Message-Passing Interface for Microsoft Windows 3.1

A Thesis

Presented to the

Department of Computer Science

and the

Faculty of the Graduate College

University of Nebraska

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

University of Nebraska at Omaha

by

Joerg Meyer

December 1994

Thesis Acceptance

Accepted for the faculty of the Graduate College, University of Nebraska, in partial fulfillment of the requirements for the degree Master of Science, University of Nebraska at Omaha.

Committee

_____

Name Department

_____

_____

_____

Chairperson _____

Date _____

Abstract

Parallel computing offers the potential to push the performance of computer systems into new dimensions. Exploiting parallelism, concurrent tasks cooperate in solving huge computational problems. The theoretical foundations of parallel processing are well-established, and numerous types of parallel computers and environments are commercially available. The main obstacle for a broad application of parallel technology is the lack of parallel programming standards. This research is aimed to promote the acceptance of the Message-Passing Interface (MPI) Standard, which provides the means for writing portable software on a wide variety of parallel computers under UNIX.

This thesis outlines the development and implementation of MPI for MS-Windows 3.1, which we call WinMPI. The goal of WinMPI is two-fold: 1) as a development tool, to allow the easy and inexpensive implementation of parallel software, and 2) as a learning tool, to provide a larger group of computer users the opportunity to gain first experience with parallel programming.

We give an introduction to the MPI standard, illustrated by an MPI example program. We discuss design and implementation issues for WinMPI, focusing on the simulation of a UNIX-like run-time environment in Windows. Among others, preemptive multitasking, process start-up, and shared memory communication are addressed.

Special consideration is given to the implementation of WinMPI applications. We describe instructions for porting MPI applications between WinMPI and UNIX. Users with some background in MPI programming will find sufficient information to succeed in employing WinMPI for parallel programming projects.

Acknowledgments

The author wants to take the opportunity to thank people that supported this project. In particular, I am grateful to Professor Hesham El-Rewini, who suggested this project and helped me organize the work. Thanks to Professor Stan Wileman for tips and answers to countless questions concerning Windows programming and for his prompt support regarding hardware and software problems. Professor Hesham Ali and Professor George Pfeffer provided many suggestions for this thesis. Debbie deGraw helped me to step over linguistic traps. Last but not least, I want to thank all the people in the computer science department at University of Nebraska at Omaha for their support that enabled me to finish this project within the given time.

Table of Contents

1. INTRODUCTION 1

2. MPI - A NEW STANDARD FOR PARALLEL COMPUTING 4

2.1. Why Another Standard? 4

2.2. Introduction to MPI 9

2.3. MPI Example Program 16

2.4. Why MPI for MS-Windows 3.1 19
2.4.1. MPI for Windows as Development Tool 21
2.4.2. MPI for Windows as Learning Tool 21

2.5. Outlook: MPI for MS-Windows NT 22

3. IMPLEMENTATION FOR MICROSOFT WINDOWS 3.1 25

3.1. The MS-Windows 3.1 Programming Environment 25
3.1.1. Basic MS-Windows Program Structure 25
3.1.2. Memory Models 27
3.1.3. Non-preemptive Multitasking 29
3.1.4. Window Management 31

3.2. Structure of MPI under MS-Windows 3.1 31

3.3. Mapping of MPI Functionality to MS-Windows 3.1 34
3.3.1. Placement of main() Function 35
3.3.2. The Upper Layer in MPI for Windows 37
3.3.3. MPI_Win_yield() - Message-Loop Function 40
3.3.4. Communication Primitives 41
3.3.5. Slave Processes 42
3.3.6. Synchronization Primitives 44
3.3.7. Input/Output Functions 45

4. RUNNING MPI-PROGRAMS UNDER MICROSOFT WINDOWS 3.1 50

4.1. Porting MPI Programs from UNIX to MS-Windows 3.1 50

4.2. Start of a WinMPI application 53

4.3. Porting MPI Programs from MS-Windows 3.1 to UNIX 57

5. CONCLUDING REMARKS **59**

6. BIBLIOGRAPHY **62**

Table of Figures

Figure 1 Structure of current parallel and distributed systems _____ *6*
Figure 2 MPI Concept _____ *8*
Figure 3 Basic structure of an MPI application _____ *12*
Figure 4 Basic MPICH structure under UNIX _____ *14*
Figure 5 Calculation of π by numerical integration _____ *16*
Figure 6 Distribution of 10 rectangles among 3 processes _____ *17*
Figure 7 MPI example program - Computation of π _____ *18*
Figure 8 Basic C Program Structure for MS-Windows 3.1 _____ *26*
Table 1 Memory models _____ *28*
Figure 9 Basic MPI structure under MS-Windows 3.1 _____ *34*
Figure 10 Embedding of MPI application into MS-Windows program _____ *36*
Figure 11 Processing of Windows messages during MPI/p4 program execution _____ *40*
Figure 12 Start of slave processes in **create_bm_processes()** in UNIX version of p4 _____ *42*
Figure 13 Start of slave processes in **create_bm_processes()** in Windows version of p4 _____ *43*
Figure 14 Basic locking functions in p4 for MS-Windows _____ *45*
Figure 15 Data and stack segments of WinMPI processes and p4-DLL _____ *47*
Figure 16 WinMPI example program - Computation of π _____ *53*
Figure 17 Start of WinMPI application _____ *55*
Figure 18 WinMPI application after execution of **MPI_main()** function _____ *56*

1. Introduction

Since the invention of computers about 50 years ago, the world of computing has changed rapidly. Roughly each decade, a new generation of computer systems sets higher standards with regards to performance, size, price, and usability. Computers continue to conquer new spheres in industry, research, and management and affect virtually every aspect of daily life.

In the 1990's, this race goes into a new round, the Era of Parallel Computers. Ideas that have determined the basic mechanisms of computers for 40 years are being revised and replaced by a principle known to human society since its earliest civilizations. Division of labor enabled human cultures with access to low level technology to succeed in erecting gigantic buildings like the Egyptian pyramids. Parallel computer systems adopted this idea of cooperation by employing multiple processors. Huge computational problems are divided, separately solved, and integrated into a final solution.

The boost of parallel computers is no coincidence. Numerous applications in research and industry demand tremendous computational power. Complex processes such as atmospheric activities, crash test simulations, and fluid dynamics are modeled theoretically by mathematical methods, resulting in complicated systems of equations with hundreds of variables. Only parallel computers provide the computational performance to solve such problems in an acceptable period of time.

Parallel computer systems can employ various fundamental occurrences of parallelism. [Flynn66] suggests a widely used classification with regards to the number of instruction and data streams, resulting in four acronyms: SISD, SIMD, MIMD, and MISD.[1] SISD describes sequential computers while the remaining three exploit different types of parallelism.

---

[1] **S**ingle-**I**nstruction-**S**ingle-**D**ata, **S**ingle-**I**nstruction-**M**ultiple-**D**ata, etc.

After researchers had established the theoretical foundations for parallel computing, many vendors began developing and marketing parallel computers, employing various parallel computing paradigms. The programming environments of sequential machines served as a base; each manufacturer provided additional functionality for writing parallel programs.

Due to the utilization of the latest technology, immense development costs, and lack of competition, parallel computers have been very expensive and available to only a limited number of institutions., Distributed systems of interconnected workstations are an alternative. They are widely available and offer a far better price-performance ratio. As a result, many manufacturers of parallel machines have severe financial problems. There are even serious predictions that highly-specialized parallel machines will disappear in the near future.

Currently, many users have access to parallel computing technology and are willing to use it for their particular needs. Both demand and supply of parallel software are increasing considerably. Software developers and users face the nonexistence of a widely accepted standard as a severe obstacle. Software for a particular parallel computer is expensive to rewrite for another system or is simply not portable. Numerous attempts have been made to establish a standard. Due to a broad foundation of support from vendors and researchers, a standard called Message-Passing Interface (MPI) is likely to be adopted as the base for future parallel application software. The standard is now stable, and first implementations are available for parallel and distributed systems.

The focus of this project is an MPI implementation for Microsoft Windows 3.1, which we call WinMPI. Several reasons exist for the belief that such an implementation for a single-processor operating system like MS-Windows 3.1 has a broad range of applications as a development and learning tool.

An introduction to MPI will be given in Chapter 2. The current situation in the parallel computing field is assessed by comparing it to the situation for sequential computers prior to the appearance of high-level languages, deriving the necessity of MPI. Some historical

background will be provided, and the goals and basic concepts of MPI are discussed. A short example program illustrates the programming approach suggested by MPI. The subsequent sections will discuss the motivation for WinMPI and the feasibility of an MPI implementation on Microsoft Windows NT and Windows 95.

In Chapter 3, the implementation details for MPI for MS-Windows are discussed. After a brief introduction to the Windows programming approach, the changes to the basic MPI structure are presented. Differences between UNIX and Windows are discussed which introduced several hurdles during the porting process. The essential pieces of source code illustrate the solutions to these problem.

In Chapter 4, guidelines for the implementation of MPI programs under MS-Windows are addressed. The main problems are examined, and appropriate solutions are given which even users that are not familiar with Windows programming can master. Chapter 5 generalizes the accomplishments and presents suggestions for further work.

2. MPI - A New Standard for Parallel Computing

2.1. Why Another Standard?

In this section, the rationale behind the introduction of the MPI standard is presented. The current status of parallel computing is compared to the time before the invention of high-level programming languages.

By the mid-1950's, computers had reached a state of development that suggested their successful utilization not only for research but also for applications in industry and administration. A number of manufacturers designed and marketed computer systems. Many computer programs were written for these systems. During this period, assembly language was used for software development, which, from today's point of view, has several drawbacks:

- designed for a particular hardware
- source code difficult to understand
- laborious and time-consuming debugging

As a result, computer programs could be used only for a particular computer system and proved to be very expensive and error-prone. Efficient software development required new concepts.

In the late 1950's, the solution appeared in the form of high-level languages such as Fortran and Algol 60 [Meek78]. Source code became more intuitive and easier to read and to debug. A compiler translated the source code to machine-dependent assembly (or executable) code. Suddenly, it was possible to write reliable software of greater complexity in a shorter time.

Additionally, high-level languages defined a hardware-independent programming interface; programs could run on different computer systems. The standardization of high-level languages allowed for portable software.

Computers became even easier to use when UNIX was adopted as the de-facto operating system standard. Today, no major workstation vendor can afford not to offer UNIX for its computer systems. Once familiar with the UNIX user interface, user are able to operate machines from different manufacturers without additional training.

During the last two decades, parallel computing has evolved into a major field of research in computer science. Manufacturers such as Cray, IBM, INTEL, nCube, Maspar, Sequent, and Silicon Graphics have developed and marketed computer systems equipped with multiple processors. Two major groups of parallel systems can be distinguished[2]:

- **Shared memory machines** - Multiple processors have access to a common main memory (shared memory communication).
- **Distributed memory machines** - Each processor has associated its own local main memory. Processors exchange messages via separate communication channels (message passing).

The UNIX standard proved to be so strong that even vendors of parallel machines felt compelled to offer UNIX-like operating systems for their computers. Parallel machines can be programmed using popular high-level languages such as C, Fortran, and Pascal. However, parallel computers require specific extensions to the UNIX operating system for communication and synchronization. Hence, each vendor added functions to the UNIX programming interface which are used for implementing parallel application software.

These parallel computers are very expensive since the number of problems in research and industry requiring huge computational power is steadily increasing but the number of sold systems is still small. A less expensive alternative is the use of distributed systems which

[2] More details on shared-memory and distributed-memory parallel programming can be found in [Lewis92].

consist of a number of workstations connected by a communication network (workstation cluster). Distributed systems are suitable for parallel applications, although they do not reach the performance of parallel systems.

Figure 1 illustrates the structure of currently available parallel and distributed systems. Most computers offer an UNIX-compatible operating system. However, the communication primitives are very different, resulting in incompatible extensions for parallel programming.

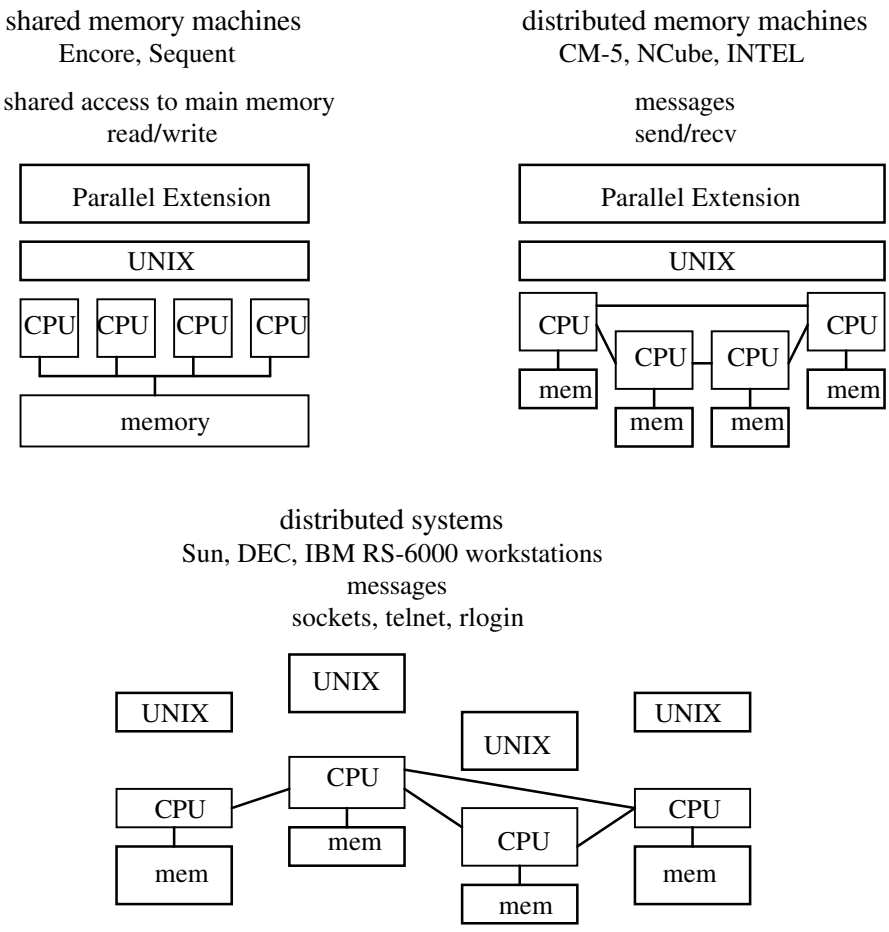


Figure 1 Structure of current parallel and distributed systems

The current state of development for parallel and distributed systems is comparable to the crisis in assembly language programming in the late 1950's. An application program written for a particular parallel computer system is difficult to port to another manufacturer's hardware because the parallel extensions to the UNIX programming environment differ. As a solution to this dilemma, a standard is desirable. This new standard will accomplish for parallel applications what high-level languages permitted for sequential application software - application programs running on different parallel machines without any changes.

Numerous attempts have been made to propose a standard. Intel's NX/2 [Pierce88], PICL [Geist90], Express [Parasoft92], p4 [Butler92], and PVM [Dongarra94b] are only a few examples. Common drawbacks of these programming systems are:

- Designed with regard to the hardware of a particular computer manufacturer.
- Designed as a research project, not suitable for commercial use.

Meanwhile, the availability of a widely-accepted standard for parallel computers is the key to growing acceptance of parallel computers in research and industry. In order to be suitable for all types of parallel computers, a parallel programming standard must be based on the most general communication paradigm, the message-passing paradigm. In the message-passing paradigm, sharing resources and synchronization among processes is achieved by sending messages.

About 60 people from 40 organizations participated in the standardization of the Message-Passing Interface Standard (MPI). Most major manufacturers of parallel computers and researchers from universities, government laboratories, and industry were involved in the development of MPI.

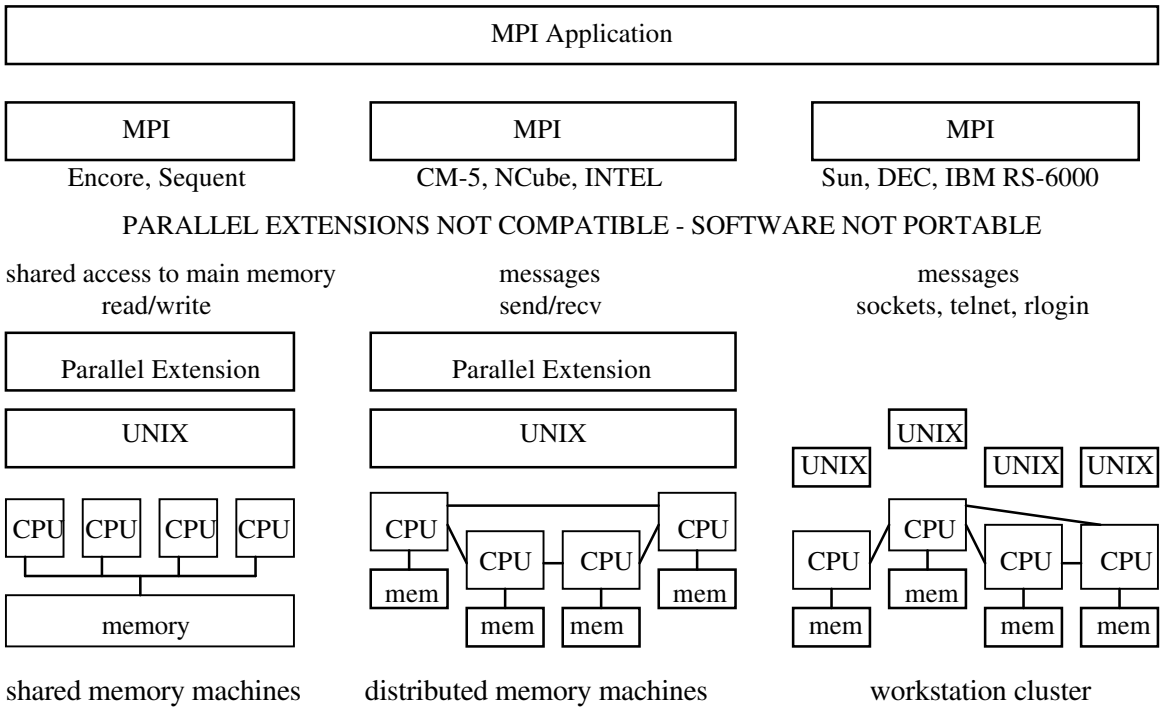


Figure 2 MPI Concept

Derived from Figure 1, Figure 2 illustrates how MPI permits writing parallel application software that is portable to a wide variety of existing concurrent computers. MPI provides a software layer that hides the peculiarities of a particular system and provides a hardware-independent interface to an MPI application.

The MPI standardization process began in April, 1992, with the Workshop on Standards for Message Passing in a Distributed Memory Environment, which was held in Williamsburg, Virginia [Walker92]. At this workshop the essential features for a message-passing interface standard were discussed, and the MPI working group was formed to continue the standardization process.

In November 1992, this working group held a meeting in Minnesota where the standardization process was reformed by adopting the organization and procedures of the

High Performance Fortran Forum.[3] For each major component area, a subcommittee was formed and an email discussion service provided. Both together embodied the MPI Forum, which invited all members of the high performance computing community to participate in the standardization process.

A revised version of the preliminary draft proposal, called MPI1, was available in February 1993 [Dongarra93a]. The intention behind MPI1 was to provide a base for a broad discussion. The draft MPI standard was presented at the Supercomputing '93 conference in November 1993. It is available as a technical report from the University of Tennessee report [MPIF94a], as a postscript file by ftp (from info.mcs.anl.gov in /pub/mpi/mpi-report.Z), as hypertext on the World Wide Web at http://www.mcs.anl.gov/mpi, and as an article in the Journal of Supercomputing Applications [MPIF94b].

Meanwhile, MPI has reached a state where several proprietary, native implementations are in progress and portable public domain implementations are available [Bridges94].

A discussion at the final MPI meeting in February, 1994, resulted in the decision to postpone plans for extending MPI until users had gained some experience with the current MPI standard. Current plans of the MPI Forum are to discuss the possibility of an MPI-2 standard in the near future.

[3] The High-Performance Fortran Forum is a broad coalition of industrial and academic groups with the goal of establishing a standard for High-Performance Fortran (HPF).

2.2. Introduction to MPI

This section gives a brief overview of the goals of MPI, the content of the standard, and the basic programming approach. The basic structure of the MPI implementation that was used for this WinMPI port is also presented.

The basic goal of MPI is stated by the MPI forum as

> "to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message passing." [MPIF94a]

Other major goals are:

- Design an application programming interface.
- Allow efficient communication (avoid memory copying, overlap computation and communication).
- Allow implementations in heterogeneous environments.
- Define an interface not too different from current practice, such as PVM, Express, etc., and provide extensions that allow greater flexibility.
- Define an interface that can be easily implemented on many vendors' platforms.

The MPI standard chose the message-passing paradigm because of its wide portability. Programs run on distributed-memory or shared-memory machines, networks of workstations, and combinations of these. Neither new architectures combining shared- and distributed-memory views nor increases in network speed will make the standard obsolete.

The standard satisfies the needs of fully general MIMD programs, as well as applications written in the more restrictive SPMD[4] style. MPI uses the static process group concept,

[4] **S**ingle-**P**rogram-**M**ultiple-**D**ata - see [Lewis92]

i.e. the number of concurrent tasks is determined at the starting time, and no dynamic spawning of tasks is supported.

The MPI standard includes:

- Point-to-point communication
- Collective operations
- Process groups
- Communication contexts
- Process topologies
- Bindings for Fortran 77 and C
- Environmental management and inquiry
- Profiling interface

An MPI application can be visualized as a collection of concurrent processes as illustrated in Figure 3. An MPI application program includes code written by the application programmer and linked with a functions library provided by the MPI software package. Each process is assigned a unique **rank**, an integer number between 0 and *n-1* for an MPI application consisting of *n* processes. These ranks are used by MPI processes to address other processes to send or receive messages, to execute collective operations, and to cooperate in other forms.

MPI processes can run on the same computer or on different machines concurrently. For an application program, sending a message to a process on the same or another machine is a transparent operation. MPI will automatically select the most efficient communication mechanism available on a particular machine or between machines. The usage of ranks makes all cooperative operations independent of the physical location of the participants.

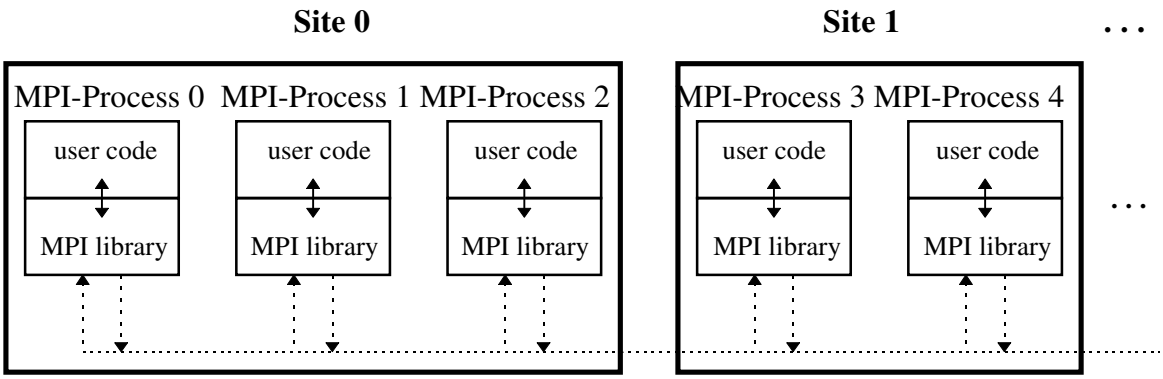


Figure 3 Basic structure of an MPI application

The MPI standard defines no rules for how to assign MPI processes to processors. No interface is provided for the definition of an MPI application's process structure. Procedures for starting an MPI application are also not included in the standard. Since current parallel computers use very different startup procedures, a standardization attempt appeared inappropriate.[5]

Four MPI implementations are currently available:

- MPICH implementation from Argonne National Laboratory/Mississippi State University
- LAM implementation from the Ohio Supercomputing Center
- CHIMP implementation from Edinburgh Parallel Computing Centre
- UNIFY implementation from Mississippi State University (subset of MPI)

References to all four implementations can be found on the World-Wide Web (http://www.mcs.anl.gov/mpi/index.html).

[5] Current MPI implementations do, however, provide a programming environment with tools for configuring and starting MPI applications.

MPI for Windows 3.1 is based on the MPICH implementation from Argonne National Laboratory/Mississippi State University since the author gained his first MPI experiences with this package. Early investigations suggested it was portable to MS-Windows, although some issues could not reach resolution until after the porting process was in progress.

In order to simplify the implementation process, MPICH uses a two-layer approach. All MPI functions comprise the upper layer and access the lower layer through an ADI (Abstract Device Interface), which provides hardware-independent access to communication and synchronization primitives in the lower layer. Depending on the particular environment, the lower layer is the native communication subsystem of the parallel machine or another message-passing system, such as p4, PVM, or Chameleon.

p4 is MPICH's preferred option for the lower layer if the ADI interface does not directly support the particular hardware environment. p4 was developed at Argonne National Laboratory about three years ago for the same purpose as MPI [Butler92]. It provides a message-passing interface for writing parallel applications and supports a wide variety of parallel computer systems, although it offers less functionality than MPI. By using p4 as the communication subsystem, MPI can run on all these machines without containing machine-dependent source code in the upper layer. Through the remainder of this thesis, the terms upper layer and MPI layer as well as the terms lower layer and p4 layer are used interchangeably.

In order to keep the MPI layer hardware-independent, p4 was chosen as the lower layer for WinMPI. The motivation for that decision is discussed in more detail in Section 3.2.

Figure 4 shows the configuration of MPICH under UNIX using p4. The static MPI library linked to each MPI process is divided into the MPI function part (the upper layer) and the p4 layer. p4 divides parallel applications into clusters which combine processes with identical code running on the same machine. p4 clusters are not visible to an MPI application. For communication, p4 uses shared memory for all processes in the same

cluster, special message-passing interfaces for all processes on the same site, and the socket interface for all processes. The socket interface is a de-facto standard for all UNIX machines and provides networking services to computers systems. It is designed for data transmission in global networks and lacks the performance required for high-performance parallel computers. For this reason, those provide an additional high-speed message-passing interface. However, a computer system is not required to provide all three interfaces mentioned above.

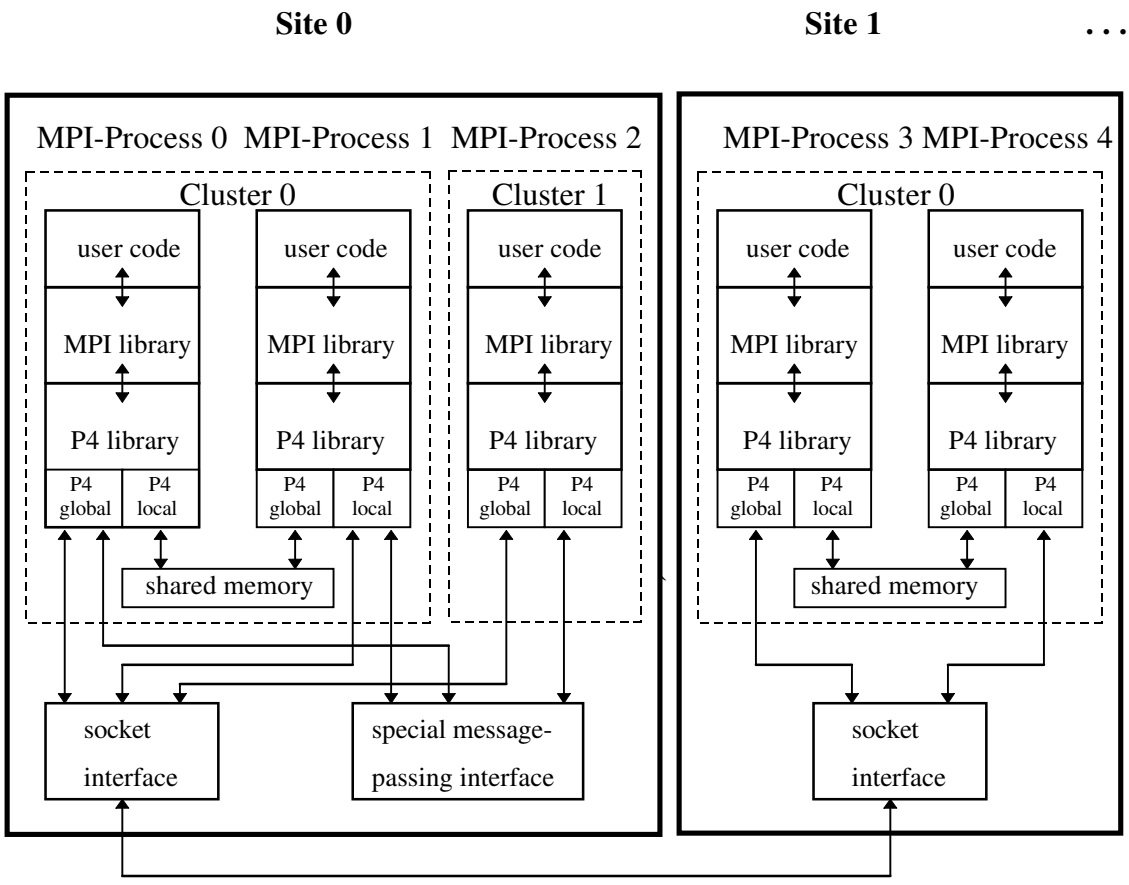


Figure 4 Basic MPICH structure under UNIX

Figure 4 also shows p4's two major data structures. Each p4 process stores a copy of a structure **`p4_global`** with information about all processes within the parallel application. Additionally, a private data structure **`p4_local`** contains data such as the rank in the whole application and a queue of buffered messages.

As mentioned earlier in this section, p4 as the underlying communication subsystem is also in charge of the definition of an MPI application's process structure. As described in [Butler92], it uses a configuration file which contains a line for each cluster determining 1) the site the cluster is located on and 2) the executable file to be executed (multiple times). Along with p4 this configuration file scheme is adopted for WinMPI.

2.3. MPI Example Program

In this section, a simple example program will illustrate the idea of parallel programming. The program uses the SPMD paradigm, i.e. all MPI processes run identical code (for heterogeneous environments the source code). Since illustrating all features of MPI in a single program is impossible; only point-to-point communication and global synchronization are shown.

The example program performs the computation of π. From the equation

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

π can be calculated by numerically integrating the area beneath the function $y = \frac{4}{1+x^2}$ in the range $0 \le x \le 1$, as shown in Figure 5.

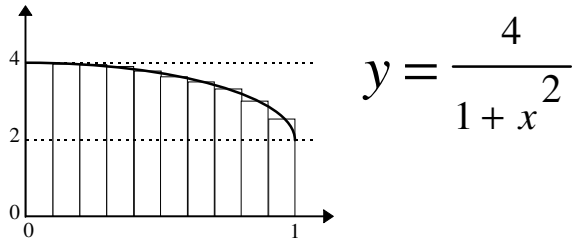


Figure 5 Calculation of π by numerical integration

For the numerical integration, the sizes of all rectangles are computed and added together. Now, this work is easily divided into several pieces. If *n* processes work together to sum up *m* rectangles, Process 0 takes rectangles 0, *n, 2n, 3n, ...*; Process 1 takes rectangles 1, *n+1, 2n+1, 3n+1, ...*; and so on. Figure 6 illustrates this division for *n* = 3 and *m* = 10.

Each process adds up $\frac{m}{n}$ rectangles Afterwards, the partial sums are added together by MPI-Process 0. The execution time decreases almost by factor *n*.

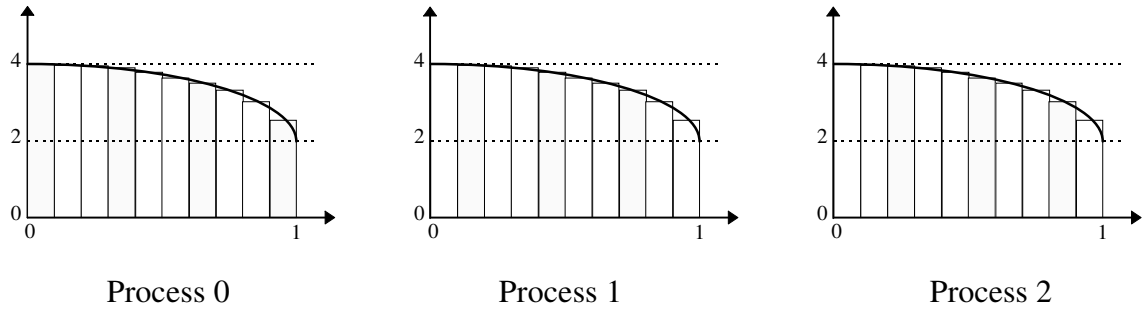


Figure 6 Distribution of 10 rectangles among 3 processes

The program in Figure 7 performs the numerical computation of π as described above.

```
/*****************************************************************
*                                                                *
*  pimpi.c                                                       *
*  Calculation of PI (numerical integration method)              *
*  MPI for UNIX version                                          *
*  Joerg Meyer       University of Nebraska at Omaha             *
*  11/15/94          Computer Science Department                 *
*                                                                *
*****************************************************************/

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

/*****************************************************************
*                                                                *
*  compute_interval - numerical integration over                 *
*                     part of interval                           *
*                                                                *
*****************************************************************/
double compute_interval (myrank, ntasks, intervals)
    int myrank, ntasks;
    long intervals;
{
    double width, x, localsum;
    long j;

    width = 1.0 / intervals;  /* width of single stripe */
    localsum = 0.0;
    for (j = myrank; j < intervals; j += ntasks)
    {
```

```
        x = (j + 0.5) * width;
        localsum += 4 / (1 + x * x);
    }
    return (localsum * width);      /* size of area */
}

int main (argc, argv)
    int argc;
    char **argv;
{
    long intervals;
    int myrank, ranksize;
    double pi, di;
    int i;
    MPI_Status status;

    MPI_Init (&argc, &argv);       /* initialize MPI system */
    MPI_Comm_rank (MPI_COMM_WORLD, &myrank);    /* my place in MPI system
*/
    MPI_Comm_size (MPI_COMM_WORLD, &ranksize);  /* size ofMPI system */

    if (myrank == 0)               /* I am the parent/master */
        intervals = atoi (argv[1]);

    MPI_Barrier (MPI_COMM_WORLD);  /* make sure all MPI tasks are running
*/
    if (myrank == 0)               /* I am the parent/master */
    {
/* distribute parameter */
        for (i = 1; i < ranksize; i++)
            MPI_Send (&intervals, 1, MPI_LONG, i, 98, MPI_COMM_WORLD);
    }
    else  /* I am a child/slave */
/* receive parameters */
        MPI_Recv (&intervals, 1, MPI_LONG, 0, 98, MPI_COMM_WORLD, &status);

/* compute my portion of interval */
    pi = compute_interval (myrank, ranksize, intervals);

    if (myrank == 0)   /* I am the parent/master */
/* collect results, add up, and print results */
    {
        for (i = 1; i < ranksize; i++)
        {
            MPI_Recv (&di, 1, MPI_DOUBLE, i, 99, MPI_COMM_WORLD, &status);
            pi += di;
        }
        printf ("Pi estimation: %14.12lf\n", pi);
        printf ("%d tasks used\n", ranksize);
    }
    else  /* I am a child/slave */
/* send my result back to parent/master */
        MPI_Send (&pi, 1, MPI_DOUBLE, 0, 99, MPI_COMM_WORLD);

    MPI_Finalize ();
    return 0;
}
```

Figure 7 MPI example program - Computation of π

The program expects the number of rectangles in the interval from 0 to 1 as a command line argument.[6] Each process invokes **`MPI_Init()`** first to initialize all MPI data structures. Next, each process obtains its rank and the number of MPI processes within the application by calling **`MPI_Comm_rank()`** and **`MPI_Comm_size()`**. In the following, the process with rank 0 is called the master; all other processes are slaves. The master is responsible for cooperation among all processes.

Before the processes begin executing the algorithm, a call to **`MPI_Barrier()`** blocks a process until all processes have invoked that function, ensuring that all processes have reached a common point.[7] Next, the master sends the number of intervals to all slaves by **`MPI_Send()`**. The slaves receive this value via **`MPI_Recv()`**. Now each process has the necessary information to perform its share of the work: the number of rectangles (intervals), the number of processes, and its own rank.

Each process passes these values to **`compute_interval()`**, which performs the computation. The parameter **`myrank`**, which is different for each process, determines the group of intervals to be added.

Each slave sends the size of the area assigned to it back to the master by **`MPI_Send()`**. The master receives all values via **`MPI_Recv()`** and adds them together, along with its own sum, to obtain the sum of all intervals. This value is the estimation of π, which is printed.[8] After a call to **`MPI_Finalize()`**, all processes terminate.

Chapter 4 will present the WinMPI version of this example program and discuss the porting procedure of an MPICH application to WinMPI and vice versa. Except for a few peculiarities, this procedure can be performed in a straightforward manner.

---

[6] Another command line parameter for p4 would be the configuration-file name, which determines the number and location of processes to run.

[7] This call is not important for the correct program execution here. It was originally included to permit more accurate execution-time measurements. As timing function could be the next statement. (p4 causes considerable start-up delays, which distort speedup measurements.)

[8] As the next statements, the current time could be obtained and the total execution time could be determined.

2.4. Why MPI for MS-Windows 3.1

One of the main goals for the MPI standard design is portability. A message-passing standard is useful only if implemented on virtually all computer systems that support concurrent execution of programs, even when very different architectures and programming paradigms are used. As discussed in Section 2.1, MPI is already available for a wide variety of shared memory and distributed memory systems and for interconnected workstations. However, since all MPI ports target workstations or parallel systems running derivatives of the operating system UNIX, potential MPI users are required to 1) have access to expensive parallel computers or UNIX workstations, and 2) be familiar with the basics of the UNIX operating system

Apart from UNIX, another multitasking operating system[9] of wide acceptance is MS-Windows 3.1. It is not as powerful as UNIX and is only a single-user operating system. However, MS-Windows 3.1 has considerable advantages over UNIX. The most important for this project are that 1) Windows runs on IBM-compatible personal computers (PC's), which are, on average, less expensive than UNIX workstations and that 2) MS-Windows is accepted by and available to potential MPI users that are not familiar with or do not want to deal with UNIX.

What about using WinMPI to initially write the program, test, debug, and run it before it is moved to a parallel machine? And is WinMPI not an appealing option for people in the significantly larger PC community? In Section 2.4.1 and 2.4.2, both thought are discussed in greater detail.

[9] A multitasking operating system can run several programs concurrently. On single-processor systems, the operating system switches between programs in short periods of time, creating the illusion of concurrent execution.

2.4.1. MPI for Windows as Development Tool

The results of this project show the possibility for implementing WinMPI in a form which permits :

- Implementing MPI programs
- Testing and debugging
- Evaluation of run-time behavior

The most important issue is the portability of MPI programs. As will be discussed in Chapter 4, this problem could be resolved in a satisfactory way. The necessity of minor changes may arise when moving MPI programs between UNIX and Windows systems, but these changes are obvious and suggest the usage of a conversion tool which is easy to implement.

MPI programs can be implemented independently of the availability of the parallel machine or workstation cluster. The whole phase of writing the source code is done under MS-Windows. The program is tested and its performance evaluated. At a later stage, the program is easily moved to the parallel machine where the test results are verified. The program is finally tested and tuned on the parallel machine until completed.

As a result, a significant amount of work can move from the parallel system to the PC. The parallel computer is used only for executing the completed MPI program, permitting a more efficient usage of this expensive hardware.

Hence, WinMPI is an appealing tool for experienced MPI users that want to write MPI applications in a more cost-effective way.

2.4.2. MPI for Windows as Learning Tool

The second field where MPI for Windows is useful is within the educational sector. If MPI becomes a widely accepted parallel programming interface standard, it should use all

opportunities to provide access for interested people to MPI. In order to enhance the acceptability of MPI, a great achievement would be if MPI were taught to students in universities or even in high schools. Today, many schools are equipped with computers such as IBM-compatible PC's or Macintoshes. In general, students are more familiar with MS-DOS and MS-Windows than with UNIX. Additionally, many families own IBM-compatible personal computers running MS-Windows. WinMPI provides the opportunity for gaining parallel programming experience to these people.

MPI for Windows will help the MPI standard become the generally accepted message-passing interface standard by making it available to a large group of users who were previously excluded from this current stage of parallel programming research.

2.5. Outlook: MPI for MS-Windows NT

Recently, Microsoft released the operating system MS-Windows NT 3.5, which is currently gaining increasing acceptance. Although backward-compatible, the system is far more than a revised, slightly improved upgrade, as the version number 3.5 might suggest. "NT" stands for "New Technology", suggesting a new generation of operating systems. Indeed, Windows NT is a large step forward from MS-Windows 3.1 Its new features include:

- 32-bit operating system
- Preemptive multitasking
- Flat 32-bit memory model - no memory segmentation
- Full protection for application address space
- Interfaces for shared memory, semaphores, and threads

Windows NT still runs MS-Windows 3.1 applications. Hence, WinMPI programs are executable without recompilation. For full compatibility, however, Windows NT uses a particular mode that lacks all its new capabilities. The problems discussed in Chapter 3 and 4 are still present; thus, only a real port of MPI to Windows NT enables its new features to come into play. Major advantages are:

- Multitasking issues are hidden from the programmer.
- System is protected from malicious applications and is very stable.
- 32-bit memory model resolves memory segmentation problem.
- Inter-process communication and synchronization is handled more UNIX-like.

Since the MS-Windows NT programming approach is derived from Windows 3.1 rather than from UNIX, WinMPI can be chosen as the base for a Windows NT port. While the MPI layer requires only slight changes, some NT concepts will heavily affect the p4 layer. Fortunately, no tangible new solutions are necessary. Windows NT will require UNIX-like approaches. [Niezgoda94] discusses porting shared memory, process management,

and semaphore calls from UNIX to Windows NT. Source code from the original UNIX-p4 files will help implement the NT port.

Since Microsoft is likely to establish MS-Windows NT as the de-facto operating system standard for PC's in the near future, WinNTMPI is a worthwhile enterprise.[10]

[10] All facts stated here apply to Windows 95 as well. Windows 95 intends to replace MS-Windows 3.1 but is probably not available until after II. Quarter 1995.

3. Implementation for Microsoft Windows 3.1

3.1. The MS-Windows 3.1 Programming Environment

This section gives a short introduction to the programming environment of MS-Windows 3.1. This programming environment provides rich functionality and supports all features of a graphical user interface. Especially the aspects affecting porting MPI to this programming environment are discussed.

3.1.1. Basic MS-Windows Program Structure

MS-Windows 3.1 uses a programming approach that differs from other well-known operating systems such as UNIX and MS-DOS. Figure 8 shows the basic structure of a Windows program . Besides the function **`WinMain()`** that is executed initially, a function **`WndProc()`** is required to process messages sent by the Windows kernel.

```
#include <windows.h>

long FAR PASCAL _export WndProc (HWND, UINT, UINT, LONG) ;

int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance,
                    LPSTR lpszCmdParam, int nCmdShow)
{
    static char szAppName[] = "Generic" ;
    HWND        hwnd ;
    MSG         msg ;
    WNDCLASS    wndclass ;

    if (!hPrevInstance)
    {
        /* definition of parameters for
           window class */
        RegisterClass (&wndclass) ;
    }
    hwnd = CreateWindow (szAppName,         // window class name
                         /* default values for
                            window */) ;
    ShowWindow (hwnd, nCmdShow) ;
    UpdateWindow (hwnd) ;

    while (GetMessage (&msg, NULL, 0, 0))
    {
```

```
        TranslateMessage (&msg) ;
        DispatchMessage (&msg) ;
    }
    return msg.wParam ;
}

long FAR PASCAL _export WndProc (HWND hwnd, UINT message, UINT wParam,
                                                          LONG lParam)
{
    switch (message)
    {
        case WM_CREATE:
            /* actions for message WM_CREATE */
               return 0 ;
        case WM_PAINT:
            /* actions for message WM_PAINT */
               return 0 ;
        case WM_DESTROY:
            PostQuitMessage (0) ;
            return 0 ;
    }
    return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

Figure 8 Basic C Program Structure for MS-Windows 3.1

The main difference is that, unlike the C **`main()`** function, no code is placed in **`WinMain()`** to perform the task for which the program is written. This is due to the message-driven programming approach adopted by MS-Windows. **`WinMain()`** only initializes the window and makes it visible by calling a Windows function. Then it processes messages in a loop. The program's functionality is placed or called from a function named **`WndProc()`**, not from **`WinMain()`**. Although included in the application, **`WndProc()`** is called from the Windows-Kernel. A similar approach is found in UNIX only for signal handlers. Windows makes heavy use of this "call-back" function and invokes **`WndProc()`** whenever sending a message to the application. Messages are used mainly to notify the application of relevant events, such as keystrokes at the keyboard, mouse movement, requests to repaint a window, movement of a window, clicking a button, and many others. In **`WndProc()`**, an application has the chance to process these messages and take the appropriate actions; otherwise, it simply passes them back to Windows (via **`DefWindowProc()`**), which executes a default action.

The message-driven approach described above is contrary to applications in a UNIX environment where an application executes standard procedures or system calls to receive input data such as characters from the keyboard. Because a UNIX-MPI application will employ the UNIX approach, a way of embedding it into the Windows-program structure must be found.

3.1.2. Memory Models

Another peculiarity of Windows applications is the choice among several memory models, depending on size and functionality of an application.

MS-Windows runs on IBM-compatible personal computers, using processors from the INTEL 80x86 16-bit microprocessor family. Many years ago, during the design of the first processor of that family, the 8086, the decision was made to use a segmented address space. This decision appeared prudent since it permitted many application programs written for 8-bit microprocessors to be easily rewritten for the 8086. The basic idea is to break a memory address into two pieces: the address of the beginning of a segment in the physical address space and an offset in that segment. For the 8086, a 16 bit segment address and a 16-bit offset are combined to a 20-bit address resulting in segment sizes of 64Kbytes and a 1Mbyte physical address space. An application program is easily loaded into different segments and, since it uses only offsets to reference executable code and data, is executed without changes once the segment address is set appropriately in a 8086 register.

The first problem arose when, along with increasing main memory sizes, the programs provided more functionality and required more memory space. One 64KByte segment became too small to hold both the code and the data, necessitating several segments. Moreover, referencing an object in memory required both segment address and segment offset. Depending on its particular needs, an application program can be compiled using one of the memory models shown in Table 1 (two less important models are omitted).

Memory Model	Code Segments	Data Segments
small	1 only	1 only
compact	1 only	multiple
medium	multiple	1 only
large	multiple	multiple

Table 1 Memory models

When an application uses only one segment for code or data, only the 16-bit offset is necessary and the C compiler generates `near pointers` of 16 bits length for code or data. If multiple segments are used for code or data, the C compiler includes both the segment address and the offset in a 32-bit `far pointer`. The penalty of using larger memory models is reduced speed and larger programs, since 32-bit pointers are loaded and stored to reference memory objects.

For MS-Windows 3.1 applications, the penalty for using the compact or large memory model is even more severe. In addition to the drawbacks mentioned above, the following points were discussed in [Petzold92]:

1. Windows uses a clever memory management technique to avoid memory fragmentation. Unless a memory segment is locked, Windows will move that segment in memory, if necessary, to obtain a contiguous memory block of sufficient size. This method, called compaction, permits the effective use of the memory resources but forces additional restrictions on the programmer. Moving a memory segment means that all far pointers to that segment become invalid. Hence, far pointers can only be used as long as the memory block is locked and can never be moved. Using a compact or large memory model, the programmer has no means to always update far pointers since this is handled by the compiler. As the only solution, data segments referenced by far pointers are fixed in memory, seriously impeding memory management.

   It was not mentioned in [Petzold92] that this is only true for MS-Windows running in real mode which is no longer supported by version 3.1. The severe restrictions of real mode, especially the maximum memory size of 640 Kbytes, make it infeasible for WinMPI. Therefore, WinMPI is not backward-compatible to Windows versions prior

to 3.1. Both standard and extended mode of MS-Windows 3.1 use the protected mode of the 80286 and its successors. The usage of selectors [Nelson91] makes far pointers independent of the physical location of a memory object and greatly simplifies memory issues. Applications can use far pointers and keep memory segments locked without affecting memory management.

2. Applications using the compact or large memory model can only run one at a time. This restriction was mentioned in [Petzold92] without explanations. Microsoft has published an article that deals with the topic of using the large memory model [Microsoft94]. It confirms that restriction and provides some solutions, which, however, are not practical for WinMPI. The ability to start an application multiple times is essential(correspondes to the SPMD model).

As a consequence, the static library containing all MPI functions is compiled using the medium memory model. This decision introduces several problems, but is still the best alternative. Consequences for writing/porting MPI application programs are discussed in Chapter 4.

3.1.3. Non-preemptive Multitasking

When the first Windows version was released in 1985, multitasking was considered to be an exiting feature for IBM-compatible PC's. Performance and design evaluations led to the decision to employ non-preemptive multitasking for Windows.

In non-preemptive multitasking, an application, usually one task, can maintain control over the CPU as long as desired. The operating system kernel has no means to forcibly gain control from a malicious application. Cooperation among all applications is assumed. This policy is tolerable in a single-user environment. Non-preemptive multitasking simplifies certain operations but can cause additional problems. These problems are discussed here with regards to an MPI implementation:

- MPI applications require multiple tasks to cooperate for message exchange and synchronization. UNIX ensures the concurrent execution of processes, even enforcing it. Therefore, MPI applications need not contain any statement to ensure the execution of other processes within that MPI application. Under MS-Windows, a task has to explicitly relinquish the processor which is difficult to accomplish under non-preemptive multitasking. It is probably the most challenging task of the whole implementation. In general, the solution is two-fold:

  1. If an operation such as receiving a message or acquiring a lock blocks, the possible wait is obviously infinite unless another task is given the chance to send a message or release the lock. Therefore, the receive and lock functions provided by the MPI library relinquish the processor, relieving the user from that issue.

  2. When an MPI task performs a long computation between two communication or synchronization calls, it maintains control over the processor and all other Windows programs halt. Windows is unable to keep other applications running. The user needs to deal with this problem, either by accepting this shortcoming or by frequently invoking a function **`MPI_Win_yield()`** provided by the WinMPI library. This MPI function is the only extension of the MPI Standard that cannot be omitted.

- The implementation aspects of the locking mechanism are simplified. In Windows, a task can rely on maintaining control over the processor as long as it wishes. As a result, achieving mutual exclusion is easy by simply not calling any functions that might cause a rescheduling. For the locking mechanism, the test-and-set operation need not be atomic; it cannot be interrupted. On the other hand, simply "spinning" on a lock results in an infinite loop because no other process has the chance to release that lock.

- Program development can be difficult. Whenever a programming error results in a malfunctioning program, the whole system becomes unstable and cannot be recovered.

Usually, rebooting the computer is necessary to ensure a stable and consistent system state.

Today non-preemptive multitasking is considered obsolete. However, various Windows applications rely on it. For backward-compatibility, Windows remains non-preemptive. For the discussion of the actual implementation of receive and locking mechanism, refer to Section 3.3.4 and 3.3.6, respectively.

3.1.4. Window Management

While MS-DOS and UNIX provide character-based user interfaces, MS-Windows supports a graphical user interface. The monitor screen is not a square of text characters but of colored pixels. A Windows application creates windows on the screen and fills them with pixels, lines, graphics, or text of different fonts and colors. At any time, an application must be able to redraw the entire content of a window after it was hidden or moved on the screen.

A WinMPI application will consist of multiple Windows tasks, only one of which will open a window and display the output of all MPI processes. However, for input from the keyboard and output to the screen, Windows does not provide the UNIX abstraction of a standard input and standard output stream **`stdin`** and **`stdout`**.[11] WinMPI must emulate these abstractions and provide the accompanying functions. Refer to Section 3.3.7 for a more detailed discussion of input/output issues.

[11] In UNIX, **St**an**d**ard **in**put and **st**an**d**ard **out**put streams provide access to keyboard and screen of a UNIX terminal and can be redirected to other devices

3.2. Structure of MPI under MS-Windows 3.1

After the embedding of an MPI program into an MS-Windows program is determined, the general structure of MPI (as shown for UNIX in Figure 4) has to be revised.

This redesign requires the replacement of unavailable or inappropriate components for MS-Windows. The following three points deserve special consideration:

1. All MPI processes run on the same machine, so networking facilities are not necessary. All tasks can communicate using shared memory.
2. MS-Windows does not provide a convenient software interface for tasks to use shared memory.
3. MPI suggests the simultaneous execution of a larger number of tasks. The rich MPI functionality causes executable files of significant size. As a result, the number of concurrent tasks may be limited by the available main memory. Windows uses memory in a very efficient manner. If multiple instances of a program run simultaneously, the code segments are held only once in memory. However, if an MPI application comprises several executables, then each must stay in memory, even though the largest part, the MPI library is kept multiple times.

All three issues mentioned above suggest the usage of a dynamic link library (DLL). A DLL contains functions that are linked to an application when it starts execution. It is no longer necessary to include all referenced library functions in an executable file. A DLL permits the sharing of program code (and data) among Windows tasks; thus, DLL's use memory very efficiently. Moreover, DLL can store data; hence, tasks can exchange data by calling functions in a DLL. Further information about Windows DLLs can be found in [Conger92].

For WinMPI, the whole p4 code was rewritten for a DLL.

At the present time, the two-layer approach of MPICH, the MPI implementation from Argonne National Laboratory, appears to have great advantage over a single-layer approach. The lower layer comprises p4, a parallel programming environment which provides the software interface for :

- message passing
- synchronization primitives
- execution of multiple cooperating processes.

The p4 system is available for a variety of computer systems that are feasible for parallel programming (parallel and distributed systems). It handles all the architecture-dependent functions in a transparent way, allowing programs written for p4 independence of the type of machine(s) they are executed on. p4 is available in version 1.4 and unlikely to experience significant interface changes [Lusk94].

The upper layer of MPICH comprises the MPI library. MPICH is still under development. At the time of this writing, the MPICH implementation used as a base for Windows is evolving rapidly. It relies on p4 as underlying programming system to simplify the implementation. The current implementation of WinMPI version 0.99 uses the MPICH version from June 17, 1994. At the time of this writing, the latest MPICH version is dated November 3, 1994; the release of future versions is certain. As a result, a new version of WinMPI requires verification or repetition of all source code changes.

Since further development on p4 is unlikely, the changes in MPICH will affect only the upper layer. Therefore, the most prudent way of porting MPICH to MS-Windows is to change the p4 layer to use the MS-Windows capabilities while limiting changes in the MPI layer. These considerations resulted in the structure of WinMPI shown in Figure 9.

The changes are twofold. First, all communication is executed by functions from the same DLL. The only communication mechanism available is memory allocated by the DLL from Windows' global heap and marked as shared memory. All messages are stored temporarily in these memory segments.

The second change involves the data structures used by p4 processes. In MPICH, each p4 process stores a copy of a structure **`p4_global`** with information about all processes within the p4 application, such as number of processes, and certain information about all other processes. Additionally, each process has a private structure **`p4_local`** containing local data such as its own rank and a queue of buffered messages. In a DLL only one copy of **`p4_global`** has to be held while the **`p4_local`** structures of all processes are placed in a table.

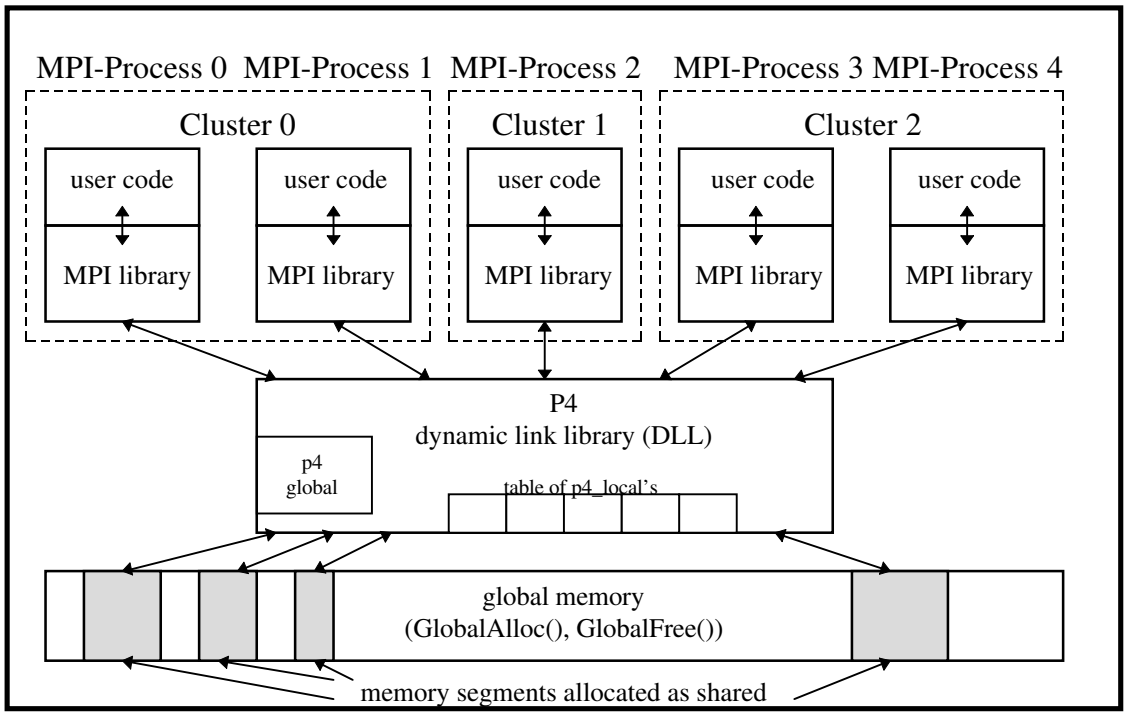


Figure 9 Basic MPI structure under MS-Windows 3.1

The experience gained during the porting process suggests that the WinMPI structure was designed in an optimal way, exploiting Windows' memory management capabilities. Porting the upper layer was finished without great difficulties in the minimum time. Tests showed the excellent performance of the p4 communication subsystem.

3.3. Mapping of MPI Functionality to MS-Windows 3.1

3.3.1. Placement of `main()` Function

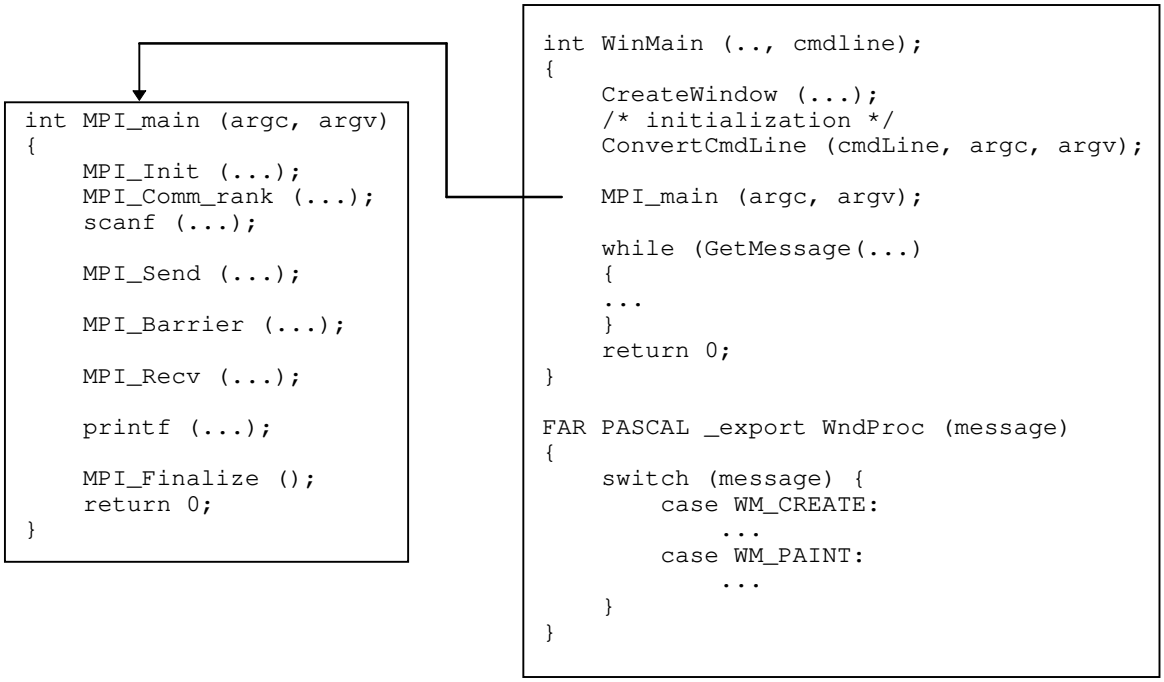
In section 3.1.1, the basic structure of an MS-Windows program was discussed. Due to Windows' programming approach, embedding the **`main()`** function of an MPI program into a Windows program is not a trivial task.

The Microsoft Visual C++ package offers a convenient way of running a Standard-C program under MS-Windows. The C program is simply compiled as a QuickWin application, running in a window representing a text-mode screen. Slight problems may arise when moving Standard-C programs for UNIX to MS-Windows or MS-DOS, mainly due to the different representations of the integer datatype and to the limited size of static arrays. Fortunately, solutions are usually straightforward.

QuickWin has the great advantage that the UNIX abstractions standard input and standard output stream and functions such as **`scanf()`**, **`printf()`**, and **`getchar()`** are available. A running QuickWin application owns a window where all input and output appears. The Windows-typical functionality such as creating the output window and processing of messages in a call-back function is hidden from the user.

If an MPI application ran as a QuickWin program, one window for each MPI process would be created. The output of the MPI application would appear in the window of MPI-Process 0. All other windows would remain unused. Since preventing the creation of these windows is impossible, a QuickWin program is not an acceptable solution for an MPI application.

As a result, embedding the MPI program in a MS-Windows program as shown in Figure 10 is the only remaining option. Hence, after the initialization part and before entering the message loop, **`WinMain()`** has to call the MPI program's **`main()`** function.



```
int MPI_main (argc, argv)
{
    MPI_Init (...);
    MPI_Comm_rank (...);
    scanf (...);

    MPI_Send (...);

    MPI_Barrier (...);

    MPI_Recv (...);

    printf (...);

    MPI_Finalize ();
    return 0;
}
```

```
int WinMain (.., cmdline);
{
    CreateWindow (...);
    /* initialization */
    ConvertCmdLine (cmdLine, argc, argv);

    MPI_main (argc, argv);

    while (GetMessage(...)
    {
    ...
    }
    return 0;
}

FAR PASCAL _export WndProc (message)
{
    switch (message) {
        case WM_CREATE:
            ...
        case WM_PAINT:
            ...
    }
}
```

Figure 10 Embedding of MPI application into MS-Windows program

Unlike Standard-C programs which use the **`main()`** function as a program starting point, Windows programs begin execution in the **`WinMain()`** function. The attempt to call the **`main()`** function of the MPI program from **`WinMain()`** fails, however, probably since the Microsoft Visual C++ package uses already a label **`main`** internally. The only solution is to change the name of the MPI program's entry point from **`main()`** to **`MPI_main().`** Note that the name of the **`main()`** function in the MPI program has changed to **`MPI_main()`** in Figure 10.

The message loop in **`WinMain()`** is not reached until **`MPI_main()`** returns. In order for Windows to continue other tasks, message-processing functions such as **`GetMessage()`** have to be called within **`MPI_main().`** Several MPI functions contain a message loop. For longer sections of MPI application code without a call to an MPI function, a call to **`MPI_Win_yield()`** can be inserted to permit multitasking.

Another difference between **WinMain()** and **MPI_main()** is the way command line arguments are passed to these functions. The functions are declared as follows:

```
int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance,
                    LPSTR lpszCmdLine, int nCmdShow);
int MPI_main (int argc, LPPSTR argv);
```

where LPSTR and LPPSTR are defined as:

```
typedef char far * LPSTR;
typedef LPSTR far * LPPSTR;
```

MS-Windows passes the command-line arguments as a single character string to **WinMain().** The programmer must process this string to check for command-line arguments. **MPI_main()** expects an array of pointers to all single arguments. Therefore, **WinMain()** calls a function to convert the single parameter string to an array and passes it to **MPI_main().** By calling **MPI_Init()**, a pointer to that array is finally passed to the p4-DLL, requiring the usage of a far pointer to an array of far pointers.

Despite the differences in using **main()** and **WinMain()**, a way was found to integrate the **main()** function of an MPI application in a Windows program. The Windows environment is almost transparent to a UNIX-like MPI application, although the programmer is not entirely dismissed from the whole issue.

3.3.2. The Upper Layer in MPI for Windows

As illustrated in Section 3.2, MPICH for UNIX is divided into two layers. The lower layer consists of p4 and is unlikely to change significantly. The upper or MPI layer requires steady enhancements and corrections, i.e. experiences significant modifications. Therefore, one of the goals for WinMPI is avoiding changes in the upper layer as much as possible. This way, a new MPICH version can be ported to Windows in relatively short time. Indeed, the efforts for upper layer modifications are considerably limited to the following procedures:

1) WinMPI applications use the medium memory model, i.e. all data pointers are near pointers, limiting the amount of data to one segment or 64 Kbytes. For data transfer between an MPI application and the p4-DLL, the p4 interface uses far data pointers without exceptions. Moreover, the upper layer allocates memory blocks from Windows' global heap dynamically, requiring far pointers. Therefore, all pointers in the source code were declared as far pointers, including both the MPI function interface and internal pointer variables.

2) The medium memory model limits the sum of global variables and stack to 64 Kbytes. In order to provide as much space as possible to the user code, the static MPI library representing the upper layer allocates memory from the global heap (except for a block for command-line arguments). Thus, all memory allocation functions have been changed to the p4 functions **p4_shmalloc()** or **p4_shfree()**. As a result, about 42 Kbytes are currently available to the user. [12]

3) The third problem is caused by an ordinary gap in the standard for the C language, left due to efficiency considerations. Since the size of the integer datatype is not defined, compilers always choose what best suits the hardware. While UNIX workstations employ 32-bit microprocessors, MS-Windows 3.1. is only a 16-bit operating system designed during the era of 16-bit microprocessors. Thus, under UNIX, integer variables are 32-bit values while only 16 bit wide for Windows. Many situations are imaginable where a C functions works with 32-bit integers but fails with 16-bit integers due to an undetected overflow. Tracking these situations is difficult because they occur only occasionally and make written source code insecure. Moreover, sometimes source code relies on the assumptions that integers are 32-bit values, for example if certain bits are used as flags. A common method was utilized to prevent difficulties from the beginning. The global headerfile **mpi.h** contains the following lines:

[12] assuming stack size of 8 Kbytes

```
#ifndef Int
#define Int long
#endif
```

Using the replace function of the text editor, all occurrences of **int** were replaced by **Int**. As a result, all 16-bit integers are changed to 32-bit long variables, which match the 32-bit integers of UNIX machines.

4) C source code for UNIX is still written in K&R C. The improved ANSI-C standard (or Standard-C) is not supported by all UNIX-C compilers. One major difference regards the declaration and definition of functions. MPICH code is still written in K&R C, which lacks any kind of type-checking for argument lists passed to functions. Code that works for UNIX can fail under Windows. For example, a constant number `1` is passed to a function as 32-bit value under UNIX but only as 16-bit value in the Windows code if there is no function declaration. Therefore, all function definitions were changed to the Standard-C format, and a full list of function declarations was included in a headerfile to enable type-checking to the full extent. This measure was the right decision because it uncovered flaws whose discovery would otherwise have taken significant efforts.

The changes to the MPI layer described here may appear difficult and time-consuming but all four measures are straight-forward and unlikely to produce additional flaws in the source code.

After describing the modifications to the upper layer, the changes to the p4 layer are discussed in the following sections. The lower layer deals with communication, synchronization, and startup mechanisms and provides these functions to the upper MPI layer in a transparent way, very much like under UNIX. Due to the considerable differences between UNIX and MS-Windows, the p4-layer modifications required the most time and effort.

3.3.3. `MPI_Win_yield()` - Message-Loop Function

As discussed in Section 3.1.3, the non-preemptive nature of Windows requires MPI processes to cooperate with other Windows tasks by relinquishing the processor frequently. As the only extension to MPI, WinMPI provides a function **`MPI_Win_yield()`** for that purpose, which does nothing but call **`P4WinMessageLoop()`** in the p4-DLL.

Figure 11 shows the source code of **`P4WinMessageLoop()`**. It performs operations similar to the message loop at the end of **`WinMain()`**. The while-loop receives and processes messages as long as Windows' message queue is not empty. MS-Windows schedules tasks while executing **`PeekMessage()`** or **`GetMessage()`**. As a result, the MPI task relinquishes the processor, enabling other Windows tasks to run.

```
P4VOID P4WinMessageLoop (void)
{
    MSG msg;

    while (PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE))
    {
        if (!GetMessage (&msg, NULL, 0, 0))
            p4_error ("Received WM_QUIT - Aborting", p4_get_my_id ());
        TranslateMessage (&msg) ;
        DispatchMessage (&msg) ;
    }
}
```

Figure 11 Processing of Windows messages during MPI/p4 program execution

`GetMessage()` always blocks until a message becomes available since the Windows programming approach assumes that tasks are idle without a message to respond to. This assertion is not true for MPI tasks. Therefore, **`GetMessage()`** is invoked only if signals a nonempty message queue.

In **`WinMain()`** the message loop is exited only if a **`WM_QUIT`** message is received, resulting in exiting **`WinMain()`** and termination of the task. **`P4WinMessageLoop()`**

is executed only during normal run and should not receive a **WM_QUIT** message. Therefore, only an error condition or the user can cause the generation of a **WM_QUIT** message. A call to **p4_error()** generates a dialog box to inform the user about the terminating MPI task and never returns.

The decision where to place the **MPI_Win_yield()** function in a WinMPI application is left to the programmer. Section 4.1 uses an example program to give giudelines.

3.3.4. Communication Primitives

As discussed in Section 3.2, shared memory is the only suitable form of communication between MPI processes. p4 for UNIX has the ability of transmitting messages using shared memory but only for message exchange between tasks located in the same cluster. Message buffers and the function interface for the allocating, queuing, and transferring of messages are available. p4 for Windows uses all these facilities, but the source code was modified to handle message exchange cluster-independently. Instead of the cluster rank assigned to a process, the communication functions use the global rank of a process to specify sender and receiver.

In p4 for UNIX, clusters play an important role for global operations such as broadcast and reduce functions. Effective implementations employ binary trees for global operations. One process in each p4 cluster, the “cluster master”, comprises the root of a binary tree of all cluster processes, which communicate via shared memory. In turn, all cluster masters form a binary tree, employing another communication systems. For MS-Windows, this tree of trees was reorganized to form a cluster-independent global binary tree.

For performance measurements, the hardware clock of an IBM-compatible PC with a resolution of 55 milliseconds was not precise enough to deliver accurate transfer rates. Since the transmission of a message in memory requires only the allocation of a buffer and

a few pointer operations to move it from one queue to another, message transfer is very fast. In comparison, the scheduling of the receiving process is much more expensive.

3.3.5. Slave Processes

This section discusses the MPICH approach for starting MPI processes. The Windows procedure is presented and solutions to related problems are given.

In p4 for UNIX, the concurrent processes are started in two steps:

Step 1: The process started first is called "big master" and responsible for creating the "cluster master" processes (using **`rsh`** or a server process for both the local or a remote site).

Step 2: The "cluster master" processes create all slaves in their cluster. They execute a function **`create_xm_processes()`**, which is shown in short form in Figure 12.

```
int create_bm_processes(struct p4_procgroup *pg);
{
    /* ... */
for (slave_idx = 1; slave_idx <= nslaves; slave_idx++)
{
    slave_pid = fork_p4();
    if (slave_pid == 0)   /* At this point, we are the slave. */
    {
        p4_local = alloc_local_slave();
        p4_local->my_id = p4_get_my_id_from_proc();
        return;
    }

    /* master installing local slaves */
    /* ... */
    }
    return (nslaves);
}
```

Figure 12 Start of slave processes in **`create_bm_processes()`** in UNIX version of p4

In Step 2, the slaves are created by the UNIX system call **`fork()`**. As a result, **`create_xm_processes()`** is called by the "cluster master," but both master and slaves return from that function.

The startup procedure for WinMPI is modified with regards to two aspects:

- All MPI processes run locally and can access all resources.
- MS-Windows does not provide a system call similar to **`fork()`**.

In WinMPI, "cluster masters" do not exist, the "big master" starts all tasks. The WinMPI version of **`create_xm_processes()`** in Figure 13 contains a loop for the slaves nested in a loop for all clusters.

```
Int create_bm_processes (struct p4_procgroup far *pg)
{
    /* ... */

    for (cluster_idx = 0; cluster_idx < pg->num_entries; cluster_idx++)
    {
        /* initialize cluster */
        pc = /* ... */

        for (; slave_idx < nclustslaves; slave_idx++)
        {
            hSlave = LoadModule(local_pg->slave_full_pathname, &parms);

            /* master installing local slaves */
            /* ... */
        }
    }
    return (p4_global->n_started_slaves);
}
```

Figure 13 Start of slave processes in **`create_bm_processes()`** in Windows version of p4

The absence of a **`fork()`**-like system call is a severe problem. Windows does not distinguish parent and child processes, but it provides a function **`LoadModule()`**, which starts an independent task. Unlike **`fork()`**, this call returns only in the calling function while the started process begins execution in **`WinMain()`**.

A complete restructuring of the p4 startup functions was necessary. Using **`fork()`** under UNIX, only a master process calls the **`MPI_Init()`** function, but all slaves return from it. In WinMPI, all processes enter **`MPI_Init()`**, and must check if they are master or slave since only the master can start other processes. A new mechanism was designed for this purpose, using the fact that all tasks call the **`create_xm_processes()`** in the same DLL. Local information stored there enables a process to check if it called the DLL first (master) or later (slave). For that mechanism to work, a call to **`get_new_p4_local()`** is hidden in **`WinMain()`** which assigns a rank to an MPI task even before the task enters **`MPI_main()`**.

Experiments showed that the start of sixteen WinMPI processes is possible. Currently, the dimensions of the p4 data structures permit a maximum of sixteen tasks. In case more processes become necessary, the p4-DLL can be rebuilt to meet these needs.

3.3.6. Synchronization Primitives

p4 provides functions for three different synchronization primitives:

- barriers
- monitors
- locks

Monitors are abstract data types for defining shared objects (resources) and for scheduling access to these objects in a multiprogramming environment. For a discussion of monitors, refer to [Singhal94]. p4 implements *barriers* by using *monitors*, which in turn rely on *locks*. Only the implementation of *locks* remains hardware-dependent. Functions for the initialization, locking, and unlocking of *locks* are provided.

In a preemptive multitasking environment, the lock functions are required to be atomic (non-interruptible) operations. Otherwise, concurrent processes using the same lock can produce race-conditions, causing the lock to fail. The locking function is the most critical.

An atomic test-and set instruction or an atomic swap-operation are usually used to implement it.

In Windows, the problems are essentially different. Since a process is never interrupted between certain system calls, race conditions are impossible. On the other hand, a process waiting for a lock to become unlocked in a loop may wait infinitely, unless it relinquishes the processor enabling other tasks to unlock the lock. Figure 14 shows p4's lock functions. **`p4_lock()`** calls **`P4WinMessageLoop()`**[13] to let other tasks run whenever it finds the lock locked.

```
P4VOID FAR PASCAL _export p4_lock_init (p4_lock_t far *l)
{
    *l = 0; // unlocked
}

P4VOID FAR PASCAL _export p4_lock (p4_lock_t far *l)
{
    while (*l != 0)
    {
        /* WIN: If locked, let others unlock it */
        P4WinMessageLoop ();
    }
    *l = 1; // locked now
}

P4VOID FAR PASCAL _export p4_unlock (p4_lock_t far *l)
{
    *l = 0; // unlocked
}
```

Figure 14 Basic locking functions in p4 for MS-Windows

The lock implementation shown here ensures message processing even in the case that an WinMPI process blocks. In case of a deadlock, it is still possible to terminate a WinMPI process by sending a **`WM_QUIT`** message.

[13] discussed in Section 3.3.3

3.3.7. Input/Output Functions

As discussed in Section 3.1.4, MS-Windows lacks the abstraction of a **stdin** and **stdout** stream for input from the keyboard and output to the screen (or window). Popular Standard-C functions like **printf(), _fputchar(), scanf()** and **getchar()** are not available. However, WinMPI must provide them for MPI programs, i.e. the abstract **stdin** and **stdout** byte streams must be implemented.

3.3.7.1. Standard Output

MPI-Process 0 (the "master" process) creates a window of the right size for the 80 by 24 characters of a terminal text screen. Since both MPI processes and p4-DLL need to print to the screen, the DLL contains the full functionality which is divided into two layers.

The lower layer keeps a two-dimensional array of characters **P4_screen**, containing the current screen content. Whenever MPI process 0 receives a **WM_PAINT** message, it prints the characters from the array in its window. The only interface between the layers is a function **P4_Puts()** which stores a character string in **P4_screen** and sends a **WM_PAINT** message to the master process. The lower layer keeps the cursor position, processes special characters such as '**\n**' and '**\b**', and scrolls the screen.

The Standard-C functions are implemented as follows: **_fputchar()** converts the character to a string of length 1, then passing it to **P4_Puts()**. **Puts()** simply calls **P4_Puts()**. **Printf()** is implemented as a call to **vprintf()**. **vprintf()** must expand the format string to a character string, using additional arguments and passes it to **P4_Puts()**. Windows offers a function **_vsnprintf()** that expands the format string to a character buffer. Now the most important **stdout** functions are available.

The above concept works only if **P4_Puts()** is an exported DLL function and anything else resides in the MPI application. The MS Visual C++ documentation states that **_vsnprintf()** is not available for DLL programming. Experiments showed that this

function is available but not guaranteed to work properly. One alternative would have been to rewrite the **`_vsnprintf()`** source code. Due to **`vprintf()`**'s extreme complexity, every attempt was made to avoid a reimplementation and a more elegant solution was found. As first step on the way to that solution, the explanation for **`vprintf()`**'s failure is given below.[14]

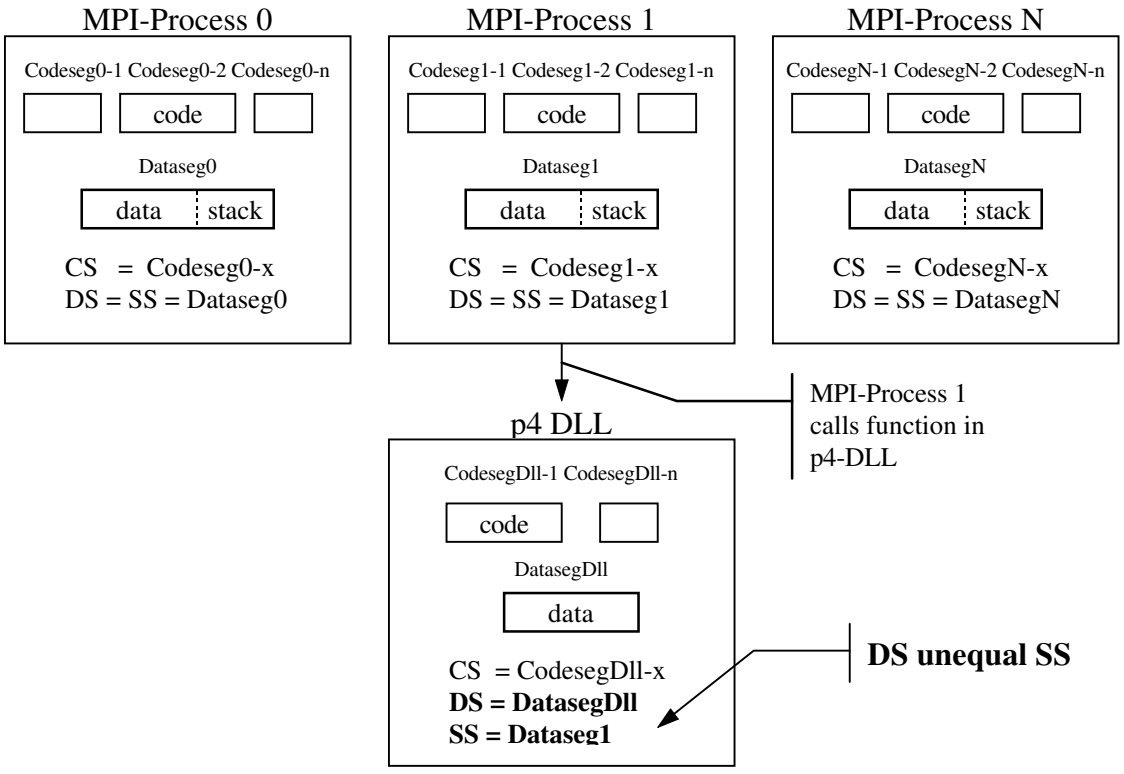


Figure 15 Data and stack segments of WinMPI processes and p4-DLL

The 80x86 processors store the segment addresses for code, data, and stack segment in registers CS, DS, and SS, respectively. DS points to global variables while SS is used to address automatic (local) variables. For the medium memory model, only one segment exists for data and stack together, and DS is equal to SS. The p4-DLL maintains a private

[14] [Petzold92] elaborates on the whole issue in greater detail.

data segment for local data. Hence, DS is set to another segment. SS remains unchanged and is used to pass arguments. As a result, during execution of DLL code, DS is unequal to SS. Figure 15 illustrates this issue.

Data pointers are near pointers. They contain only the offset address of data item but no information if located in the data or stack segment. As long as DS equals SS, the offset suffices to address the data item. As experiments showed, the **`_vsnprintf()`** function receives a near pointer to the argument list and interprets it as offset to DS, the DLL data segment. However, in **`vprintf()`**, it receives a pointer to arguments in the stack segment. As a solution, instead of passing a pointer to the arguments on the stack, **`printf()`** copies the arguments to an array in the data segment and passes a pointer to this array. This solution makes the source code non-portable, but saved a considerable amount of time and works well. Hence, the p4-DLL maintains its own versions of **`_fputchar()`**, **`puts()`**, **`vprintf()`**, and **`printf()`**; but still, all functions work on top of **`P4_Puts()`**.

3.3.7.2.Standard Input

WinMPI provides the most widely used standard input functions **`getchar()`**, **`_fgetchar()`**, **`fgets()`**, and **`scanf()`**.

`getchar()` and **`_fgetchar()`** block and wait for one character from the keyboard. During the wait, **`WndProc()`** processes a **`WM_CHAR`** message to obtain a character typed. A p4 lock ensures that multitasking continues while the MPI process blocks. Once a character is received, **`WndProc()`** unlocks the lock to indicate the delivery. This mechanism accepts only one character at a time, is non-buffered, and may lose typed characters but proved sufficient.

`fgets()` continues calling **`_fgetchar()`** and storing in a buffer until a newline character is received. **`Scanf()`** calls **`fgets()`** first to read a string in a buffer. Since

no function of the **`scanf()`** family accepts a **`va_list`** parameter, **`sscanf()`** is used resulting in non-portable source code.

This chapter discussed the basic features of the Windows programming approach. In many regards, programming for Windows is different from UNIX. Even basic functionality like keyboard input and screen output and the cooperation between application program and operating system kernel use dissimilar approaches. Therefore, finding solutions to all problems was the main condition for the feasibility of the WinMPI implementation. As this chapter showed, it is possible to provide the UNIX functionality assumed by MPI applications with MS-Windows means.

4. Running MPI-Programs under Microsoft Windows 3.1

4.1. Porting MPI Programs from UNIX to MS-Windows 3.1

This section shows the procedure of porting MPI programs written under UNIX to WinMPI. First the steps are briefly discussed and illustrated. The second part elaborates on some of the problems and pitfalls faced by the programmer.

The following steps can be performed routinely, with a comfortable text editor speeding up the process considerably:

Step 1. Replace the line

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

with

```
int MPI_main (int argc, LPPSTR argv)
```

LPPSTR is a far pointer to a far pointer to char, the data type for passing the array of command line strings.

Step 2. Change all occurrences of **`int`** to **`Int`**. **`Int`** is defined as long. All integers in the program are 32-bit values.

Step 3. For all pointers referencing far data objects, change to definitions and declarations to far pointers.

Step 4. Check the **`printf()`** functions: Since all integer variables are now 32-bit values, they require a "**`%ld`**" in the format string. If inappropriate format specifications are used, incorrect values may be printed or, in case of pointers, protection faults may be caused!

Step 5. Estimate the amount of memory required by the application. The medium memory model allows for one data segment only, i.e. 64 Kbytes minus stack size

minus global MPI variables are available, right now approximately 42 Kbytes. If the application uses larger global or automatic (local) data structures, allocate this memory from the global Windows heap using **`p4_shmalloc()`**.[15]

Step 6. Although Windows permits the allocation of memory blocks greater than 64Kbytes, many functions do not work with larger blocks, for example the **`memcpy()`** family. The same restriction applies to all MPI-send/receive functions. A solution is breaking a large single message into multiple small messages.

Step 7. Estimate the running time of the MPI application. As long as the program does not call an MPI function that executes a blocking receive, a synchronization, or a global operation, it is running without interruption, blocking all other Windows processes. For smooth execution, insert calls to **`MPI_Win_yield()`** in program fragments that could require a considerable execution time. On the other hand, too many calls (for example in each iteration of a tight loop) may result in many task switches and a considerable overhead.

Step 8. Compile the program. Take compiler warnings seriously

```
warning C4759: segment lost in conversion
warning C4762: near/far mismatch in argument : conversion supplied
```

Both warnings signal collisions between near and far pointers. The first warning should not be ignored! (exception: command line arguments) For example, copying data into dynamically allocated memory blocks cannot be handled by **`memcpy()`** because it uses only near pointers. Instead, the (Windows-specific) **`_fmemcpy()`** function can be used. More examples exist where a closer understanding of Windows' internals is required to succeed in porting an MPI application. The problems may be alleviated in later WinMPI versions to a certain extent, but never completely eliminated. MS-Windows 3.1 is still a 16-bit operating system.

---

[15] Restriction may be lifted later if large memory model with single data segment is usable.

The seven rules given above are applied to the MPI program for the computation of π shown in Figure 5. Figure 16 shows the modified program. Step 5 may deserve closer consideration: The program does not call **MPI_Win_yield()**. Hence, the program does not relinquish the processor while executing **compute_interval()**. For 100,000 intervals and more, the execution time is in the range of seconds. On the other hand, simply invoking **MPI_Win_yield()** once per iteration results in 100,000 calls and significant overhead. Breaking the loop into two nested loops and invoking **MPI_Win_yield()** about each 10,000 iterations is an alternative.

```
/*****************************************************************
*                                                                *
*  pimpi.c                                                       *
*  Calculation of PI (numerical integration method)              *
*  MPI for MS Windows 3.1 version                                *
*  Joerg Meyer       University of Nebraska at Omaha             *
*  11/16/94          Computer Science Department                 *
*                                                                *
*****************************************************************/

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

/*****************************************************************
*                                                                *
*  compute_interval - numerical integration over                 *
*                     part of interval                           *
*                                                                *
*****************************************************************/
double compute_interval (Int myrank, Int ntasks, long intervals)
{
    double width, x, localsum;
    long j;

    width = 1.0 / intervals;  /* width of single stripe */
    localsum = 0.0;
    for (j = myrank; j < intervals; j += ntasks)
    {
        x = (j + 0.5) * width;
        localsum += 4 / (1 + x * x);
    }
    return (localsum * width);      /* size of area */
}

int MPI_main (int argc, LPPSTR argv)
{
    long intervals;
    Int myrank, ranksize;
    double pi, di;
    Int i;
    MPI_Status status;
```

```
    MPI_Init (&argc, &argv);       /* initialize MPI system */
     MPI_Comm_rank (MPI_COMM_WORLD, &myrank);     /* my place in MPI system
*/
    MPI_Comm_size (MPI_COMM_WORLD, &ranksize);  /* size ofMPI system */

    if (myrank == 0)               /* I am the parent/master */
    {
        printf ("Calculation of PI by numerical Integration\n");
        intervals = atoi (argv[1]);
        printf ("%ld intervals used\n", intervals);
    }
     MPI_Barrier (MPI_COMM_WORLD);  /* make sure all MPI tasks are running
*/
    if (myrank == 0)               /* I am the parent/master */
    {
/* distribute parameter */
        for (i = 1; i < ranksize; i++)
            MPI_Send (&intervals, 1, MPI_LONG, i, 98, MPI_COMM_WORLD);
    }
    else  /* I am a child/slave */
/* receive parameters */
        MPI_Recv (&intervals, 1, MPI_LONG, 0, 98, MPI_COMM_WORLD, &status);

/* compute my portion of interval */
    pi = compute_interval (myrank, ranksize, intervals);

    if (myrank == 0)   /* I am the parent/master */
/* collect results, add up, and print results */
    {
        for (i = 1; i < ranksize; i++)
        {
            MPI_Recv (&di, 1, MPI_DOUBLE, i, 99, MPI_COMM_WORLD, &status);
            pi += di;
        }
        printf ("Pi estimation: %14.12lf\n", pi);
        printf ("%d tasks used\n", ranksize);
    }
    else  /* I am a child/slave */
/* send my result back to parent/master */
        MPI_Send (&pi, 1, MPI_DOUBLE, 0, 99, MPI_COMM_WORLD);

    MPI_Finalize ();
    return 0;
}
```

Figure 16 WinMPI example program - Computation of π

In general, porting an MPI application to WinMPI follows a relatively simple procedure. It will never be necessary to rewrite larger portions of the program. Programmers with some Windows experience should always be able to avoid pitfalls and resolve all problems.

4.2. Start of a WinMPI application

As discussed in Section 2.2, MPI does not include a standard start-up procedure or rules for the MPI process structure definition. Both issues are left to the underlying communication subsystem. With WinMPI adopting p4, the p4 start-up procedure applies to WinMPI applications.

To start a WinMPI application, p4 requires a configuration file. Refer to [Butler92] for more information on p4 configuration files. The UNIX version for a configuration file for a master-slave application (named **`demo.p4g`**) may look like:

```
local 0
thor.unomaha.edu 2 /u1/jmeyer1/mpiappl/demoslave
csalpha.unomaha.edu 2 /home/jmeyer1/mpiappl/demoslave
```

In general, each line contains the name of a site, the number of processes to start, and the path to the executable file. Only slight changes are necessary to use this file for WinMPI. The site is ignores, only field 2 and 3 are relevant. The path name must match the location of the executable under Windows; however, both UNIX-style paths with "**/**" and MS-DOS-style path name with "****" are accepted. The Windows version of **`demo.p4g`** could look like :

```
local 0
thor.unomaha.edu 2 /winmpi/mpiappl/demslave
csalpha.unomaha.edu 2 \home\jmeyer1\mpiappl\demslave
```

The MPI application is started like any other Windows application, for example from the Program Manager or the File Manager. Besides application-specific command-line arguments, all useful p4 options can be given. The master-slave application from above is started with the command line:

```
demaster 10 20 30 -p4pg demo.p4g
```

The procedure used is identical to the startup for MPICH from June 17, 1994. The latest MPICH version provides a tool **`mpirun`** that hides the p4 peculiarities from the user. It could be supplied in a later version of WinMPI.

In the following, a demonstration of running the WinMPI program from Section 4.1 is given. The p4 configuration file **`mpi_is15.p4g`** contains only one line:

```
local 15
```

and will create sixteen MPI processes that cooperate in computing π by numerical integration. Figure 17 illustrates how to start the MPI application from the File Manager.

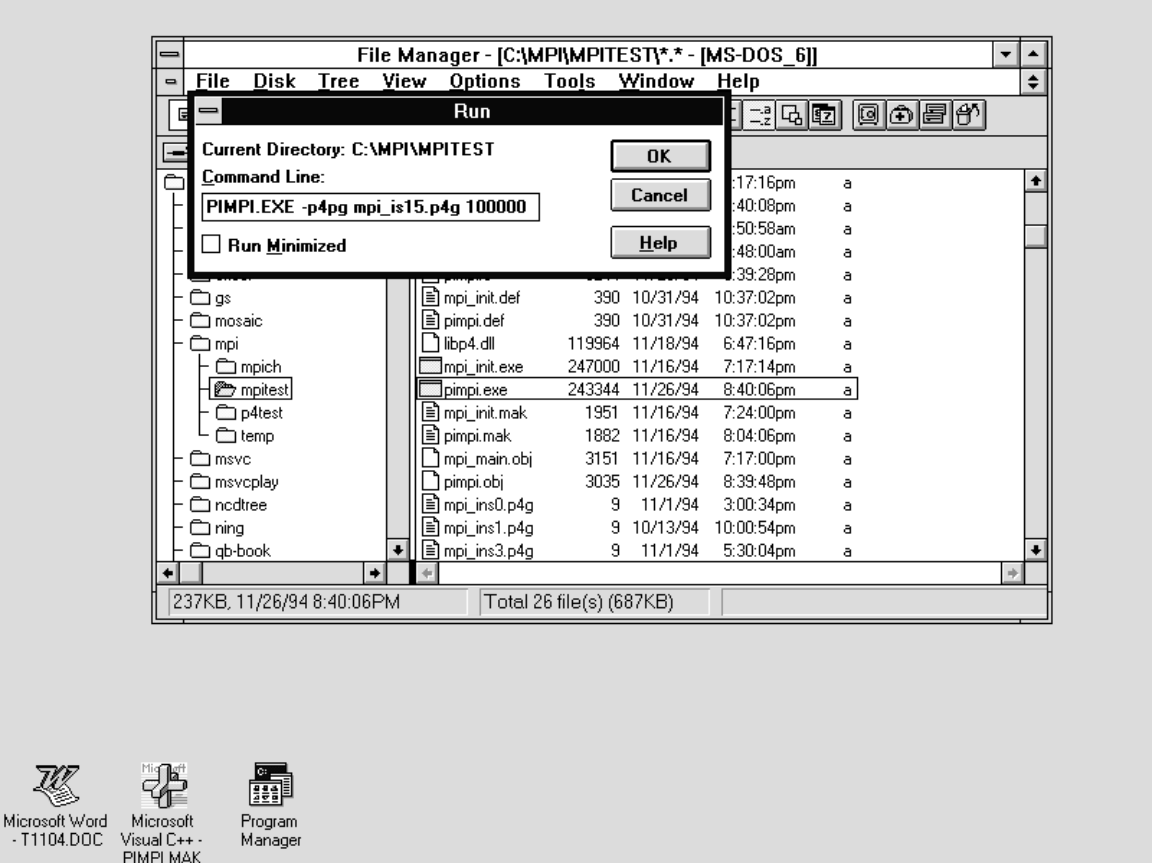


Figure 17 Start of WinMPI application

After **`MPI_main()`** returns, all MPI processes enter the message loop in **`WinMain()`**. On the screen, all slave processes are visible as icons while the master process shows the last output of the WinMPI application. For the example program, Figure 18 shows this final state. The whole application can be terminated like any other Windows application. Closing any slave process kills all slave processes; closing the master process terminates the entire WinMPI application.

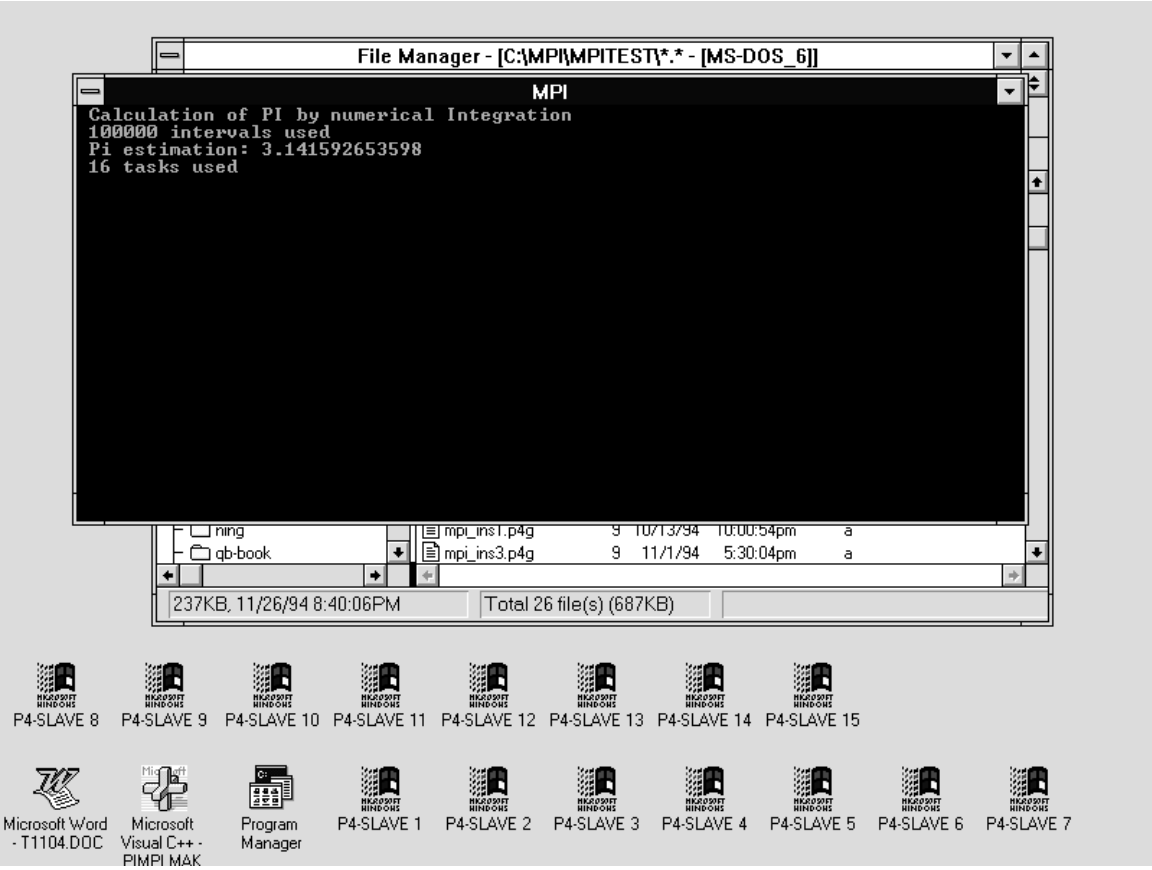


Figure 18 WinMPI application after execution of **`MPI_main()`** function

4.3. Porting MPI Programs from MS-Windows 3.1 to UNIX

Once an MPI application is written and debugged under WinMPI, one might want to move it to a parallel or distributed system running MPICH and examine its scalability. Since all restrictions imposed by WinMPI are lifted and no new limitations apply, this port is straight-forward and should not face any problems. There are five steps to be performed:

Step 1. Replace the line

```
int MPI_main (int argc, LPPSTR argv)
```

with

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

Step 2. All Int variables have to be 32-bit integers in UNIX. There are three equivalent option:

1. Replace Int with int.
2. Replace Int with long.
3. At beginning of program insert the line

```
#define Int int
```

Step 3. Remove all far and near declarations and definitions.

Step 4. Remove all invocations of **`MPI_Win_yield()`**

Step 5. Replace WinMPI-specific function calls

`p4_shmalloc().`	⇒	**`malloc()`**
`_fmemcpy()`	⇒	**`memcpy()`**
	`etc.`	

The simplicity of these five steps proves that the WinMPI design is appropriate. Despite the great differences between the Windows and UNIX programming environment, it is

possible to port programs from UNIX to Windows and vice versa with little effort. WinMPI can be considered almost 100% source-code portability to MPI programs.

5. Concluding Remarks

MPI will be the parallel programming standard for the future. It is designed for writing message-passing programs that are portable to all existing parallel architectures. Current MPI implementations are based on UNIX as operating system and are available for most parallel computers and distributed systems.

The idea behind WinMPI, an MPI implementation for MS-Windows 3.1, is to make MPI available to the large community of Windows users that do not have access to UNIX or are not familiar with this operating system. WinMPI fills a gap left by the currently available MPI implementations. Its goals are promoting the acceptance of MPI and improving the implementation process of MPI applications.

Both goals can be achieved with an WinMPI implementation that utilizes only a single IBM-compatible personal computer (PC). MS-Windows is a multitasking operating system and permits multiple concurrent processes. Therefore, WinMPI supports currently only shared memory for the exchange of messages. In the future, it can be enhanced to utilize a distributed system of PC's connected by a local area network.

MPICH, an existing MPI implementation, provided the basis for WinMPI. Due to the great differences between the UNIX and Windows programming approach, especially the p4 layer, the lower layer of MPICH, is completely re-structured. Numerous UNIX features are replaced with Windows functionality. Employing a DLL permits communication via shared memory and offers optimal memory utilization. The p4-DLL is extensively tested and is considered to be a reliable communication subsystem. p4 test programs taken from the UNIX distribution achieve remarkable throughput even in stress-test situations.

Porting the upper MPICH layer required relatively little effort. While the most difficult procedure for p4, the coding and debugging process of the upper layer, the MPI layer, proved to be straightforward. However, due to the number of files, is was still time-

consuming. The planning phase consisted of careful devising and numerous revisions but grew into a concept that achieved instant success. The time for testing and debugging was kept to a minimum. All test were successful, suggesting that the optimal porting approach was adopted.

The results suggest that acceptable solutions to all conceptual problems could be found. The WinMPI allows writing parallel programs which are portable to other (UNIX) MPI systems with minimum effort.

The author is convinced that the software made available to the public will promote the acceptance of MPI. WinMPI as a development tool helps decrease implementation cost for MPI applications by relying on less expensive hardware. As a learning tool, it is available to a growing audience of people that feel attracted or challenged by the virtues of parallel computing. WinMPI can play an important role for years to come.

There has been contacts with members of the MPICH implementation team, who showed interest in WinMPI and its goals [Lusk94]. Currently, the costs of merging MPICH and WinMPI are evaluated and there is a chance that WinMPI might become available as part of the MPICH distribution.

WinMPI is available in a form that permits writing and executing MPI applications under MS-Windows 3.1. The interface for the programming language C supports the entire standard. However, other parts of MPICH have not been ported yet.

Fortran interface. Besides supporting the C language, MPICH provides a Fortran 77 interface since Fortran is widely used for parallel applications. This interface is implemented by wrapper functions that simply call the matching C functions. Due to lack of time and the unavailability of a Fortran compiler for Windows, the Fortran interface is not included in WinMPI yet.

Profiling interface. Since information about the runtime behavior of a parallel application is extremely important but difficult to collect with existing tools, MPI defines a profiling

interface. This interface is currently disabled but not removed from the source code. It can be added with little effort.

MPICH version update. WinMPI was ported from MPICH dated June 17, 1994. The latest MPICH version stems from November 3, 1994, and is considerably improved with regards to bug-fixes and programming environment. Hence, incorporating this new version into WinMPI is desirable. As a final goal, both packages should be merged and maintained together. Hopefully, a way can be found to cooperate with the MPICH implementors' team.

C compiler. Currently, Microsoft Visual C++ 1.5 is required for writing WinMPI programs. Support for other compilers such as Borland is desirable. The cost of supporting other compilers has to be evaluated and compared to their acceptance among Windows programmers.

WinMPI programming environment. During the last half year, MPICH has improved its programming environment considerably. For instance, it provides a portable startup mechanism that hides the p4 peculiarities from the user. These and other features are desirable for WinMPI as well to make it more user-friendly.

Further work should probably be continued in the order given above.

6. Bibliography

[Bomans90] L. Bomans, R. Hempel. "The Argonne/GMD Macros in Fortran for Portable Parallel Programming and Their Implementation on the Intel iPSC/2." *Parallel Computing*, 15:119-132, 1990.

[Bridges94] P. Bridges, et.al. "User's Guide to mpich, a Portable Implementation of MPI." Available by ftp from info.mcs.anl.gov in /pub/mpi/guide.ps.Z, November 1994.

[Butler92] R. Butler, E. Lusk. "User's Guide to the p4 Programming System." Technical report TM-ANL-92/17, Argonne National Laboratory, 1992.

[Conger92] J. L. Conger. The Waite Group's API bible : the Definitive Programmer's Reference. Waite Group Press, Corte Madera, CA, 1992.

[Dongarra93a] J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. "A Proposal for a User-level, Message Passing Interface in a Distributed Memory Environment." Technical Report TM-12231, Oak Ridge National Laboratory, February 1993.

[Dongarra93b] J. Dongarra. et. al. "Integrated PVM Framework Supports Heterogenous Network Computing." *Computers in Physics*, 7(2):166-75, April 1993.

[Flynn66] M. J. Flynn. "Very High Speed Computing Systems," *Proc. IEEE,* 54, pages 1901-1909, 1966.

[Geist90] G.A. Geist, et.al. "A User's Guide to PICL: a Portable Instrumented Communication Library." Technical Report TM-11616, Oak Ridge National Laboratory, October 1990.

[Geist94] G. A. Geist, et al. "PVM 3 User's Guide and Reference Manual." Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1994.

[Gropp94] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, Cambridge, 1994.

[Hatcher91] P. J. Hatcher, M. J. Quinn. *Data-parallel Programming in MIMD computers*. MIT Press, Cambridge, MA, 1991.

[Lewis92] T. Lewis, H. El-Rewini. *Introduction to Parallel Computing*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[Lusk94] E.Lusk. E-mail sent to the author. November 18, 1994.

[Meek78] B. Meek. *Fortran, PL/1, and the Algols.* Macmillan Press, London, 1978.

[Microsoft94] *INF: Using Large Memory Model, Microsoft C/C++, & Windows 3.1*, Document Q90294 available from gopher.microsoft.edu, October 1994

[MPIF94a] Message Passing Interface Forum. "MPI: A Message-passing Interface Standard." Technical Report CS-94-230, Computer Science Dept., University of Tennessee, Knoxville, TN, 1994.

[MPIF94b] Message Passing Interface Forum. MPI: "A Message-passing Interface Standard." *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[Nelson91] R. P. Nelson. *The 80386/80486 Programming Guide*. Microsoft Press, Redmond, WA, 1991

[Niezgoda94] S. Niezgoda. "Charting the Uncharted." *BYTE*, October 1994: 203, 204.

[Parasoft92] Parasoft Corporation, Pasadena, CA. *Express User's Guide*, version 3.2.5 edition, 1992.

[Petzold92] C. Petzold. *Programming Windows: the Microsoft Guide to Writing Applications for Windows 3.1*. Microsoft Press, Redmond, WA, 1992.

[Pierce88] P. Pierce. "The NX/2 Operating System." *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 384-390. ACM Press, 1988.

[Singhal94] M. Singhal, N. Shivaratri. *Advanced concepts in operating systems: distributed, database, and multiprocessor operating systems,"* McGraw-Hill, New York, 1994.

[Walker92] D. Walker. "Standards for Message Passing in a Distributed Memory Environment." Technical Report TM-12147, Oak Ridge National Laboratory, August 1992.