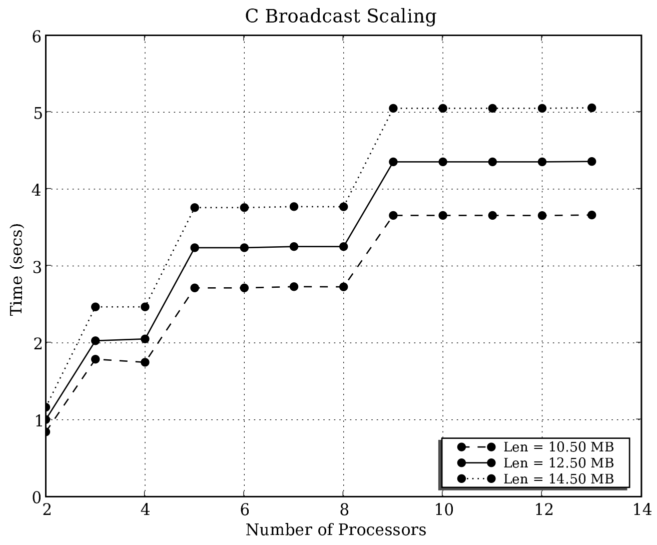

Figure 6.8: Scaling of MPI_Bcast() in simple C implementation.

tree based algorithm (as described in Chapter 4), the wall time of the MPI_Bcast() call increases distinctly when the number of processes becomes a power of two.

As soon as the number of processors exceeds a power of two, a new level in the tree is created, creating another level of sends/recvs to be performed. Figures 6.11, 6.12 and 6.13 illustrate the wall time for MPI_Bcast in our C, Python and SML implementations respectively, when the message size is increased for a fixed number of processes. It can be observed that the wall time is proportional to the binary log of the number of processes. P=2 corresponds to the lowest line, P=3, 4 corresponds to the next line, P=5, 6, 7, 8 corresponds to the next line and P=9, 10, 11, 12 and 13 corresponds to the highest line depicting the distinct characteristic of the binary tree based broadcast algorithm.

In order to portray the relative performance of the three implementations, we

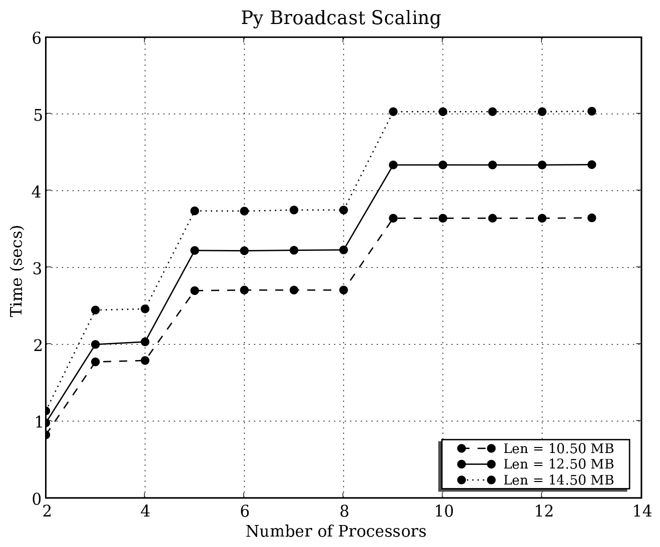

Figure 6.9: Scaling of MPI_Bcast() in Python implementation.

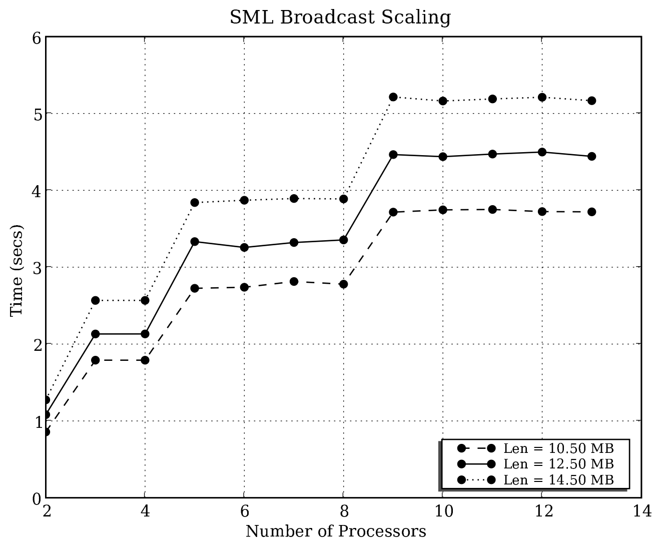

Figure 6.10: Scaling of MPI_Bcast() in SML implementation.

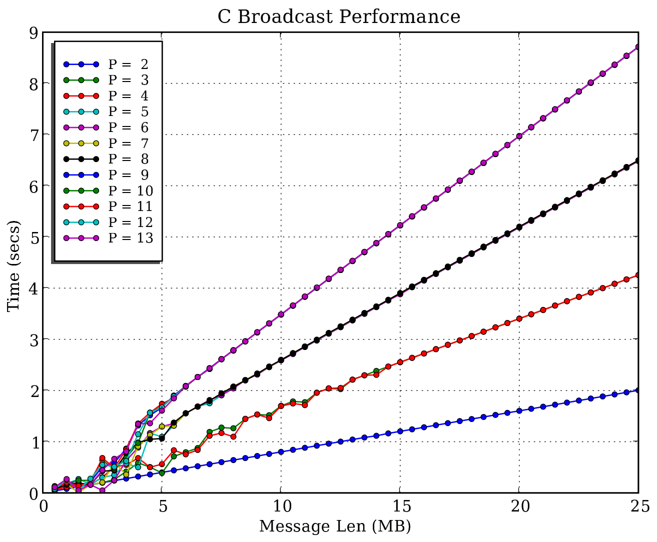

Figure 6.11: Performance of MPI_Bcast() in simple C implementation.

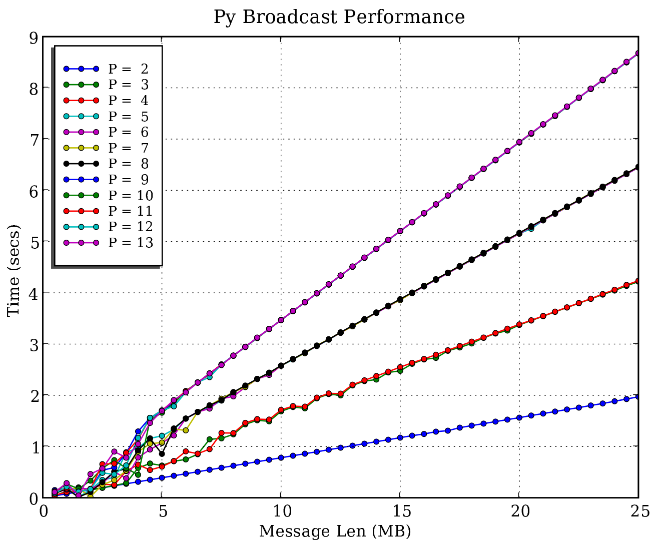

Figure 6.12: Performance of MPI_Bcast() in Python implementation.

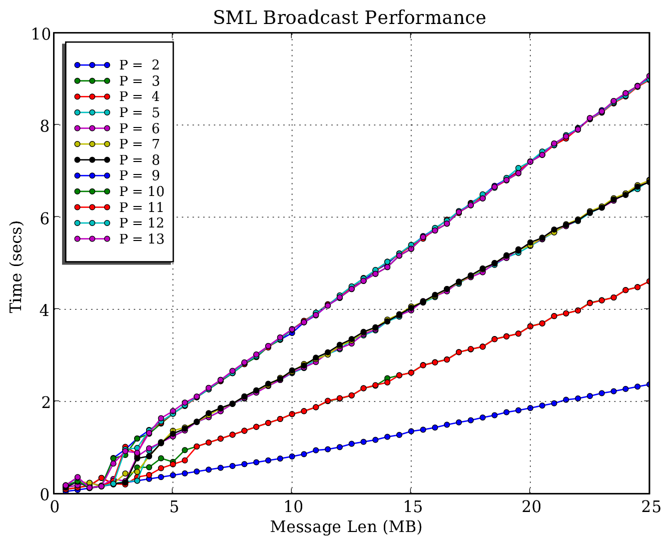

Figure 6.13: Performance of MPI_Bcast() in SML implementation.

the plot the relative difference in wall time for the MPI_Bcast() call for our Python and SML implementations against our C implementation. The relative difference is computed by equation 6.1 Figure 6.14 plots the relative difference for an increasing number of processors and a fixed message size. As the message size increases, the plots stabilize and it can be observed that the SML implementation performs marginally better than the Python implementation. Figure 6.15 plots the relative difference for an increasing message size and a fixed number of processors.

6.4.2 Reduce

Figures 6.16, 6.17 and 6.18 illustrate the wall time for MPI_Reduce() in our C, Python and SML implementations respectively when the message size is increased for a fixed number of processes. The bands that were observed in MPI_Bcast()'s

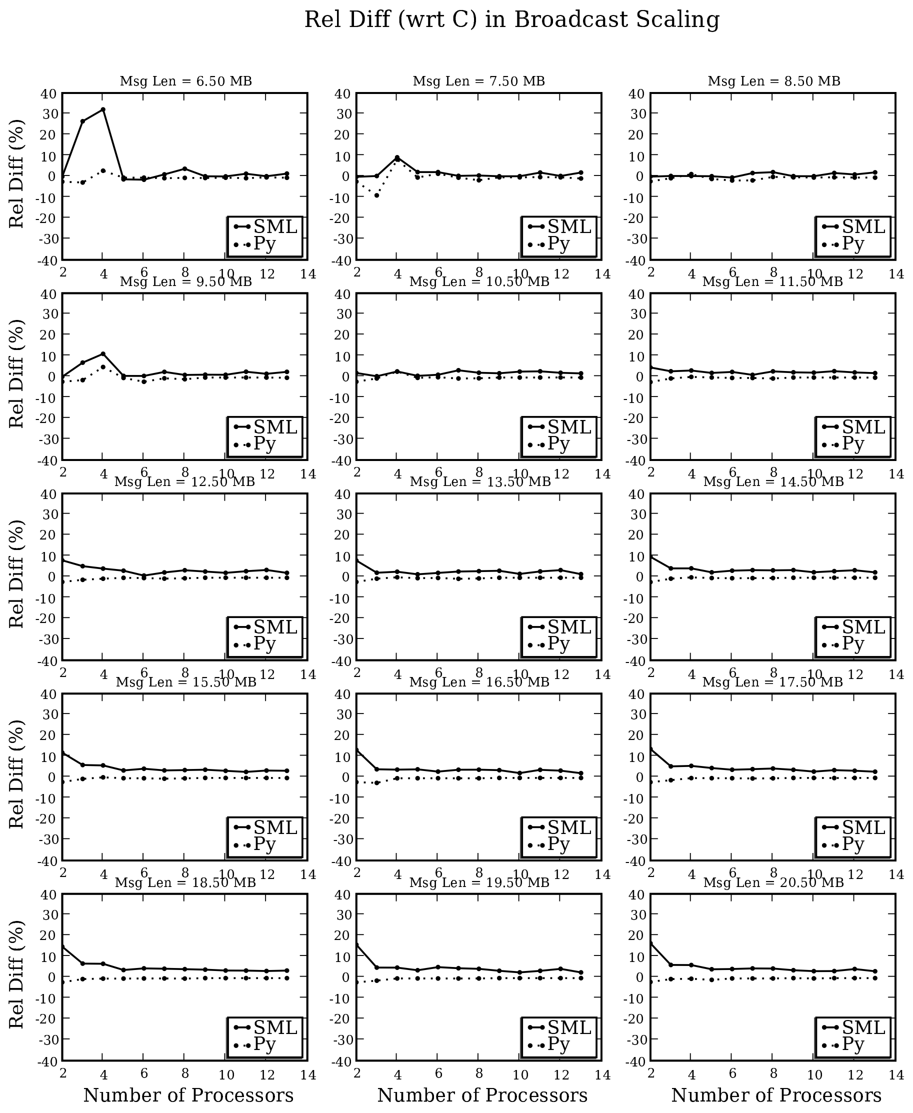

Figure 6.14: Relative Difference (wrt C) in scaling of MPI_Bcast().

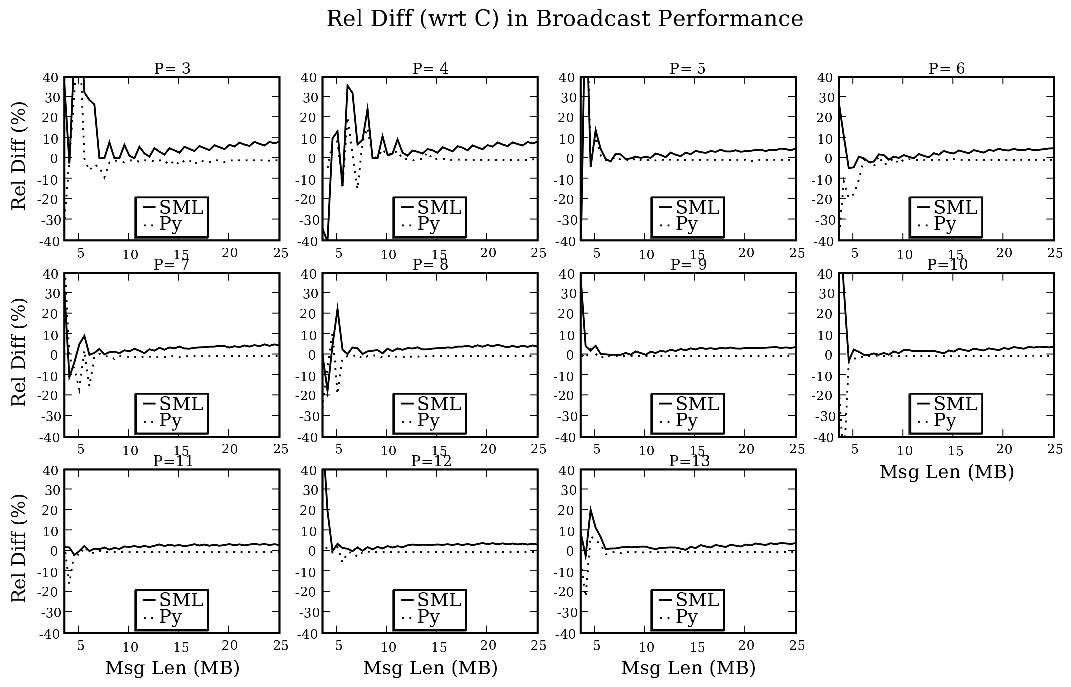

Figure 6.15: Relative Difference (wrt C) in performance of MPI_Bcast().

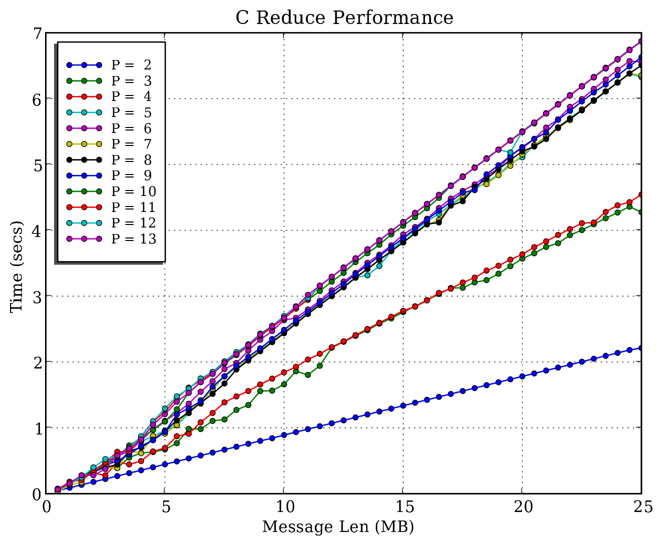

Figure 6.16: Performance of MPI_Reduce() in simple C implementation.

performance (Figures 6.11, 6.12 and 6.13) are not so distinct in the case of the MPI_Reduce() function. The reason for this is that heterogeneity in the CPUs of the nodes in the cluster. The MPI_Reduce() function is a lot more CPU intensive than the MPI_Bcast() function due to the arithmetic operation that is performed on the entire array at each processor. In the case of a homogeneous cluster, a banding similar to that of MPI_Bcast() can be expected even in the case of MPI_Reduce().

Figures 6.19, 6.20 and 6.21 illustrate the wall time for MPI_Reduce() in our C, Python and SML implementations respectively, when the number of processors is increased for a fixed message size. The step pattern is not as distinct as in the case of MPI_Bcast() due to the heterogeneity of the cluster.

Our MPI_Reduce function provides for a few predefined operations that can be performed such as addition, subtraction, multiplication, minimum and maximum.

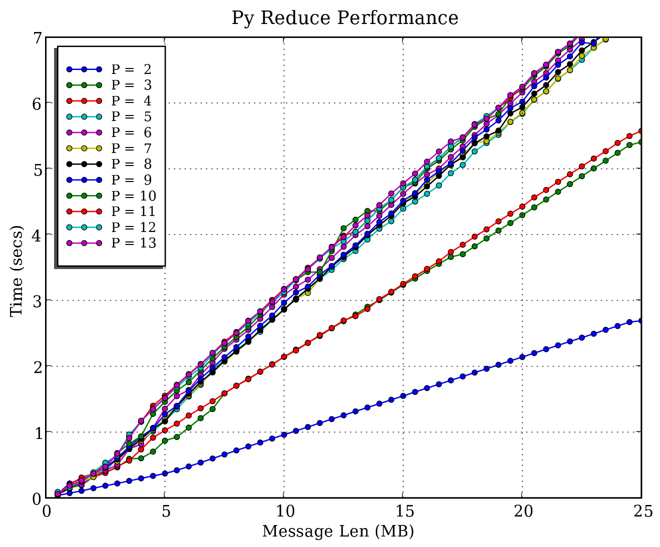

Figure 6.17: Performance of MPI_Reduce() in Python implementation.

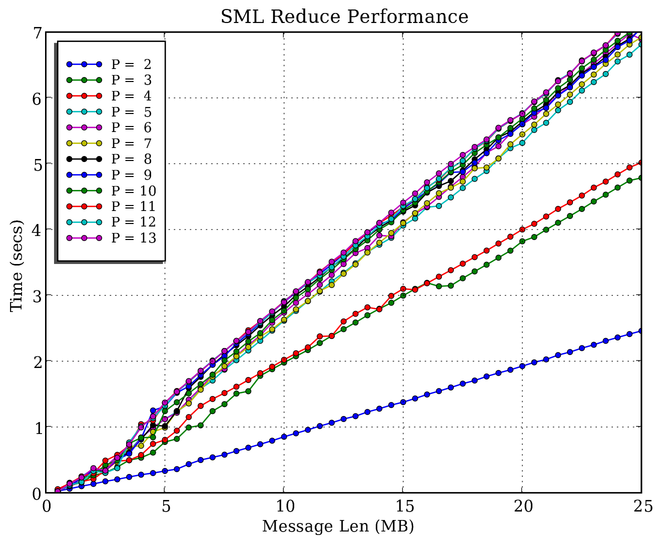

Figure 6.18: Performance of MPI_Reduce() in SML implementation.

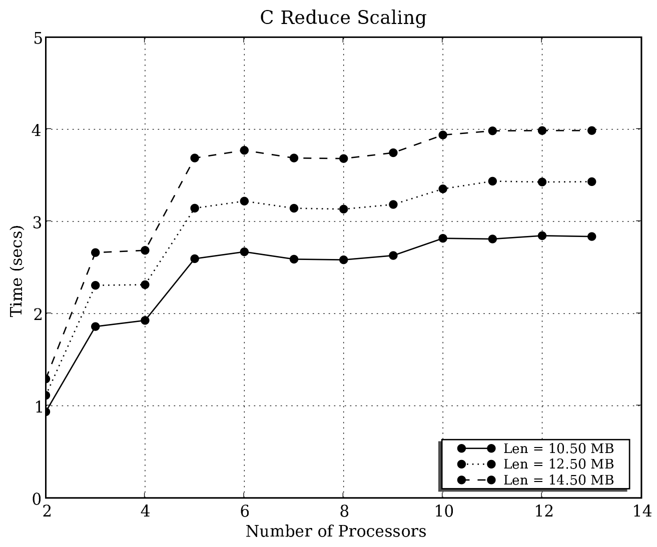

Figure 6.19: Scaling of MPI_Reduce() in simple C implementation.

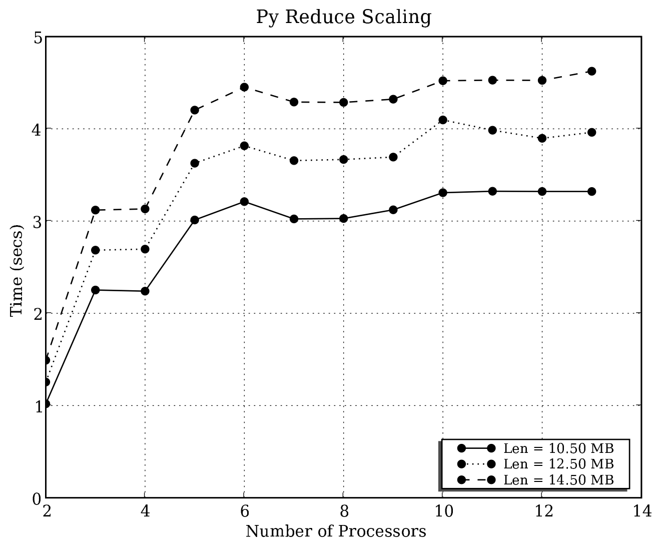

Figure 6.20: Scaling of MPI_Reduce() in Python implementation.

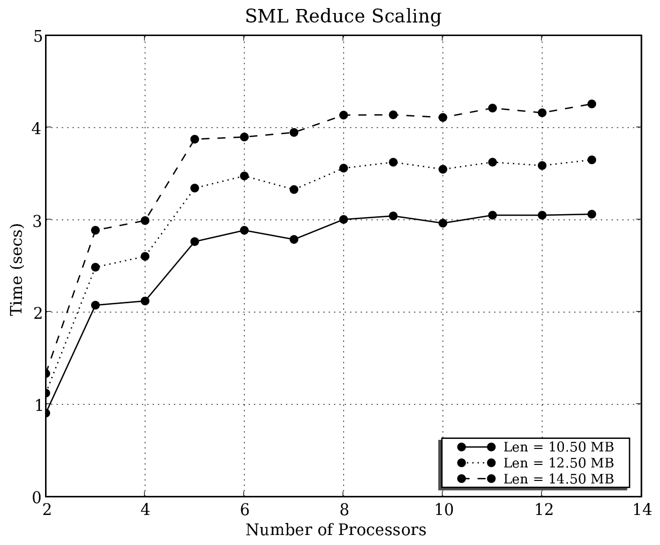

Figure 6.21: Scaling of MPI_Reduce() in SML implementation.

We have defined these operations internally in C in the SML runtime library for speed. The user can also provide an SML function that can be used to perform a custom reduce operation. Figure 6.22 illustrates the performance of MPI_Reduce when the reduce operation is performed in C and in SML (both in the SML MPI implementation) when 13 processors are used. Figure 6.23 illustrates the corresponding relative difference i.e. it shows by what percentage the C version is better than the SML version. The performance difference of about 7.5% is quite acceptable in most applications.

6.5 Numerical Integration Application

We also tested our message passing libraries with a real world application. For this we chose to implement the parallel algorithm to calculate the value of π which uses

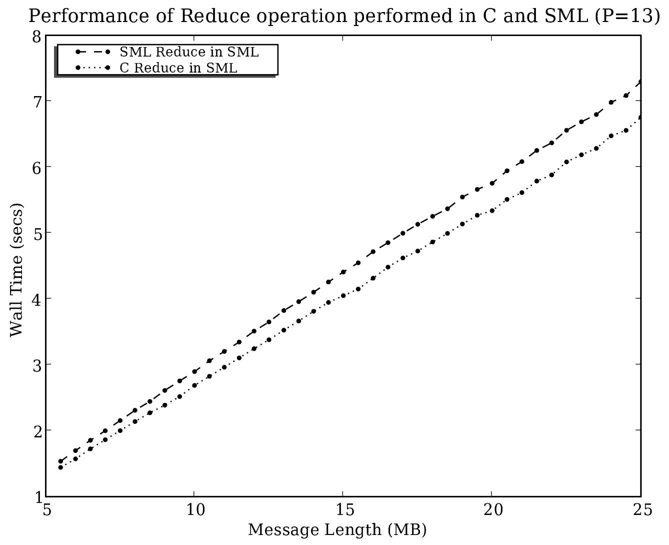

Figure 6.22: Performance of SML's MPI_Reduce(), when reduce operation is performed in C and SML.

the Broadcast and the Reduce MPI operations. For the MPICH implementation we used the example program *cpi.c* that is provided with the MPICH distribution and rewrote the same algorithm using our C, SML and Python MPI libraries. This program determines the value of π by evaluating the following definite integral

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi$$

We tested our application using 8 homogeneous nodes from the same cluster. Figure 6.24 illustrate the execution time for our SML and C implementation along with MPICH. In the case of SML, C and MPICH, we used 8×10^8 samples for the numerical integration and observed that the performance of SML is between 1 and 2 times slower than C. Python being an interpreted language is many orders of magnitude slower than C and SML which are compiled languages. Our python

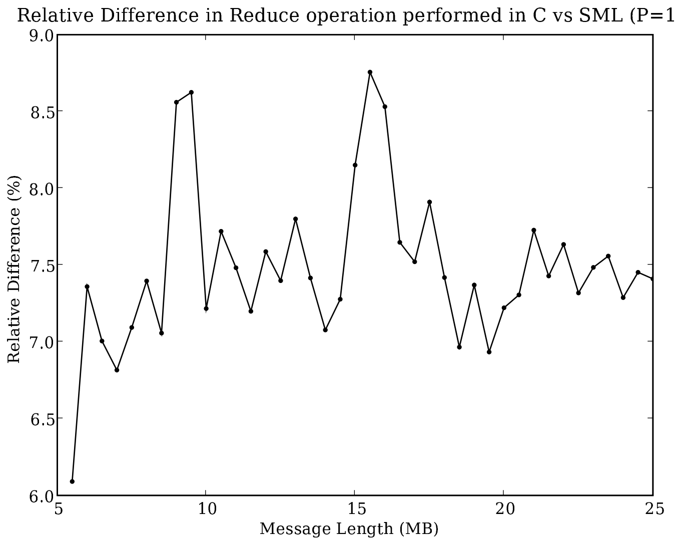

Figure 6.23: Relative Difference in Performance of SML's MPI_Reduce(), when reduce operation is performed in C vs. SML.

implementation takes 526 seconds when 8×10^8 samples are used with two processors. This is because the loop that evaluates the integral is evaluated by the python interpreter at every iteration causing it to be much slower than the compiled languages that we used. So for Python we reduced the number of samples to 8×10^6 for the numerical integration and plotted its performance separately in 6.26.

We also plotted the parallel efficiency (η) of our implementations using the formula

$$\eta = \frac{T_1}{pT_p}$$

where, T_1 is the time it takes to execute the application on a single processor and T_p is the time it takes to execute the same application with the same problem size on p processors. Figures 6.25 and 6.27 illustrate the parallel efficiencies for our implementations. The efficiencies drop off from 100% due to the additional communication overhead involved in using more than one processor. The efficiency of our SML implementation is very similar to that of our C implementation.

This application illustrates that SML strikes a better balance between ease of use and performance than Python.

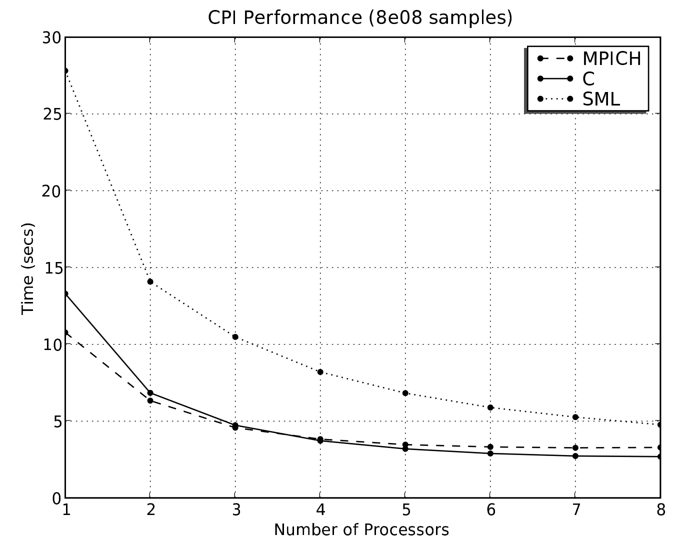

Figure 6.24: CPI performance of C, MPICH and SML.

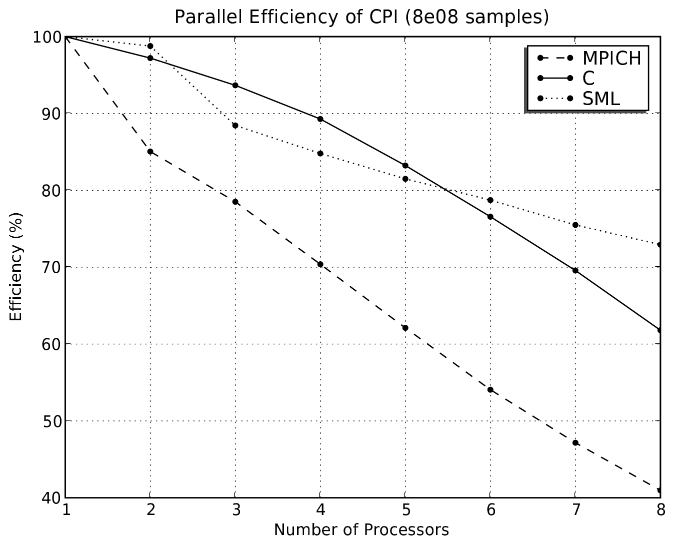

Figure 6.25: Efficiency of CPI in C, MPICH and SML.

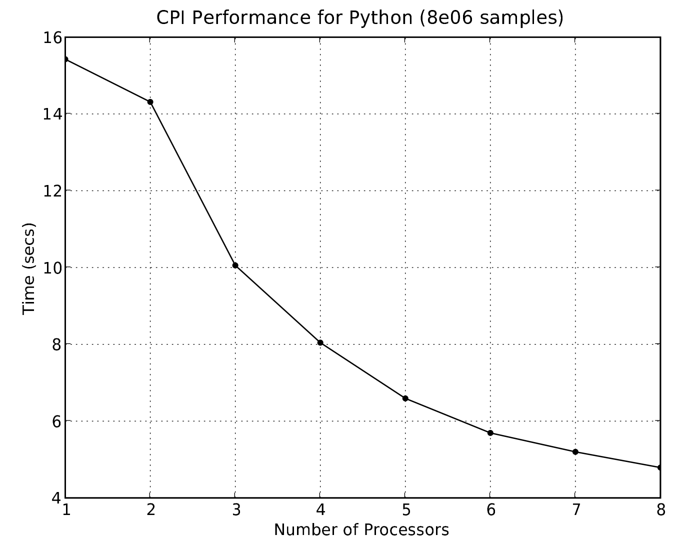

Figure 6.26: CPI performance of Python.

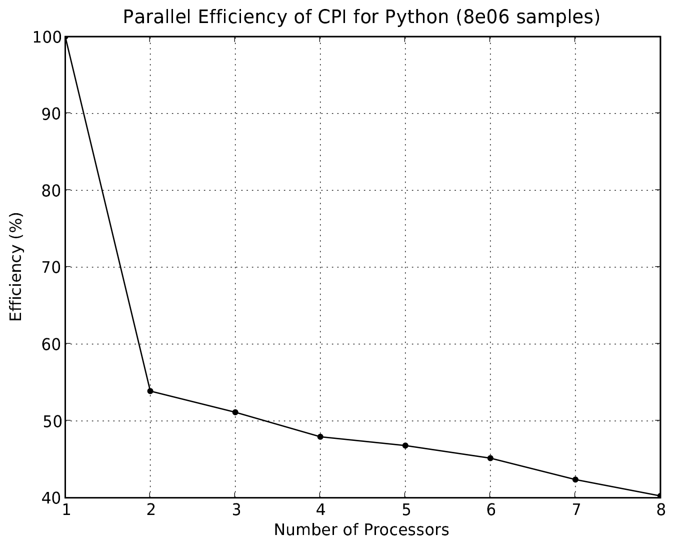

Figure 6.27: Efficiency of CPI in Python.

Chapter 7

Conclusion and Future Work

This chapter summarizes the work presented in this thesis and explores the possibilities of expanding the realm of SMPI.

7.1 Conclusion

This thesis has provided a native implementation of a message passing interface in an advanced programming language Standard ML. The main contribution of the thesis is the design of the SMPI implementation and its realization in SML. The structured implementation is based on the four layered architecture of SMPI. This implementation encourages programmers to do parallel programming in functional languages. We also implemented the same MPI primitives in Python and C in order to compare the performance of our SML implementation. We chose not to use existing Python MPI implementations since they are wrappers around MPICH. We have also compared our implementation with MPICH.

For small message sizes, our SML implementation performs much better than MPICH since in order to be highly portable, MPICH has a higher overhead. For most of our experiments, the performance of our SML implementation is better than that of the Python implementation and closer to that of the C implementation.

We have demonstrated how SMPI allows a programmer to use parallel programming constructs in a functional programming language. This allows the application developer to use higher order functions, automatic storage management, strong typing and exception handling mechanism provided by SML to write well structured, concise and robust code.

7.2 Future Work

This thesis provides only the basic communication primitives for MPI in SML and currently supports string and real datatypes. This library can be extended to include all the function defined in the MPI standard and other data types. The collective operations are implemented using the tree based algorithms. Other algorithms can be implemented to further optimize the performance. This work, along with MPI implementations in other functional languages, will form the basis for the defining language bindings for functional languages in the MPI standard.

Appendix A

SML MPI Library API

This appendix provides a reference for the SMPI library. This can be included as an extension to the Standard ML Basis Library. SMPI provides a set of structures that can be included in SML programs to incorporate MPI features into SML. The structures defined in this library are listed below:

structure MPI

> This module contains all the basic primitives including primitives for point-to-point communication and collective operations required for a Message Passing Interface.

structure SOCK

> This module defines the underlying reliable socket interface which is used to communicate messages.

structure MYTIMER

> This module contains calls to calculate elapsed time using system calls to the C interface.

A.1 SMPI Reference

The MPI structure

The MPI structure is a collection of library functions which is essential for any message passing program. It contains all the essential functions required to write any MPI program. Functions included in this structure are the functions to to create and destroy the communicator handle MPI_COMM_WORLD which is basically a tuple containing rank, the host name list, and an array of socket handles. It also defines the data type MPI.REAL64 which is tuple of functions used to manipulate Real64Array an Real64ArraySlice.

Interface

```
type comm ={nodes:string list, rank:int,
     sockets:SOCK.clientSocket option ref array}
val Rank :  comm → int
val Size :  comm → int
val Init :  string list → comm
val Send :  string * int * comm → unit
val Recv:  int * int * comm → string
val SendArr :  'a * int * comm * dt → unit
val RecvArr :  'a * int * comm * dt → 'b
val Barrier :  comm → unit
val Bcast :  string * int * comm → string
val BcastArr :  'a * comm * dt → 'b
val ReduceArr :  'a * ('b * 'b -> 'c) * comm * dt → 'b
val Finalize :  comm → unit
```

Description

type `comm ={nodes:string list, rank:int,`

`sockets:SOCK.clientSocket option ref array}`

This is an user defined datatype which acts as the communication handler. It consists of the host name list, the rank and an array of socket handles.

val `Rank :  comm → int`

This returns the rank of the host.

val `Size :  comm → int`

This returns the number of processors participating in the computation i.e. the number of hosts stored in the handler comm.

val `Init :  string list → comm`

This function is called by all MPI programs. This sets up the MPI_COMM_WORLD and makes all the server and client connections. There must exist a corresponding Finalise function for every Init function.

val `Send :  string * int * comm → unit`

Sends the message (string) to the destination specified.

val `Recv:  int * int * comm → string`

Receives a message and returns it as a string.

val `SendArr :  'a * int * comm * dt → unit`

Sends the array to the destination specified. dt represents the data type of the elements in the array e.g MPI.REAL64

val `RecvArr : 'a * int * comm * dt` $\rightarrow$ `'b`

Receives an array and populates the array passed to it.

val `Barrier : comm` $\rightarrow$ `unit`

This function ensures that all the processors have executed their program upto the point where Barrier is called.

val `Bcast : string * int * comm` $\rightarrow$ `string`

This function is used to broadcast a message to all the processors.

val `BcastArr : 'a * comm * dt` $\rightarrow$ `'b`

Broadcasts the array of datatype dt to all the processors.

val `ReduceArr : 'a * ('b * 'b -> 'c) * comm * dt` $\rightarrow$ `'b`

Reduces arrays from all the processors by applying the function passed into it as the second parameter. The fully reduced array is obtained at the root processor. dt is the data type of the elements in the array e.g. MPI.REAL64

val `Finalize : comm` $\rightarrow$ `unit`

This function is called at the end of each MPI program. The communicator handle is given as input. It closes all the open sockets in the handle.

The Sock structure

The Sock structure provides a collection of utility functions for creating and closing sockets. It also provides functions for reading and writing into sockets. This is essential part of MPI as all communication between processes takes place with this socket interface. INetSock i.e. Internet domain sockets are used for this purpose.

Interface

val `PORT : int`

type `clientSocket =(INetSock.inet,Socket.active Socket.stream) Socket.sock`

val `createClientSocket : string → clientSocket`

val `createServerSocket : unit → Socket.passive INetSock.stream_sock`

val `acceptServerSocket : (INetSock.inet,Socket.passive Socket.stream) Socket.sock → clientSocket * string`

val `close : ('a,'b Socket.stream) Socket.sock → unit`

val `send : clientSocket * Word8Vector.vector → unit`

val `receive : ('a,Socket.active Socket.stream) Socket.sock * int → Word8Vector.vector`

val `sendArr : clientSocket * 'a * ('a -> ''b) * (clientSocket * 'a → ''b) → unit`

val `recvArr : 'a * 'b * ('a * 'b → 'c) → 'c`

Description

type `clientSocket =(INetSock.inet,Socket.active Socket.stream) Socket.sock`

data type to define client sockets

val `createClientSocket : string → clientSocket`

reads the network host database and gets the internet address which is then converted to socket address (in the INet address family). Creates a socket of the type clientSocket and connects it to the previously acquired address and returns the clientSocket entry.

val `createServerSocket : unit → Socket.passive INetSock.stream_sock`

creates a stream socket in the INet address family in passive mode with the default protocol. It binds the socket to a socket address and creates a queue (of size n) for pending questions associated to the socket. Connections (via connect) to the socket are queued, and later accepted by a call to accept. Raises SysErr if there are too many sockets in use.

val `acceptServerSocket : (INetSock.inet,Socket.passive Socket.stream) Socket.sock → clientSocket * string`

calls Socket.accept and extracts the first connection from the queue of pending connections of the socket, which must be a passive stream socket bound to an address via bind and listening to connections after a call to listen. If a connection is present, Socket.accept returns a pair *(s,sa)* with *s* a new active socket with the properties of the socket passed to the method and *sa* the corresponding socket address. If no pending connections are present on the queue and the socket is not marked as non-blocking, accept blocks until a connection is requested; if the socket is marked as non-blocking, a SysError exception is raised. Returns *s* and the host name of the socket address returned.

val `close : ('a,'b Socket.stream) Socket.sock` $\rightarrow$ `unit`

closes the connection to the socket.

val `send : clientSocket * Word8Vector.vector` $\rightarrow$ `unit`

send calls Socket.sendVec and blocks until all the bytes in the vector are sent across.

val `receive : ('a,Socket.active Socket.stream) Socket.sock * int` $\rightarrow$ `Word8Vector.vector`

receives the number of bytes specified on the socket using the call Socket.recvVec and blocks until all the bytes are successfully received. We modified the SML runtime to use the MSG_WAITALL flag for all receive calls. This flag ensures that an entire message is received with a single call to the receive() socket function.

val `sendArr : clientSocket * 'a * ('a -> ''b) * (clientSocket * 'a` $\rightarrow$ `''b)` $\rightarrow$ `unit`

sends an array over the socket handle passed to it. Second parameter is the ArraySlice that needs to be sent across. Third parameter is a function that returns the length of the array slice. Fourth parameter is the function thats supports sending an array slice of the required data type. It has been implemented this way so that it can be extended to support multiple data types.

val `recvArr : 'a * 'b * ('a * 'b` $\rightarrow$ `'c)` $\rightarrow$ `'c`

receives an array and populates the array slice supplied to it. The first parameter is the server socket, second parameter is the array slice to be populated and third parameter is the function that receives the required data type and populates the array slice.

The MyTimer structure

The MyTimer structure is used to calculate elapsed time.

Interface

val `getTimeOfDay : unit → Int32.int * int`

val `start_timer : unit → real`

val `elapsed_timer : real → real`

Description

val `getTimeOfDay : unit → Int32.int * int`

This function uses Unsafe C Interface to call getTimeofDay provided by C standard library.

val `start_timer : unit → real`

Starts the timer.

val `elapsed_timer : real → real`

Calculates the elapsed time from the time the timer was started.

A.2 Starting an SMPI job

The list of available hosts is assumed to be in a text file whose location is specified in our $mpirun$ script. If not specified, the script defaults to using the machines file used by MPICH usually located at $/usr/local/mpich/share/machines.LINUX$. $mpirun$ has to be provided with the number of hosts to use and the program to run as arguments. The script then connects to the required number of hosts via SSH and starts an instance of the specified program on each of them.

The command used to run a SMPI program is :

./mpirun -np <num hosts> sml @SMLload=exec.x86-linux <arg 1> <arg 2> ...

where

> *exec.x86-linux* is the executable heap image that is created by compiling the application program
>
> *num hosts* is the number of hosts that will participate in the computation
>
> *arg 1, arg 2, ...* are arguments to the application program

When $mpirun$ starts up the application program on all the required hosts, it also appends the list of all participating hosts, port and rank for each host to the command line arguments sent to the application program. These additional arguments are used by the MPI_Init() call.

A.3 Examples

Listing A.1 illustrates how to use the MPI primitives described above.

Listing A.1: Demo using our SMPI Library

```
structure DEMO = struct

fun addRealArray(sl, out, 0) = (Real64ArraySlice.update(out, 0, Real64ArraySlice.sub(sl, 0)
                                        + Real64ArraySlice.sub(out, 0)))
| addRealArray(sl, out, i) = (Real64ArraySlice.update(out, i, Real64ArraySlice.sub(sl, i)
                                        + Real64ArraySlice.sub(out, i)); addRealArray(sl, out, i-1))

fun op2(x,y) = addRealArray(x, y, Real64ArraySlice.length(x)-1)

fun main(pgmName, argv) =
let
        (* Initialize MPI *)
        val COMM = MPI.Init(argv)

        (* Determine my rank and size *)
        val rank = MPI.Rank(COMM)
        val size = MPI.Size(COMM)

        (* Number of elemets in the array *)
        val n = 3

        (* Create array to be used *)
        val buf = Real64Array.array(n, 1.0)
        val recvBuf = Real64ArraySlice.full(Real64Array.array(n, 0.0))

        (*Sample Send and Receive *)
        val _ = if (rank = 0) then
                                MPI.SendArr(Real64ArraySlice.full(buf), 1, COMM, MPI.REAL64)
                              else ()
        val _ = if (rank = 1) then
                                ignore(MPI.RecvArr(recvBuf, 0, COMM, MPI.REAL64))
                              else ()

        (* Sample Broadcast *)
        val a = MPI.BcastArr(buf, COMM, MPI.REAL64)

        (* Sample Reduce with predefined operator *)
        val b = MPI.ReduceArr(buf, MPI.ADD_REAL64, COMM, MPI.REAL64)

        (* Sample Reduce with user defined operator *)
        val c = MPI.ReduceArr(buf, op2, COMM, MPI.REAL64)

        (* Sample Barrier call *)
        val _ = MPI.Barrier(COMM)
in
        MPI.Finalize(COMM);
        OS.Process.success
end
```

Appendix B

Patching and Installing SML/NJ

In order to use our SML/NJ MPI library, it is necessary to apply our *mpi-smlnj.patch* patch. Our patch has been tested with SML/NJ versions 110.59 to 110.65. The patch modifies the SML/NJ runtime as well as the Basis library's implementation. The SML/NJ compiler is also written in SML/NJ. Hence, in order to install the patched Basis library, one has to first install an unpatched SML/NJ compiler and bootstrap the new compiler. The required steps to install version 110.59 on an x86 Linux system are detailed below.

The source code, examples and patch can be downloaded at http://mindspawn.unl.edu/vaishali/thesis-v0.3.tar.bz2.

1. Download the *config.tgz* file from the SML/NJ website into a suitably named folder.

```
$ mkdir smlnj
$ cd smlnj
$ wget http://smlnj.cs.uchicago.edu/dist/working/\
  110.59/config.tgz
$ tar -zxf config.tgz
```

2. Edit *config/targets* and uncomment the line *request src-smlnj*. This instructs

the build process that all the source files are to be downloaded to the local system.

3. Build and install the SML compiler and runtime system.

```
$ config/install.sh
```

4. Now, the *bin* and *lib* folders in the current directory contain the SML/NJ compiler and runtime. The source code to all the other packages is installed in the *src* folder.

5. Before bootstraping the new compiler, in case you already have an SML/NJ installation, unset the environment variable *SMLNJ_HOME*.

```
$ unset SMLNJ_HOME
```

6. Download our *mpi-sml.patch* patch to the current directory. The patch can be applied as follows:

```
$ patch -p0 < mpi-sml.patch
```

7. Build the new compiler and Basis library. The $ corresponds to the shell prompt and the - corresponds to the SML/NJ interpreter's prompt.

```
$ cd src/system
$ ../../bin/sml '$smlnj/cmb.cm'
Standard ML of New Jersey v110.59
[built: Sun Aug 26 00:16:25 2007]
[library $smlnj/cmb.cm is stable]
- CMB.make();
- <Ctrl>+D
$ ./makeml
$ rm -rf ../../lib
$ ./installml
$ cd ../..
```

8. Build the runtime system.

```
$ cd src/runtime/objs
$ make clean
$ make -f mk.x86-linux
$ cp run.x86-linux run.x86-linux.so run.x86-linux.a \
      ../../../bin/.run/
$ cd ../../..
```

9. The *bin* and *lib* folder in the *smlnj* folder now contain the newly built SML/NJ compiler and runtime system. If desired, these two folders can be copied to a different location and the SMLNJ_HOME environment variable can be made to point to the new location.

When patching and building newer versions of SML/NJ such as 110.65, the build process does not create the *src* folder described above. The source code is directly downloaded to the current directory. In this case, the *src* folder can be manually created and the *cm.tgz*, *compiler.tgz*, *system.tgz* and *runtime.tgz* can be copied into it from the *smlnj* directory. If the build process fails complaining about not being able to download *lexgen*, the file *config/allsources* can be edited and the line containing *lexgen* should be commented. Other than these changes, the build process is identical to that of SML/NJ 110.59.

Appendix C

C/Python MPI Library API and examples

C.1 Python MPI API

Currently, all socket operations are blocking.

- MPI_Init(argv)

  Initialize the MPI environment. The command line arguments are passed into this functions so that participating nodes can be determined. The return value is a handle to the current communication group.

- MPI_Finalize(comm)

  Close open sockets between the participating nodes in communication group *comm*. Nothing is returned from this call.

- MPI_Send(buf, dest, comm)

  Send string *buf* to process with rank *dest* in the communication group *comm*.

If other datatype such as arrays are used, their *tostring()* method can be used to convert the array into a string representation. Similarly, any object can be serialized into a string with Python's pickling modules. The lengh of the buffer can be automatically determined and does not have to be specified. Nothing is returned from this call.

- MPI_Recv(len, src, comm)

  Receive a string of length *len* from process with rank *src* in the communication group *comm*. The received string is returned. If expecting an object such as an array, the *fromstring()* method can be used to create an array from the received string.

- MPI_Barrier(comm)

  Ensure that all processes in communication group *comm* have reached this call before proceeeding.

- MPI_Bcast(buf, len, comm)

  Broadcast the string *buf* from the root process (rank=0) to all the other processes in communication group *comm*. The received string is returned to all processes (although it is not required on the root process). The non-root processes should pass an empty string for the argument *buf*, since it is meaningful only for the root process to pass in data that is to be broadcast.

- MPI_Reduce(buf, len, fn, comm)

  Perform the operation *fn* on a string *buf* contained in all processes in communication group *comm*. The operation is performed in a bottom up fashion from the leaves to the root. The non-root processes receive only partial results and only the root process contains the fully reduced buffer. *fn* is a function object that is to be of form *fn(x, y)*, where x and y are two strings that are passed in. The function object *fn* is expected to perform an arbitrary operation on the strings and return a new string. As described in the above operations, the *tostring()*, *fromstring()* and pickling utilities can be used when arrays or other arbitrary objects are used.

C.1.1 Examples

Listing C.1 illustrates how to use the MPI primitives described above.

Listing C.1: Demo using our Python MPI Library

```python
#!/bin/env python

import mpi
import sys
import numpy

def fn(x, y):
        """ User defined function to add two arrays """
        # Convert bytes to a double precission array
        xr = numpy.fromstring(x, numpy.float64)
        yr = numpy.fromstring(y, numpy.float64)

        # Add the two arrays
        zr = numpy.add(xr, yr)

        # Convert array to bytes and return it
        return zr.tostring()


def main():
        # Initialize MPI
        MPI_COMM_WORLD = mpi.MPI_Init(sys.argv)

        rank = mpi.MPI_Comm_rank(MPI_COMM_WORLD)
        size = mpi.MPI_Comm_size(MPI_COMM_WORLD)

        NumReals = 5

        # Construct an array
        X = numpy.ones(NumReals, numpy.float64)
        Xstr = X.tostring()

        # Sample Barrier Call
        mpi.MPI_Barrier(MPI_COMM_WORLD)

        # Sample Send and Receive
        if rank == 0 and size > 1:
                mpi.MPI_Send(Xstr, 1, MPI_COMM_WORLD)
        if rank == 1:
                recvbuf = mpi.MPI_Recv(len(Xstr), 0, MPI_COMM_WORLD)

        # Sample Broadcast
        if rank == 0:
                mpi.MPI_Bcast(Xstr, len(Xstr), MPI_COMM_WORLD)
        else:
                recvbuf = mpi.MPI_Bcast('', len(Xstr), MPI_COMM_WORLD)

        # Sample Reduce
        Ystr = mpi.MPI_Reduce(Xstr, len(Xstr), fn, MPI_COMM_WORLD)

        # Only root process received the fully reduced array
        if rank is 0:
                Y = numpy.fromstring(Ystr, numpy.float64)
                # Print the reduced array
                print Y


        # Cleanup MPI environment
        mpi.MPI_Finalize(MPI_COMM_WORLD)


if __name__ == '__main__':
        main()
```

C.2 C MPI API

Our C MPI library defines the following datatypes for use with the functions defined below MPI_INT, MPI_FLOAT, MPI_DOUBLE and MPI_CHAR.

- COMM* MPI_Init(int argc, char **argv)

  Initialize the MPI environment. The command line arguments *argv* and the number of arguments *argc* are passed into this function so that participating nodes can be determined. The return value is a handle to the current communication group.

- void MPI_Finalize(COMM *comm)

  Close open sockets between the participating nodes in communication group *comm*. Nothing is returned from this call.

- int MPI_Send(void *buf, int count, int dest, COMM *comm, int datatype)

  Send *count* elements of array *buf* of type *datatype* to process with rank *dest* in the communication group *comm*. On successful completion, a positive integer is returned.

- int MPI_Recv(void *buf, int count, int src, COMM *comm, int datatype)

  Receive an array of length *count* elements of type *datatype* from process with rank *src* in the communication group *comm* into buffer *buf*. On successful completion, a positive integer is returned.

- int MPI_Barrier(COMM *comm)

  Ensure that all processes in communication group *comm* have reached this call before proceeeding.

- int MPI_Bcast(void *buf, int count, COMM *commm, int datatype)

  Broadcast the array *buf* with *count* elements of type *datatype* from the root process (rank=0) to all the other processes in communication group *comm*. The received array is returned to all non-root processes (in *buf*). The non-root processes should pass the address of an allocated block of memory for the argument *buf*, since it is meaningful only for the root process to pass in data that is to be broadcast.

- int MPI_Reduce(void *sbuf, void *rbuf, int count, void (*op)(void *, void *, int, int), COMM *comm, int datatype)

  Perform the operation *op* on an array *buf* with *count* elements of type *datatype* contained in all processes in communication group *comm*. The operation is performed in a bottom up fashion from the leaves to the root. The non-root processes receive only partial results and only the root process contains the fully reduced buffer. *op* is a pointer to a function of form *fn(void *in, void *inout, int count, int datatype)*, where *in* and *inout* are two arrays that are passed in and the newly computer array is placed in *inout*. The type and number of elements in the two arrays are specified by arguments *datatype* and *count* respectively.

C.2.1 Examples

Listing C.2 illustrates how to use the MPI primitives described above.

Listing C.2: Demo using our C MPI Library

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

void Add(void *in, void *inout, int count, int datatype) {
        double *din=(double *)in, *dinout=(double *)inout;
        int i;

        // Add the two arrays and store results in inout[]
        for(i=0; i<count; i++)
                dinout[i] += din[i];
}

int main(int argc, char **argv) {
        // Initialize MPI
        COMM *MPI_COMM_WORLD = MPI_Init(argc, argv);

        // Determine my rank and size
        int rank = MPI_Comm_rank(MPI_COMM_WORLD);
        int size = MPI_Comm_size(MPI_COMM_WORLD);

        // Sample Barrier Call
        MPI_Barrier(MPI_COMM_WORLD);

        int NumElements = 3;
        // Create arrays that we will use
        double X[] = {1.0, 1.0, 1.0};
        double Y[NumElements];

        // Sample Send and Receive
        if (rank == 0 && size > 1)
                MPI_Send(X, 3, 1, MPI_COMM_WORLD, MPI_DOUBLE);
        if (rank == 1)
                MPI_Recv(X, 3, 0, MPI_COMM_WORLD, MPI_DOUBLE);

        // Sample Broadcast
        MPI_Bcast(X, 3, MPI_COMM_WORLD, MPI_DOUBLE);

        // Perform a sample reduce operation
        MPI_Reduce(X, Y, NumElements, &Add, MPI_COMM_WORLD, MPI_DOUBLE);

        // Make the root print out the reduced array
        if (rank == 0 )
                printf("%g %g %g\n", Y[0], Y[1], Y[2]);

        MPI_Finalize(MPI_COMM_WORLD);

        return 1;
}
```

Bibliography

[1] The caml language. Accessed from http://caml.inria.fr/.

[2] F #. Accessed from http://research.microsoft.com/fsharp/fsharp.aspx.

[3] Lam/mpi parallel computing. Accessed from http://www.lam-mpi.org/.

[4] The message passing interface (mpi) standard. Accessed from http://www-unix.mcs.anl.gov/mpi/standard.html.

[5] Ml language. Accessed from http://www.hprog.org/fhp/MlLanguage.

[6] Mpi-2: Extensions to the message-passing interface. Accessed from http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[7] Mpich-a portable implementation of mpi. Accessed from http://www-unix.mcs.anl.gov/mpi/mpich/.

[8] The scampi library. Accessed from http://www-lasmea.univ-bpclermont.fr/Personnel/Jocelyn.Serot/scampi.html.

[9] R. Alasdair, A. Bruce, J. Mills, and A. G. Smith. Chimp/mpi user guide. Technical report, Edinburgh Parallel Computing Centre, 1994.

[10] O. Babaoglu, L. Alvisi, A. Amoroso, R. Davoli, and L. A. Giachini. Paralex: an environment for parallel programming in distributed systems. In *6th ACM International Conference on Supercomputing*, pages 178–187, Washington, D.C., 1992.

[11] J. Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[12] E. Biagioni. A structured TCP in standard ML. In *SIGCOMM*, pages 36–45, 1994. citeseer.nj.nec.com/biagioni94structured.html.

[13] F. Cheng, P. Vaughan, D. Reese, and A. Skjellum. *The Unify System*. Engineering Research Center, Mississippi State University, 1 1994.

[14] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concept and Design*. Addison-Wesley, 3rd edition, 2000.

[15] D.Kafura and L.Huang. mpi++: A c++ language binding for mpi. Accessed from http://www.osl.iu.edu/download/mpidc95/papers/html/huang/.

[16] N. Doss, W. Gropp, E. Lusk, and A. Skjellum. A model implementation of mpi. Technical report, Argonne National Laboratory, 1993.

[17] I. Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1995.

[18] Gerasoulis and T. Yang. Scheduling program task graphs on MIMD architectures. In R. Paige, J. Reif, and R. Wachter, editors, *Algorithm Derivation and Program Transformation*. Kluwer, To appear.

[19] W. Gropp and E. Lusk. The MPI communication library: its design and a portable implementation. In *Proceedings of the Scalable Parallel Libraries Conference, October 6–8, 1993, Mississippi State, Mississippi*, pages 160–165, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994. IEEE Computer Society Press.

[20] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[21] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1994.

[22] W. D. Gropp and E. Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[23] K. Hammond, J. Peterson, et al. *Report on the Programming Language Haskell: A Non-strict, Purely Functional Language*. Yale University, New Haven, Connecticut, USA, 1997. Version 1.4.

[24] J. Harrison. Introduction to functional programming. URL = http://www.cl.cam.ac.uk/teaching/Lectures/funprog-jrh-1996/, 1997.

[25] A. Hey. The mpi standard: A progress report.

[26] U. Kumar, V. Rajasekaran, M. Chetlur, G. D. Sharma, R. Radhakrishnan, and P. A. Wilsey. Addressing communication latency issues on clusters for fine grained asynchronous applications - a case study, 1999.

[27] U. Kumar, V. Rajasekaran, R. Radhakrishnan, and P. A. Wilsey. Tcpmpl a tcp/ip based message passing library for warped. URL = http://www.ece.uc.edu/ paw/warped/tcpmpl/.

[28] B. C. McCandless, J. M. Squyres, and A. Lumsdaine. Object-oriented mpi (oompi): A class library for the message passing interface. In *MPIDC '96: Proceedings of the Second MPI Developers Conference*, page 87, Washington, DC, USA, 1996. IEEE Computer Society.

[29] B. C. McCandless, J. M. Squyres, and A. Lumsdaine. Object oriented MPI (OOMPI): a class library for the Message Passing Interface. In IEEE, editor, *Proceedings. Second MPI Developer's Conference: Notre Dame, IN, USA, 1–2 July 1996*, pages 87–94, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.

[30] P. Miller. pympi: An introduction to parallel python. URL = http://pympi.sourceforge.net, 2006.

[31] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[32] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[33] S. Mintchev. Functional programming helps speed up mpi collective operation. citeseer.nj.nec.com/mintchev97functional.html.

[34] O. Nielsen. Pypar - building a parallel program step-by-step. URL = http://datamining.anu.edu.au/ ole/pypar, 2007.

[35] P. S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[36] L. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1992.

[37] R. Pucella. Notes on programming in sml/nj. URL = http://www.cs.cornell.edu/riccardo/prog-smlnj/notes-011001.pdf, 2001.

[38] J. H. Reppy. *Concurrent Programming in ML*. Cambridge Univ Pr (Trd), December 1999.

[39] S. Sankaran, J. M. Squyres, B. Barrett, and A. Lumsdaine. Checkpoint-restart support system services interface (SSI) modules for LAM/MPI. Technical Report TR578, Indiana University, Computer Science Department, 2003.

[40] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proceedings, LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.

[41] M. Snir and S. Otto. *MPI-The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[42] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, November 1995.

[43] J. M. Squyres, B. Barrett, and A. Lumsdaine. MPI collective operations system services interface (SSI) modules for LAM/MPI. Technical Report TR577, Indiana University, Computer Science Department, 2003.

[44] J. M. Squyres, B. Barrett, and A. Lumsdaine. Request progression interface (RPI) system services interface (SSI) modules for LAM/MPI. Technical Report TR579, Indiana University, Computer Science Department, 2003.

[45] J. M. Squyres, B. C. McCandless, and A. Lumsdaine. Object oriented MPI reference. Technical Report TR 96-12, Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, USA, 1996.

[46] Technion-Israel Institute of Technology. *OCamlMPI Tutorial*. Accessed from http://www.cs.technion.ac.il/Labs/dsl/projects/starfish/release/htmldocs/OCamlMPI.html.

[47] D. Turner, S. Selvarajan, X. Chen, and W. Chen. The mp_lite message-passing library. In S. G. Akl and T. F. Gonzalez, editors, *IASTED PDCS*, pages 429–434. IASTED/ACTA Press, 2002.

[48] P. Wadler. How to solve the reuse problem? functional programming. pages 371–372.

[49] P. Wadler. Why no one uses functional languages. *SIGPLAN Notices*, 33(8):23–27, 1998.