Parallel Programming on Linux Networks Using MPI & PVM

Graeme S. McHale

Contents

Chapter 1

Message Passing Interface

1.1 Introduction

Message-Passing Interface (MPI) is a library of functions which assist programmers in writing applications programs for parallel systems. These functions can be called from standard Fortran, C or C++[1]. The core of this library is the set of functions which implement *message passing*. These assist in achieving parallelism by allowing data to be transmitted from one process[2] to another. Message passing is extremely powerful, is it very "low level" thus the programmer has to handle much of the detail.

When comparing MPI to other available schemes, [1] gives the following as its major advantageous features:

- It is expected to be faster in a large multiprocessor.
- It has a large set of point-to-point communication functions.
- It has a large set of collective communications functions for communicating amongst groups of processes.
- It has the ability to specify communication topologies[3].
- It has the ability to create derived datatypes that describe messages of non-contiguous data. See section 6.

1.2 MPI Program Structure

As mentioned before MPI is just a set of functions which can be called from standard C. In order to gain access to these functions any MPI program must include:

[1] The C++ libraries shall not be discussed in this document.

[2] For real parallisation, it would be normal for each process to be running on a seperate processor. MPI does not assume this however.

[3] not disscussed in this document, see [2] and [3] for details.

```
#include ‘‘mpi.h’’
```

All MPI programs, before calling any other MPI function must call `MPI_Init()`, which normally takes any arguments passed to the program as it arguments.

To facilitate an SPMD[4] program, processes must be able to identify themselves and the other processes they are working with. In an MPI system, each process is identified using a unique integer termed its *rank*. Processes can determine their rank by calling `MPI_Comm_rank()`. It's syntax is discussed in section 1.5.1. The number of processes available to the system can be determined using `MPI_Comm_size`() discussed in section 1.5.2.

All MPI programs must include, as their last MPI function call, a call to `MPI_Finalize()`.

1.3 Sending & Receiving Using MPI

The most basic message passing functions in MPI are `MPI_Send()`, used for sending messages and `MPI_Recv()`, used for receiving messages.

Any message which is sent, either in MPI or in the outside world must contain information other than the actual contents of the message. Obviously, the address to which the message must be sent should be included but other details are needed. The first of these is the size of the message so that its end can be determined. In order to send a message in MPI you must provide the location of the data you wish to send, it's size and it's type, all of these ensuring that the message is physically received correctly. The possible types of message which can be sent and received using MPI functions are discussed in section 1.4.

In order that different types of message can be determined, MPI adds another integer variable to represent the *tag* or type of a message. This is completely determined by the programmer and is only used to ensure that messages should not be interpreted incorrectly. For example, it is not always guaranteed that two messages sent between the same processes will be received in the same order as they were sent, so if you wished to send two values to another process, one a product the other an average you could set the tag of the first as 0 and the second as 1 so that you can distinguish between the two easily at the other end.

The last piece of information required in order to send or receive information using MPI concerns the processes that are involved. In MPI processes are grouped into collections called *communicators*. For two processes to communicate they must belong to the same communicator. The standard communicator is `MPI_COMM_WORLD` which every process belongs to. It is possible to create user defined communicators but this is beyond the scope of this text.

Taking the above into account, the syntax for MPI's basic send function is as follows:

```
int MPI_Send(
                void*           message      /* in */,
```

[4]The SPMD programming model is discussed in section 1.7.3.

```
            int            count        /* in */,
            MPI_Datatype   datatype     /* in */,
            int            dest         /* in */,
            int            tag          /* in */,
            MPI_Comm       comm         /* in */)
```

The syntax of the basic receive function, `MPI_Recv()` is given below. Most of the arguments have already been discussed. The last argument of type `MPI_Status` is a struct with the following definition:

```
struct MPI_Status {
   int  MPI_SOURCE;
   int  MPI_TAG;
   int  MPI_ERROR;
};
```

. The last of these members is an error code. The first two are used to determine values if the function was called using wildcards.

```
int MPI_Recv(
            void*          message      /* out */,
            int            count        /* in  */,
            MPI_Datatype   datatype     /* in  */,
            int            source       /* in  */,
            int            tag          /* in  */,
            MPI_Comm       comm         /* in  */,
            MPI_Status     status       /* out */)
```

Wildcards

When calling `MPI_Recv()` we need not always give exact values for all arguments. If you wish to receive a message from any process in a given communicator then the wildcard `MPI_ANY_SOURCE` can be passed as the source argument to the function.

If you wish to receive a message of any tag then the wildcard `MPI_ANY_TAG` can be passed to the function.

Blocking & Non-Blocking Communication

`MPI_Recv()` and `MPI_Send()` are *blocking* communication functions. In the case of `MPI_Recv()` this means that if it is called by a process A, if there is no message available then A will block (sit idle) until the message becomes available. `MPI_Send()` behaves in a similar fashion. Non-blocking versions of both functions are available. See [2] for details.

1.4 Types in MPI

1.4.1 Basic Types

Throughout the descriptions of the communication functions given elsewhere in the document reference has been made to `MPI_Datatype` variables. The MPI functions can only send data that they know the type of. The type of data being sent or received is indicated by a variable of type `MPI_Datatype`. This variable can take one of several values outlined in the table below. In general however, for every built-in C datatype, there is a corresponding MPI type.

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_INT	signed int
MPI_LONG	signed long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_SHORT	signed short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_SHORT	unsigned short int
MPI_BYTE	
MPI_PACKED	

These definition handle the transmission of single values of C built-in types adequately. If it is required that an array of values of these types be transmitted, then the count argument of the message-passing function should indicate the size of the array.

1.4.2 Derived Types

Suppose it was required that a struct be sent from one process to another. If only the builtin types could be used, each component of the struct would need to be sent as a separate message. Sending messages is very computationally expensive. In order to minimise the number of messages which need to passed, MPI has several methods of allowing the building of user defined types in order to send as many different items in a single message as required.

The most general of these involves using the `MPI_Type_struct` function to build a derived datatype. This function requires the following information:

1. The number of components in the struct [5].

2. The size (in blocks) of each element.

[5] It should be noted that the struct is just used as an example. You could equally use this function for sending a single column of a multidimensional array for example.

3. The displacement of each element. That is how far in memory they are from the beginning of the message.

4. The type of each element.

To obtain the displacements, the function `MPI_Address()` should be used to obtain the address of the start of the message and then again to get the address of each component, the two should be subtracted. The syntax of `MPI_Address()` is as follows:

```
MPI_Address(
            void*     location   /* in  */,
            MPI_Aint* address    /* out */)
```

The syntax of `MPI_Type_struct` is as follows:

```
int MPI_Type_struct(
            int           count            /* in  */,
            int           block_lengths[]  /* in  */,
            MPI_Aint      displacements[]  /* in  */,
            MPI_Datatype  typelist[]       /* in  */,
            MPI_Datatype* new_type         /* out */)
```

It should be noted that the type list can take the form of any builtin MPI type but also any other derived type. It is therefore possible to nest derived types.

After `MPI_Type_struct` has been called, the type created should be passed to `MPI_Type_commit()` for finalising. A good example of the use of this type of message building is given in our example program in 1.7. Other methods of building messages and user defined types are available see [2] and [3].

1.5 Commonly Used Functions

In the following sections, basic explanations and definitions will be given for some of the most useful functions within MPI. A fuller explanation of these functions together with definitions of many other functions can be found in [2] and [3].

Function return values are error codes unless otherwise stated.

1.5.1 `MPI_Comm_rank()`

The definition of `MPI_Comm_rank()` is as follows :

```
int MPI_Comm_rank(
                  MPI_Comm     comm   /* in  */,
                  int*         rank   /* out */)
```

It gives the rank of the calling process within the given communicator `comm`.

1.5.2 MPI_Comm_size()

This function is used to determine the number of processes in a given communicator. It's definition is as follows :

```
int MPI_Comm_size(
                  MPI_Comm     comm   /* in  */,
                  int*         size   /* out */)
```

1.5.3 MPI_Bcast()

This function is used to transmit a given message from a `root` process to all processes in a given communicator. The definition of `MPI_Bcast` is as follows :

```
int MPI_Bcast(
              void*         buffer    /* in/out */,
              int           count     /* in     */,
              MPI_Datatype  datatype  /* in     */,
              int           root      /* in     */,
              MPI_Comm      comm      /* in     */)
```

As with the other basic communication functions (1.3), `buffer` specifies the location of the message to be sent, `count` specifies it's size and `comm` specifies the processes involved in the communication. The function is collective.

1.5.4 MPI_Scatter()

The definition of `MPI_Scatter` is as follows :

```
int MPI_Scatter(
                void*         sendbuf    /* in  */,
                int           sendcount  /* in  */,
                MPI_Datatype  sendtype   /* in  */,
                void*         recvbuf    /* out */,
                int           recvcount  /* in  */,
                MPI_Datatype  recvtype   /* in  */,
                int           root       /* in  */,
                MPI_Comm      comm       /* in  */)
```

This function is used to distribute the message located at `sendbuf` with the `root` process, amongst all processes in communicator `comm`. The message is split into segments, one for each process, with segment 0 being sent to process 0 etc. It is commonly used to send a portion of an array to each process. `sendcount` specifies the size of the portion which should be sent to each process and `sendtype` specifies the type of the data being distributed. Data is stored at recvbuf with the receiving processes. While in most cases `recvcount` and `recvtype` are the same as `sendcount` and `sendtype`, this need not be the case. Although the function is collective, only `root` and `comm` must be the same on each process.

1.5.5 MPI_Gather()

MPI_Gather can be thought of as the opposite of MPI_SCATTER. It's definition is as follows :

```
int MPI_Gather(
              void*         sendbuf    /* in  */,
              int           sendcount  /* in  */,
              MPI_Datatype  sendtype   /* in  */,
              void*         revcbuf    /* out */,
              int           recvcount  /* in  */,
              MPI_Datatype  recvtype   /* in  */,
              int           root       /* in  */,
              MPI_Comm      comm       /* in  */)
```

The functions collects together the data referenced by sendbuf from each process in comm and stores in the memory referenced by recvbuf on the root process in process rank order. sendcount, sendtype, recvcount, and recvtype all have the same meaning as in 1.5.4.

1.5.6 MPI_Reduce()

This function gives the result of applying a given operator to a value stored on each process. For example, if each process held a value A, this function could be used to find the minimum value of A across all processes.

The definition of MPI_Reduce() is as follows :

```
int MPI_Reduce(
              void*         operand    /* in  */,
              void*         result     /* out */,
              int           count      /* in  */,
              MPI_Datatype  datatype   /* in  */,
              MPI_Op        operator   /* in  */,
              int           root       /* in  */,
              MPI_Comm      comm       /* in  */)
```

The argument operand specifies the address of the operand value on each process. result specifies the location of the value that should hold the result on the root process. comm and datatype have their usual meanings.

operator specifies the operation you wish to be applied to the values. This can take a number of values (both builtin and user specified). The most useful of the builtin candidates are shown in the following table.

Operation	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_LOR	Logical or

Further details on other alternatives, builtin and otherwise can be found in [2] and [3].

1.5.7 `MPI_Barrier()`

This function is used to synchronise all processes in a given communicator. Any process entering it is blocked until all processes in the communicator have entered it. It's definition is as follows :

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

1.6 Compiling & Running MPI Programs

1.6.1 Before You Begin

MPI distributions are available for many different platforms and most platforms have several different distributions available. Currently within the department *MPICH* is used. MPI is pre-installed and configured ready for use within the department. It is however useful to consider how this may be achieved using a arbitrary network of workstations and is therefore discussed in Appendix A.1.

All example MPI code given in this document has been developed using MPICH 1.2.0. This is currently located at:

```
/net/dazdak/mpich-1.2.0/
```

This directory will be referred to as `MPIHOME`. Before you can compile MPI programs or run them using the scripts provided, you should append `MPIHOME/bin` to your `PATH` environment variable.

1.6.2 Compiling

There is a pre-written shell script to compile simple C programs which contain calls to MPI functions. `mpicc` is located in `MPIHOME/bin` and can be used much like a standard C compiler. For example

```
mpicc -c coins.c1
```

will create the appropriate object file and

```
mpicc -o coins coins.o
```

will link the appropriate files to create an executable. It is not advisable to compile and link in the same command.

It is also possible to compile programs without using this script providing the appropriate libraries and include files are made known to the compiler. For example:

```
gcc -c coins.c -I/net/dazdak/mpich/include
gcc -o coins coins.o -L/net/dazdak/mpich-1.2.0/lib -lmpich
```

Further information on `mpicc` can be found in the MPICH user manual [4].

1.6.3 Running

Before programs can be run you must be sure of which machines the processes that are created will be sent to run. As default processes are run on a given number of machines who's host names are listed in:

```
MPIHOME/util/machine/machines.LINUX
```

This usually holds the host names of all possible hosts available and before running any code you should use the `tstmachines` script to check which of these machines is currently available. If there are problems with any of the machines listed in the standard machines file it will output a list of those which are available, if not then there is no output. It is often useful to create your own file of hostnames, which can be fed both to the MPI run scripts and to `tstmachines` for checking.

`tstmachines` is a simple shell script which can be easily (copied and) adapted to generate output as required for specific tasks. Such a scheme was used when the programs in this document were tested, so that a machine file could be generated automatically. `tstmachines` is located in `MPIHOME/bin`.

Programs are best run using the `mpirun` script located in `MPIHOME/bin`. A typical command would be as follows:

```
mpirun -np 16 -machinefile machines.ok coins
```

This would run the executable `coins` on the first 16 machines specified in `machines.ok`. It should be noted that input and output from the program can be redirected in the normal fashion. Further information on `mpirun` can be found in the MPICH user manual [4].

1.7 Extended Example Using MPI

In order to demonstrate the features of MPI, a simple (yet computationally substantial) sequential program will be given and then it will be parallelised using MPI. The example chosen is a genetic algorithm which works out the combination of coins required to give a certain amount of money within certain bounds.

1.7.1 The Sequential Algorithm

The basic outline of a genetic algorithm is as follows: A set of possible solutions to a problem is generated at random. The *fitness* of each candidate solution (ie how well its matches the required criteria for solving the problem) is then calculated. A predefined number of the best candidate solutions are then kept, while the others are discarded. The fittest set go on to make up the next *generation*, some as they are, some *crossed* with other candidate solutions and some *mutated* or changed in a random fashion.

The fitness of each candidate is then recalculated and the processes of *reproduction* (i.e. selecting the best, crossover and mutation) begins again until a candidate is found that matches the requirements exactly (has the required fitness).

In the coin counting example, the candidates solutions are possible numbers of each coin type, the fitness of each candidate is a measure of how closely it matches the required sum of money and whether the total number of coins it uses is within the given bounds. Crossover of candidates will involve taking the numbers of a given coin from two candidates at random and swapping them over. Mutating a candidate will involve inserting a random number of a randomly chosen coin type.

A sequential coding of this algorithm in C is given in Appendix B. The code for auxiliary functions which carry out reproduction, fitness calculation, etc as well as definitions is given in Appendix E.

1.7.2 Parallelising the Algorithm

The performance of this code is largely determined by two dominant sections; the calculation of the fitness of each candidate and the sort of the candidates based on their fitness. Both these operations involve iterations through the whole candidate set (a large array). Given that as a general rule of thumb; the bigger the candidate set in a genetic algorithm, the more effective it is, both these operations are potentially time consuming. If the performance of this application is to be improved through parallelisation, then these tasks would seem suitable candidates for distribution amongst different processes.

These tasks are well suited to having a number of worker processes sent parts of the candidate set to perform operations on and then sending their results back to a master process.

Therefore the processing of this part of the algorithm will now be achieved as follows:

1. The candidate set will be divided amongst the worker processes for fitness calculation.

2. Each local set of candidates will be sorted by each process.

3. The master processor will merge these results back together before going on to produce the next generation if required.

Parallelisation of the code will proceed using an *SPMD* model. Single Program, Multiple Data means that all processes are running an identical piece of code. However their position within that code will vary as required. In this model it is common to have each process branch in the code according to its position within the parallel system. In our case, in the main part of the program, there will be a branch point where the master follows one set of instructions and the workers follow another. This branching should be achieved in such a way that the master can perform some of the work on the candidate set so it is not idle while waiting for the results from the other processes.

1.7.3 Implementation

Before processing can commence, certain control values are required from the user. This will also be the case in the parallel version, however input and output in parallel systems can be troublesome so care must be taken. It will be assumed that only the master should be responsible for input and output, so values must therefore be requested from the user and sent to all other processes. This type of communication, where values are sent from one process to all others, is termed a broadcast and can be simply achieved using the `MPI_Bcast()` function *(1.5.3)*.

As an example; the sum of money required should be available to all processes. Assuming it is stored in the variable `sum`, this value of this variable can be broadcast among the other processes using the following code:

```
MPI_Bcast(&sum, 1, MPI_LONG, 0,MPI_COMM_WORLD);
```

`MPI_Bcast()` is an example of a *collective function*. These must be called by every process, in a given group, in order to operate correctly. Any process calling this function will block until all other processes have called it (usually with the same parameters).

The first argument here is the address of the variable from which the value should be taken from on the source process and where it should be placed on the destination processes. The second argument in the length of value. It is 1 here as this a single long integer. Afterwards the type of the variable has been specified. The types used in MPI are discussed in 1.4. The next argument is the rank of the source process. Finally the group of processes involved in the broadcast is given, `MPI_COMM_WORLD` is taken to mean all processes.

Handling these control variables is the first point in our program where it is required that the master and the workers be carrying out different tasks. Therefore there shall be a branch in the program conditional on the rank of the process running it; if their rank is 0 they are the master, if not then they must be workers.

Once all the worker processes have been provided with the global control variables, the genetic algorithm itself can be entered. Again the tasks completed by the master and the workers differ with the master required to create an initial set of candidates before starting to calculate their fitness along with the workers. The only task required of the workers from now until termination will be to calculate fitness and sort until the ultimate candidate has been found. Therefore the workers branch of the program will enter a loop where it will process candidates, send back the results and then be told by the master whether another generation of candidates will be required.

The master will also do processing of candidates, then merge all the sorted candidate sets from the workers together and determine whether further computation is needed as well as providing feedback to the user.

Communication of the need for a new generation will also be achieved using a broadcast.

Actual processing of the candidates; determining there fitness and sorting them in order is achieved using a separate function. The first step of processing must be distributing the candidates among the worker processes. As candidate information is held in an array; this distribution is best handled by a *scatter* operation (discussed more fully in 1.5.4). This will split up an array sending a portion to each candidate. Example syntax is as follows:

```
MPI_Scatter(&cand, LOCAL_CANDS, mpi_cand_type,
            local_cand, LOCAL_CANDS, mpi_cand_type,
            0, MPI_COMM_WORLD);
```

Most arguments are self explanatory, the destination and source address and types and the rank of the source process and set of processes involved.

The calculation and sorting of fitness is achieved as in the sequential program. Once this is completed the workers must transmit their results back to the master. The master must accept the results and merge them together. Basic sending and receiving is discussed in 1.3. The syntax of a send command is as follows:

```
MPI_Send(local_cand,LOCAL_CANDS,mpi_cand_type,
         0,1,MPI_COMM_WORLD);
```

With a receive command taking the form:

```
MPI_Recv(&result[i][0],LOCAL_CANDS,mpi_cand_type,
         MPI_ANY_SOURCE,1,MPI_COMM_WORLD,&status);
```

Both before the array is scattered and before the results are sent, a check should be made to ensure that there is more than one process in the MPI system. This will ensure that if the program is run with only one process it will not hang and will run efficiently. Broadcasts can be made safely in systems with only one process. If there is only one process then obviously there will be no need to merge any results and the master will operate on the whole candidate set.

The functions given above were mostly sufficient to parallelise this genetic algorithm. A complexity arises as the ability to transmit a user defined type (the candidate struct) was required. The process used to facilitate this is discussed in section 1.4.2. Explanations of other useful MPI functions are given elsewhere in the document. Source code for the MPI parallel version of the coins program is given in Appendix C. The auxiliary functions and definition are those used in the sequential code and are listed in Appendix E.

Chapter 2

Parallel Virtual Machine

2.1 Introduction

PVM allows a collection of computer systems to be viewed as a single parallel virtual machine. For this virtual machine, programs are written as a set of tasks which access the resources of the virtual machine and communicate with each other through a set of library routines. These routines can be called from Fortran, standard C or C++.

When comparing PVM to other available schemes, [1] gives the following as its major advantageous features:

- It is expected to be better when applications will run over heterogeneous networks. It has good in-operability been hosts of different architectures.
- It allows the development of fault tolerant applications which can easily deal with host or task failures.
- It allows dynamic process control as it is built around the virtual machine model.

This last point is a core difference between PVM and MPI. When running a MPI program process management and distribution is handled by the operating system and cannot be affected by any MPI function. PVM is modeled on a much higher level of abstraction, thus allowing programmers to have control of the creation and deletion of processes. Also this allows for the detection and management of faults which MPI does not.

MPI however contains a much broader set of communication routines and affords the programmer much more control in this respect.

2.2 PVM Program Structure

From the programmer's view point, PVM is a set of library routines. Any program which calls these library functions should include:

```
#include <pvm3.h>
```

As with MPI programs it is important that the processes, or more correctly tasks within PVM should have a unique identifier. In PVM this is termed the task's *tid*. A task's tid is determined by called `pvm_mytid()`. This is used as an initialiser within the PVM system and should be the first PVM function called in any program. It is discussed further in section 2.4.1.

PVM does not rely on the underlying operating system to create and manage the tasks or processes within a parallel application. The programmer has control over these issues. Tasks are created using `pvm_spawn()` (discussed in section 2.4.3). Any task can *spawn* new tasks, on any host if required, allowing any virtual process topology to be modelled.

In an SPMD model (see 1.7.3) there are 2 key types of process; the master and the slave. With PVM the master should spawn the worker processes and thus there must be a way that the master process can determine that it is the master. In PVM tasks have a *parent* i.e. the ask which spawned them. As the master task will be invoked by the operating system, it will not have a parent task. The tid of a task's parent is determined using `pvm_parent()` discussed in section 2.4.2.

Usually PVM programs call `pvm_exit()` as their last function before returning control to the operating system.

2.3 Sending & Receiving Using PVM

Sending Messages

There are three steps involved in sending a message in PVM:

1. A send buffer should be created to hold the contents of the message.

2. The data you wish to be sent should be packed into this send buffer.

3. The contents of the current send buffer are sent to another process.

There are two different types of send in PVM. The first of these is a basic process to process send which is achieved using the `pvm_send()` function. The syntax of this function is as follows:

```
int result = pvm_send(
                      int       tid      /* in */,
                      int       tag      /* in */)
```

PVM allows the user to give different types of message an integer *tag* to make sure they are not confused. `tid` specifies the task identifier of the process which should be sent the message. The return value is an error code which is negative if an error occurred.

The second type of send is a multicast. The involves a message being broadcast to every process in a given set and is discussed in section 2.4.5.

Receiving Messages

Receiving messages in PVM is a two stage process:

1. Receive a message into the active receive buffer.
2. Unpack the contents of the receive buffer to gain access to the contents of the message.

No matter which send function was used to transmit a message, any of the many receive functions can be used at the other end. The most basic of these is `pvm_recv()`, it's syntax is as follows:

```
int bufid = pvm_recv(
                    int      tid      /* in */,
                    int      tag      /* in */)
```

`tid` is the task identifier of the task the message is expected from and `tag` has the meaning described above. The value returned is the identifier of the receive buffer the message was put into.

This function is a blocking receive meaning that once called it will not return until a message with the expected tag has been received from the expected task. A non-blocking receive amongst others is available, see [5] for details.

If `-1` is passed to either argument then it is taken to mean "any".

Buffer Management

Key to both sending and receiving messages is the management of buffers. Before a message can be sent it must be packed into a buffer and before this can happen a send buffer must be created. This is commonly achieved using `pvm_initsend()` discussed in section 2.4.4. This creates a new buffer and makes it *active* discarding the contents of the previously active buffer. When messages are received, a new receive buffer is created and made active discarding the contents of the currently active receive buffer.

The identifiers of the current active send and receive buffers can be set using `pvm_getsbuf()` and `pvm_getrbuf()`. Both return an integer identifier and neither take any arguments.

The currently active send or receive buffer be can set using `pvm_setrbuf()` and `pvm_setsbuf()` which take the required buffer identifier as an argument. Passing `0` to either of these functions saves the currently active buffer and makes no buffer active.

Packing & Unpacking

Packing is the process of placing a data item onto the currently active send buffer and unpacking the process of retrieving a data item from the currently active receive buffer.

Functions which pack data take the form :

```
int result = pvm_pk<code>(
                           <type>      *value     /* in */,
                           int         nitems     /* in */,
                           int         stride     /* in */)
```

Functions which unpack data take the following form:

```
int result = pvm_upk<code>(
                            <type>      *value     /* out */,
                            int         nitems     /* in  */,
                            int         stride     /* in  */)
```

Where `<type>` and `<code>` take the following values amongst others, depending on the C type of data you are working with.

`<code>`	`<type>`
`int`	`int`
`float`	`float`
`double`	`double`
`byte`	`char`
`short`	`short`
`uint`	`unsigned int`

As an example, the syntax for `pvm_pkint()`, the function which packs integers is as follows:

```
int result = pvm_pkint(
                         int      *value      /* in */,
                         int      nitems      /* in */,
                         int      stride      /* in /)
```

The `value` parameter is the address of the data item you wish to pack/unpack. Both types of function are capable of handling arrays of the specified type so the `nitems` parameter specifies the number of items that should be processed and `stride` specifies their distance apart.

2.4 Commonly Used Functions

In the following sections, basic explanations and definitions will be given for some of the most useful functions within PVM. A fuller explanation of these functions together with definitions of many other functions can be found in [5].

Function return values are usually error codes but not always. Error codes are 0 for correct operation and negative for error normally.

2.4.1 `pvm_mytid()`

The syntax of `pvm_mytid()` is as follows:

```
int tid = pvm_mytid()
```

This function should be the first PVM function called by any program. Under normal circumstances it should return the tid of the current process. If the returned value if less than 0 then an error has occurred, most likely the virtual machine cannot be contacted.

2.4.2 pvm_parent()

This function determines the tid of the process which spawned the current process. It's syntax is as follows:

```
int tid = pvm_parent()
```

The return value should be greater than zero. It can be checked against the constant `PvmNoParent` to see if the current process actually has a parent. This is especially useful if an SPMD model is being utilised.

2.4.3 pvm_spawn()

The syntax of `pvm_spawn()` is as follows:

```
int tasks = pvm_spawn(
                      char      *task     /* in  */,
                      char      **argv    /* in  */,
                      int       flag      /* in  */,
                      char      *where    /* in  */,
                      int       ntask     /* in  */,
                      int       *tids     /* out */)
```

This is the function used to start tasks within a parallel application. `task` is the name of the executable you wish to start. In an SPMD model this is commonly the name of the currently running executable. `argv` should be an array holding the arguments you wish to pass to this executable. The host you wish the executable to be launched on can be specified using `where`. The number of times you wanted the executable launched should be given as `ntask`. `tids` is given as output to the function and is an integer array containing the task identifiers of the newly spawned tasks. The value of `flag` is the sum of various constants, the most useful of which are described in the table below.

Constant	Meaning
`PvmTaskDefault`	PVM can choose any machine to run the new tasks
`PvmTaskHost`	`where` specifies a particular host to run the tasks
`PvmTaskTrace`	new tasks will generate trace data
`PvmHostCompl`	`where` specifies a host that should not run new tasks

The function returns the number of tasks it was able to spawn. This should always be checked against the expected value.

2.4.4 pvm_initsend()

This function creates a new send buffer and makes it active. It's syntax is as follows:

```
int bufid = pvm_initsend(int    encoding    /* in */)
```

It returns the identifier given to the new buffer. The `encoding` parameter is a PVM constant, usually the value `PvmDataDefault` is sufficient.

2.4.5 pvm_mcast()

This function is used to broadcast the data in the currently active send buffer to all processes in a given list. It's syntax is as follows:

```
int result = pvm_mcast(
                       int     *tids          /* in */,
                       int     ntask          /* in */,
                       int     tag            /* in */)
```

`tids` is an array holding the task identifiers of all the tasks you want to be sent the message. If the tid of the current task appears in this array, it is ignored. `ntask` is the number of tasks the message should be sent to and `tag` is the message tag it should carry. As usual, the return value is an error code which is negative if an error has occurred.

2.5 Compiling & Running PVM Programs

2.5.1 Before You Begin

Unlike MPI, PVM is not just a standard which a set of library functions must conform to. PVM is more like a complete system and there is therefore no choice regarding distribution. While the system is available for several platforms, departmentally we use Linux PVM 3.0.

Currently PVM 3.0 is located in the directory below.

```
/net/dazdak/fp/pvm3
```

This directory will hence forth be referred to as `PVMHOME`. `PVMHOME/lib` should be added to your `PATH` environment variable to gain access to scripts and executables used to compile and run PVM programs.

PVM requires two other environment variables to be set; `PVM_ROOT` should be set to `PVMHOME` and `PVM_ARCH` should be set to `LINUX`.

As with MPI the setup of PVM should not need to be completed by the day-to-day user. It is however discussed as PVM can be installed and run on any collection of workstations within the department or otherwise. Installation and basic setup is therefore covered in Appendix A.2.

The PVM virtual machine, when spawning processes (see 2.2) looks for executables in the directory:

```
$HOME/pvm3/bin/$PVM_ARCH
```

You must therefore ensure this directory exists. It will also be useful to add this to your `PATH` environmental variable.

2.5.2 Compiling

There is a pre-written shell script to compile C programs that make use of PVM function calls. aimk is located in `PVMHOME/lib`. However this is complex to use. Further information on it can be found in [5].

It is possible to compile programs without using this script providing the appropriate libraries and include files are made known to the compiler. For example:

```
gcc -c coins.c -I/net/dazdak/fp/pvm3/include
gcc -o coins coins.o -L/net/dazdak/fp/pvm3/lib/LINUX -lpvm3
```

As mentioned above, the PVM virtual machine expects executables to be found in a specific directory so you must ensure this.

2.5.3 Running

The first and most important thing that must be ensured before trying to run a PVM program is to ensure that the virtual machine is running. This is discussed in A.2.

As the virtual machine takes care of the spawning of processes etc, the only action required to run a PVM program is to invoke it from the shell, for example:

```
./coins 4
```

2.6 Extended Example Using PVM

In order to demonstrate the use of the concepts described, a parallelisation of the algorithm described in section 1.7.1 will now be presented. This genetic algorithm will be parallelised using the modifications given in section 1.7.2.

As a first step in parallelising this algorithm using PVM we must provide the ability to transmit candidate structs between processes. In the MPI example, this was achieved using a derived datatype (1.4.2). In PVM all that is required is that we write a function which takes a candidate struct and packs each member onto the active buffer in turn for sending and another function which unpacks each member in turn when receiving. The packing function takes the form:

```
int pkcand(Cand* a) {
  int ret;
  ret = pvm_pklong(&a->value[0],MAXCOIN,1);
  if (ret < 0) return 0;
```

```
  ret = pvm_pklong(&a->fit,1,1);
  if (ret < 0) return 0;
  ret = pvm_pklong(&a->total,1,1);
  if (ret < 0) return 0;
  ret = pvm_pklong(&a->count,1,1);
  if (ret < 0) return 0;
  return 1;
}
```

The function that will be used to unpack candidate structs will take a similar form.

As with the MPI main program, an SPMD model will be adopted and so the stream of control will fork on whether a process is the master or a worker. The first task of either process, after finding its own tid, is therefore to determine whether it has a parent as follows:

```
myparent = pvm_parent();
```

The first main task of the master process is to spawn the worker processes. The number of worker processes needed by default shall be 4, but can be altered to arguments passed via the command line. Spawning is achieved by the following function call:

```
res = pvm_spawn(argv[0],(char**)0,PvmTaskTrace,
                (char*)0, WORKERS-1, slaves);
```

if (res != WORKERS-1) pvm_perror("spawn error");

It specifies the name of the current executable as the new task to be launched. No arguments needed to be passed to the executable so a null pointer can be passed here. It is best to let PVM determine where to spawn the tasks but trace data is useful, therefore `PvmTaskTrace` is used. After calling any PVM function, checks should be made to ensure that it operated as expected. `pvm_spawn()` (2.4.3) returns the number of processes successfully spawned. Should any PVM function not have operated correctly `pvm_perror()` can be used to produce a meaningful error message.

Once the processes have been spawned correctly, the master program should obtain control information from the user and distribute it amongst the workers using a broadcast. The code which achieves this is as follows:

```
pvm_initsend(PvmDataDefault);
pvm_pklong(&sum,1,1);
pvm_pklong(&max,1,1);
pvm_pklong(&min,1,1);
pvm_pklong(&LOCAL_CANDS,1,1);
pvm_mcast(slaves,WORKERS-1,11);
```

This illustrates the process of sending within PVM well; the buffer is initialised, packed and then sent. In the workers control stream these control values are received as follows:

```
pvm_recv(myparent,11);
pvm_upklong(&sum,1,1);
pvm_upklong(&max,1,1);
pvm_upklong(&min,1,1);
pvm_upklong(&LOCAL_CANDS,1,1);
```

The worker then goes into a loop where it continually processes candidates and then waits to see if it needs to process more. These orders are sent and received using a broadcast as above.

The first step in actually processing the candidates is to distribute them. Unlike MPI there is no function which automatically distributes an array. Therefore it must be done manually as follows:

```
for (j=1; j<WORKERS; j++) {
  pvm_initsend(PvmDataDefault);
  for (i=LOCAL_CANDS*j; i<LOCAL_CANDS*(j+1); i++)
    pkcand(&cand[i]);
  pvm_send(slaves[j-1],77);
}
```

The same principals as used in the previous send are applied. Receiving is as expected and the same methods should be applied to gather back the results of candidate processing.

When all tasks exit using `pvm_exit()`, the algorithm is complete.

The complete coding of the algorithm discussed above is given in Appendix D. The code for the auxiliary functions is given in Appendix E.

Chapter 3

Monitoring & Timing

Of interest is how the different methodologies applied in MPI & PVM compare. To achieve this, some performance evaluation was done. This also shows some motivation for parallelising an algorithm such as the genetic coin calculator used as an example in this document. Both code examples have timing mechanisms which give the average number of seconds it took to process each generation and this gives a good indication of the performance of the programs.

For performance testing, a slightly adapted version of the code was used which produces minimal output and has had the random number generator altered so that the results of the program were predictable and reproducible. The main factor which affects the performance of this algorithm is the number of candidates used. Therefore tests were done using first 10,000 and then 100,000 candidates. The *speedup* figures given are a measure of how much faster the algorithm ran on n processors compared with one processor. The speedup formula is given below. Note that the one processor case is the parallel code run on a single process and not the sequential code given however results given are extremely similar.

$$speedup\ on\ n = \frac{time\ on\ 1}{time\ on\ n}$$

10,000 candidates

Figures given are an average across 18 attempts.

Processes	MPI	MPI Speedup	PVM	PVM Speedup
1	6.646		6.774	
2	1.719	3.87	1.288	5.26
4	0.750	8.86	0.967	7.01
8	0.577	11.52	1.546	4.38
12	0.538	12.35	2.114	3.154
16	0.493	13.48	2.728	2.483
24	0.668	9.95	3.004	2.255

100,000 candidates

Figures given are an average across 9 attempts.

Processes	MPI	MPI Speedup	PVM	PVM Speedup
1	1387.69		1785.56	
2	357.22	3.89	320.999	5.56
4	71.99	19.28	59.417	30.05
8	25.55	54.31	17.617	101.35
12	10.14	136.85	17.256	103.48
16	8.51	163.07	16.114	110.83
24	7.52	184.53	26.620	67.08

Appendix A

Setup of MPI & PVM

A.1 Installation & Setup of MPI

The current release of MPICH can be obtained from the following URL;

```
http://www-unix.mcs.anl.gov/mpi/mpich/download.html
```

If you obtain *mpich.tar.gz*, the steps for installation of the system are as follows:

- Unzip the file you download as follows:

  ```
  gunzip -c mpich.tar.gz | tar xovf -
  ```

- Tailor MPICH to your architecture by executing the following command which specifies the directory you wish to place the distribution within:

  ```
  ./configure -prefix=/usr/local/mpich-1.2.0
  ```

- Compile the distribution by typing:

  ```
  make
  ```

- Edit the following file, (where `xxx` is the mpich's name for your architecture) to include all of the host names you want to make available to the run parallel programs on. The use of this file is also discussed in section 1.6.3.

  ```
  /usr/local/mpich/util/machines/machines.xxx
  ```

- Again presuming you have used a similar installation directory; you should add the following directory to your `PATH` environment variable.

  ```
  /usr/local/mpich/bin
  ```

A.2 Installation & Setup of PVM

A.2.1 Installation

The current release of PVM can be obtained from the following URL:

```
http://www.netlib.org/pvm3/index.html
```

It comes in tar gzip format so appropriate steps to unzip, untar and compile it will need to be taken.

A.2.2 Setup of the Virtual Machine

The first step in setting up the virtual machine is to invoke a console from any of the nodes you will be using. Providing your environment has been altered correctly as described in section 2.5.1 this is done using the command `pvm`.

Within the console hosts can be added to the virtual machine using a command of the form:

```
add olorun
```

The console will also accept a file of host names to add as a command line argument.

To exit the console, but leave the virtual machine running, the command `quit` should be used. If you wish to kill the virtual machine, the command `halt` should be used from within the console.

Appendix B

Sequential Algorithm Coding

```
#include "coins.h"
#include <sys/timeb.h>

#define PROMPT 1

void newgen(Cand cands[],
    long select,
    long cross,
    long mut) {
  reproduce(cands, select);   /* reproduce selected genes */
  crossover(cands, cross);    /* crossover required number of genes */
  mutate(cands, mut);         /* mutate required number of genes */
}


void main(int argc, char* argv[]) {

  long
    SELECT,                 /* no of genes to reproduce in each new gen */
    MUTATE,                 /* no of genes to mutate */
    CROSS,                  /* no of genes to 'cross' */
    MAXPRINT,               /* print how many candidates? */
    STEP,                   /* stop after each generation? */
    CHANGE,                 /* only print if most fit has changed? */
    g=0,                    /* no of generations */
    oldfittest=0,           /* fitness of current fittest cand */
    a,b;

  struct timeb t1,t2;

  double
```

```
  gentime,
  total=0;

initrand();             /* initialise rand num generator */
initcoins();            /* initialise coin types eg 1p, 5p ... */

/* get control info from user */
get_data(PROMPT,&sum,&max,&min,&CANDS,&SELECT,&CROSS,&MUTATE,
         &MAXPRINT,&STEP,&CHANGE);

 /* generate initial candidate set */
initcands();

/* sort the candidates in order of fitness */
calcfit(cand,CANDS);
sort(cand,CANDS);
printcands(MAXPRINT);

while(cand[0].fit || cand[0].count<min || cand[0].count>max) {
  ftime(&t1);

  /* while the ultimate candidate hasn't been found */

  /* create a new generation of candidates */
  newgen(cand, SELECT, CROSS, MUTATE);

  /* sort the candidates in order of fitness */
  calcfit(cand,CANDS);
  sort(cand,CANDS);

  /* output results if required */
  if(!CHANGE || cand[0].fit!=oldfittest) {
    if(STEP)
      getchar();
    printf("GENERATION %d sum %d max %d min %d",g,sum,max,min);
    printf(" cands %d sel %d cross %d mut %d\n",CANDS,SELECT,CROSS,MUTATE);
    printcands(MAXPRINT);
  }
  oldfittest=cand[0].fit;
  g++;

  ftime(&t2);
  a = (t1.time*1000)+t1.millitm;
  b = (t2.time*1000)+t2.millitm;
  total += (b-a)/1000.0;
}
```

```
  printf("average gen proc time %f secs\n", (total / (g-1)));
}
```

Appendix C

MPI Algorithm Coding

```
#include "coins.h"
#include "mpi.h"

#define PROMPT 1

process(MPI_Datatype mpi_cand_type,
long LOCAL_CANDS) {

  Cand** result;              /* storage for results of all processes */
  Cand* local_cand;           /* storage for each processes' set of cands */
  MPI_Status status;
  int i,j,rank;

  /* FIND MY RANK */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /* DISTRIBUTE ALL CANDIDATES AMONGST WORKERS */
  if (WORKERS > 1) {
    local_cand = (Cand*) malloc(LOCAL_CANDS*sizeof(Cand));
    MPI_Scatter(&cand, LOCAL_CANDS, mpi_cand_type,
                local_cand, LOCAL_CANDS, mpi_cand_type, 0, MPI_COMM_WORLD);
  }
  else
    local_cand = cand;

  /* CALCULATE FITNESS OF EACH LOCAL CANDIDATE */
  calcfit(local_cand, LOCAL_CANDS);

  /* SORT LOCAL CANDIDATES IN ORDER OF FITNESS */
  sort(local_cand, LOCAL_CANDS);
```

```
  /* IF I AM NOT THE MASTER, SEND RESULTS TO MASTER */
  if (rank != 0)
     MPI_Send(local_cand,LOCAL_CANDS,mpi_cand_type,0,1,MPI_COMM_WORLD);

  /* ELSE I AM THE MASTER, ACCEPT THE WORKERS RESULTS */
  else {
    if (WORKERS>1) {

      result = (Cand**) malloc(WORKERS*sizeof(Cand*));
      for (i=0; i<WORKERS; i++)
        result[i] = (Cand*) malloc(LOCAL_CANDS*sizeof(Cand));

      for (i=1; i<WORKERS; i++)
        MPI_Recv(&result[i][0],LOCAL_CANDS,mpi_cand_type,
                 MPI_ANY_SOURCE,1,MPI_COMM_WORLD, &status);

      /* COPY MY RESULTS TO GLOBAL RESULTS SET */
      for (i=0; i<LOCAL_CANDS; i++)
        result[0][i] = local_cand[i];

      /* MERGE EACH SEPERATE SET OF RESULTS */
      merge(result, WORKERS, LOCAL_CANDS, cand);

      for (i=0; i<WORKERS; i++)
        free(result[i]);
      free(result);
      free(local_cand);
    }
  }
}

void newgen(Cand cands[],
    long select,
    long cross,
    long mut) {
  reproduce(cands, select);   /* reproduce selected genes */
  crossover(cands, cross);    /* crossover required number of genes */
  mutate(cands, mut);         /* mutate required number of genes */
}

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

  long
    LOCAL_CANDS,          /* no of candidates distributed to each process */
    SELECT,               /* no of genes to reproduce in each new gen */
    MUTATE,               /* no of genes to mutate */
```

```
  CROSS,                /* no of genes to 'cross' */
  MAXPRINT,             /* print how many candidates? */
  STEP,                 /* stop after each generation? */
  CHANGE,               /* only print if most fit has changed? */
  g=0,                  /* no of generations */
  oldfittest=0;         /* fitness of current fittest cand */

double
  genstart,
  genstop,
  gentimes[100],
  total = 0.0;

int
  my_rank,              /* rank of process */
  newg = 1,
  contin = 1,
  i;

MPI_Datatype mpi_cand_type;  /* transmission representation of cand */
MPI_Status status;           /* return status for receive */

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

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

/* find out number of processes */
MPI_Comm_size(MPI_COMM_WORLD, &WORKERS);

/* build transmission representation for cand structs */
build_derived_type(cand[0], &mpi_cand_type);

initcoins();            /* initialise coin types eg 1p, 5p ... */

 if (my_rank == 0) {
  /* i must be the master */

  initrand();             /* initialise rand num generator */

  /* get control info from user */
  get_data(PROMPT,&sum,&max,&min,&CANDS,&SELECT,&CROSS,&MUTATE,
           &MAXPRINT,&STEP,&CHANGE);

  MPI_Bcast(&sum, 1, MPI_LONG, 0, MPI_COMM_WORLD);
```

```
MPI_Bcast(&max, 1, MPI_LONG,  0, MPI_COMM_WORLD);
MPI_Bcast(&min, 1, MPI_LONG, 0, MPI_COMM_WORLD);
LOCAL_CANDS = CANDS / WORKERS;
MPI_Bcast(&LOCAL_CANDS, 1, MPI_LONG, 0, MPI_COMM_WORLD);

/* generate initial candidate set */
initcands();

/* sort the candidates in order of fitness */
process(mpi_cand_type,LOCAL_CANDS);
if (PROMPT) printcands(MAXPRINT);

/* tell workers to prepare for next generation */
MPI_Bcast(&newg, 1, MPI_INT, 0,  MPI_COMM_WORLD);

while(cand[0].fit || cand[0].count<min || cand[0].count>max) {
  /* while the ultimate candidate hasn't been found */
  genstart = MPI_Wtime();

  /* create a new generation of candidates */
  newgen(cand, SELECT, CROSS, MUTATE);

  /* sort the candidates in order of fitness */
  process(mpi_cand_type,LOCAL_CANDS);

  /* output results if required */
  if(!CHANGE || cand[0].fit!=oldfittest) {
    if(STEP)
      getchar();
    if (PROMPT) {
      printf("GENERATION %d sum %d max %d min %d",g,sum,max,min);
      printf(" cands %d sel %d cross %d mut %d\n",CANDS,SELECT,
             CROSS,MUTATE);
      printcands(MAXPRINT);
    }
  }

  oldfittest=cand[0].fit;
  g++;

  /*  Decide if there will be a new generation and tell workers */
  if (cand[0].fit || cand[0].count<min || cand[0].count>max)
    newg = 1;
  else
    newg = 0;
  MPI_Bcast(&newg, 1, MPI_INT, 0,  MPI_COMM_WORLD);
```

```
      genstop = MPI_Wtime();
      gentimes[g] = genstop - genstart;
    }

    for (i=0; i<g; i++)
      total += gentimes[i];
    printf("%f", total / (g));
    if (PROMPT) printf(" avg sec/gen");
  }

  else {
    /* I must be a worker */
    /* Populate global variables */
    MPI_Bcast(&sum, 1, MPI_LONG, 0, MPI_COMM_WORLD);
    MPI_Bcast(&max, 1, MPI_LONG, 0, MPI_COMM_WORLD);
    MPI_Bcast(&min, 1, MPI_LONG, 0, MPI_COMM_WORLD);
    MPI_Bcast(&LOCAL_CANDS, 1, MPI_LONG, 0, MPI_COMM_WORLD);

    while (contin == 1) {
      /* while the master still needs me */
      /* process another generation */
      process(mpi_cand_type,LOCAL_CANDS);

      /* receive new orders */
      MPI_Bcast(&contin, 1, MPI_INT, 0, MPI_COMM_WORLD);
    }
  }
  /* shut down MPI */
  MPI_Finalize();
}


void build_derived_type(Cand subject,
                        MPI_Datatype* new_type_ptr) {

  int block_lengths[4];         /* number of blocks in each element of type */
  MPI_Aint displacements[4];    /* displacement of each element from start */
  MPI_Datatype typelist[4];     /* type of each element */
  MPI_Aint start_address;       /* used for calculating displacements */
  MPI_Aint address;

  /* set lengths, first element is an array, rest are single blocks */
  block_lengths[0] = MAXCOIN;
  block_lengths[1] = block_lengths[2] = block_lengths[3] = 1;
```

```
  /* set types, all elements are 'long' */
  typelist[0] = typelist[1] = typelist[2] = typelist[3] = MPI_LONG;

  /* element 0 is displacement 0 from start */
  displacements[0] = 0;

  /* work out starting address */
  MPI_Address(&subject.value[0], &start_address);

  /* work out displacements of other elements*/
  MPI_Address(&subject.fit, &address);
  displacements[1] = address - start_address;
  MPI_Address(&subject.total, &address);
  displacements[2] = address - start_address;
  MPI_Address(&subject.count, &address);
  displacements[3] = address - start_address;

  /*build type */
  MPI_Type_struct(4, block_lengths, displacements, typelist, new_type_ptr);

  /* finalise type */
  MPI_Type_commit(new_type_ptr);
}
```

Appendix D

PVM Algorithm Coding

```
#include "coins.h"
#include <pvm3.h>
#include <sys/timeb.h>

#define PROMPT 1

void newgen(Cand cands[],
    long select,
    long cross,
    long mut) {
  reproduce(cands, select);   /* reproduce selected genes */
  crossover(cands, cross);    /* crossover required number of genes */
  mutate(cands, mut);         /* mutate required number of genes */
}

int pkcand(Cand* a) {
  int ret;

  ret = pvm_pklong(&a->value[0],MAXCOIN,1);
  if (ret < 0) return 0;
  ret = pvm_pklong(&a->fit,1,1);
  if (ret < 0) return 0;
  ret = pvm_pklong(&a->total,1,1);
  if (ret < 0) return 0;
  ret = pvm_pklong(&a->count,1,1);
  if (ret < 0) return 0;
  return 1;
}


int upkcand(Cand* a) {
```

```
  int ret;

  ret = pvm_upklong(&a->value[0],MAXCOIN,1);
  if (ret < 0) return 0;
  ret = pvm_upklong(&a->fit,1,1);
  if (ret < 0) return 0;
  ret = pvm_upklong(&a->total,1,1);
  if (ret < 0) return 0;
  ret = pvm_upklong(&a->count,1,1);
  if (ret < 0) return 0;
  return 1;
}

int process (int myparent,
     int mytid,
     int* slaves,
     int LOCAL_CANDS) {

  Cand** result;              /* storage for results of all processes */
  Cand* local_cand;           /* storage for each processes' set of cands */

  int
    res, tid, tag, len,
    master=0,
    i,j;

  if (myparent == PvmNoParent) master=1;

  /* DISTRIBUTE ALL CANDIDATES AMONGST WORKERS */
  if (WORKERS>1) {
    local_cand = (Cand*) malloc(LOCAL_CANDS*sizeof(Cand));
    if (master) {
      /* send */
      for (j=1; j<WORKERS; j++) {
        pvm_initsend(PvmDataDefault);
        for (i=LOCAL_CANDS*j; i<LOCAL_CANDS*(j+1); i++)
          pkcand(&cand[i]);
        pvm_send(slaves[j-1],77);
      }
      for (i=0; i<LOCAL_CANDS; i++)
        local_cand[i] = cand[i];
    }
    else {
      /* receive */
      res = pvm_recv(myparent,77);
      for (i=0; i<LOCAL_CANDS; i++)
```

```
      upkcand(&local_cand[i]);
    }
  }
  else
    local_cand = cand;


  /* CALCULATE FITNESS OF EACH LOCAL CANDIDATE */
  calcfit(local_cand, LOCAL_CANDS);

  /* SORT LOCAL CANDIDATES IN ORDER OF FITNESS */
  sort(local_cand, LOCAL_CANDS);

  /* IF I AM NOT THE MASTER, SEND RESULTS TO MASTER */
  if (!master) {
    /* send */
    pvm_initsend(PvmDataDefault);
    for (i=0; i<LOCAL_CANDS; i++)
      pkcand(&local_cand[i]);
    pvm_send(myparent,88);
  }

  /* ELSE I AM THE MASTER, ACCEPT THE WORKERS RESULTS */
  else {
    if (WORKERS>1) {
      result = (Cand**) malloc(WORKERS*sizeof(Cand*));
      for (i=0; i<WORKERS; i++)
        result[i] = (Cand*) malloc(LOCAL_CANDS*sizeof(Cand));

      /* receive */
      for (j=1; j<WORKERS; j++) {
        res = pvm_recv(slaves[j-1],88);
        for (i=0; i<LOCAL_CANDS; i++)
          upkcand(&result[j][i]);
      }

      /* COPY MY RESULTS TO GLOBAL RESULTS SET */
      for (i=0; i<LOCAL_CANDS; i++)
        result[0][i] = local_cand[i];

      /* MERGE EACH SEPERATE SET OF RESULTS */
      merge(result, WORKERS, LOCAL_CANDS, cand);
    }
    return 1;
  }
}
```

```
int main(int argc, char* argv[]) {

  long
    LOCAL_CANDS,          /* no of cands to be processed by each cand */
    SELECT,               /* no of genes to reproduce in each new gen */
    MUTATE,               /* no of genes to mutate */
    CROSS,                /* no of genes to 'cross' */
    MAXPRINT,             /* print how many candidates? */
    STEP,                 /* stop after each generation? */
    CHANGE,               /* only print if most fit has changed? */
    g=0,                  /* no of generations */
    oldfittest=0,         /* fitness of current fittest cand */
    a,b,
    value;

  int
    mytid,                /* task id of current process */
    myparent,             /* task id of the parent of the current task */
    master=0,             /* whether current task is the master */
    *slaves,              /* task identifiers of the workers */
    res, tid, tag, len,
    contin=1, i;

  struct timeb t1,t2;

  double
    gentime,
    total=0;

  initrand();             /* initialise rand num generator */
  initcoins();            /* initialise coin types eg 1p, 5p ... */

  /* determine tid, parent and status */
  mytid = pvm_mytid();
  myparent = pvm_parent();
  if (myparent == PvmNoParent) master=1;

  if (master) {
    /* spawn processes */
    if (argc == 2) WORKERS = atoi(argv[1]);
    else WORKERS = 4;
    slaves = (int*) malloc((WORKERS-1)*sizeof(int));
    if (WORKERS > 1) {
      res = pvm_spawn(argv[0],(char**)0,PvmTaskTrace,(char*)0,
      WORKERS-1, slaves);
```

```
  if (res != WORKERS-1) pvm_perror("spawn error");
}

/* get control info from user */
get_data(PROMPT,&sum,&max,&min,&CANDS,&SELECT,&CROSS,&MUTATE,
         &MAXPRINT,&STEP,&CHANGE);

/* broadcast test control info to workers */
pvm_initsend(PvmDataDefault);
pvm_pklong(&sum,1,1);
pvm_pklong(&max,1,1);
pvm_pklong(&min,1,1);
LOCAL_CANDS = CANDS / WORKERS;
pvm_pklong(&LOCAL_CANDS,1,1);
pvm_mcast(slaves,WORKERS-1,11);

/* generate initial candidate set */
initcands();

/* sort the candidates in order of fitness */
process(myparent,mytid,slaves,LOCAL_CANDS);
if (PROMPT) printcands(MAXPRINT);

/*  Decide if there will be a new generation and tell workers */
if (cand[0].fit || cand[0].count<min || cand[0].count>max)
  contin = 1;
else
  contin = 0;
pvm_initsend(PvmDataDefault);
pvm_pkint(&contin,1,1);
pvm_mcast(slaves,WORKERS-1,99);

while(cand[0].fit || cand[0].count<min || cand[0].count>max) {
  ftime(&t1);
  /* while the ultimate candidate hasn't been found */

  /* create a new generation of candidates */
  newgen(cand, SELECT, CROSS, MUTATE);

  /* sort the candidates in order of fitness */
  process(myparent,mytid,slaves,LOCAL_CANDS);

  /* output results if required */
  if(PROMPT && (!CHANGE || cand[0].fit!=oldfittest)) {
    if (STEP)
       getchar();
```

```
      printf("GENERATION %d sum %d max %d min %d",g,sum,max,min);
      printf(" cands %d sel %d cross %d mut %d\n",CANDS,SELECT,CROSS,MUTATE);
      printcands(MAXPRINT);
    }
    oldfittest=cand[0].fit;
    g++;

    /*  Decide if there will be a new generation and tell workers */
    if (cand[0].fit || cand[0].count<min || cand[0].count>max)
      contin = 1;
    else
      contin = 0;
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&contin,1,1);
    pvm_mcast(slaves,WORKERS-1,99);

    ftime(&t2);
    a = (t1.time*1000)+t1.millitm;
    b = (t2.time*1000)+t2.millitm;
    total += (b-a)/1000.0;
  }
  printf("%f", total / (g));
  if (PROMPT) printf(" avg sec/gen");
}
else {
  /* populate globals */
  pvm_recv(myparent,11);
  pvm_upklong(&sum,1,1);
  pvm_upklong(&max,1,1);
  pvm_upklong(&min,1,1);
  pvm_upklong(&LOCAL_CANDS,1,1);

  /* get orders */
  while (contin) {
    process(myparent,mytid,(int*)0,LOCAL_CANDS);
    pvm_recv(myparent,99);
    pvm_upkint(&contin,1,1);
  }
}
pvm_exit();
return 0;
}
```

Appendix E

Auxiliary Code Listings

E.1 Definitions

```
/* Definitions used by all  coins programs */

#ifndef MPCOINS_H
#define MPCOINS_H

#include <stdio.h>
#include <string.h>

#define MAXCOIN 7          /* max number of coin types */
#define MAXC 500000        /* max number of candidates */
#define MAXNO 100          /* max initial no of coins */

/* candidate gene definition */
typedef struct acand {
  long value[MAXCOIN];
  long fit;
  long total;
  long count;
} Cand;

/* global definitions */
long coins[MAXCOIN];       /* coin values */
Cand cand[MAXC];           /* complete candidate set */
long CANDS;                /* how many candidates in total ? */
long sum,max,min;          /* how much money/max no of coins/min no of coins? */
int WORKERS;               /* number of worker processes */

/* fitness calculation functions */
```

```
void sort (Cand[],long);
void merge(Cand**,int,int,Cand[]);
void calcfit (Cand[],long);

/* input / output functions */
void printcand(long);
void printcands(long);
void readint(int,char*,long*,long);
void readi(int,char*,long*);
void readyn(int,char*,long*);
void getdata (int,long*,long*,long*,long*,
              long*,long*,long*,long*,long*,long*);

/* genetic manipulation functions */
void initrand();
void initcands();
void crossover(Cand[],long);
void mutate(Cand[],long);
void reproduce(Cand[],long);
void initcoins();
#endif
```

E.2 Genetic & Fitness Calculation Functions

```
#include "coins.h"

void initcoins() {
  coins[0]=100;
  coins[1]=50;
  coins[2]=20;
  coins[3]=10;
  coins[4]=5;
  coins[5]=2;
  coins[6]=1;
}

void initrand() {
  int seed;
  seed = time(0);
  srandom(seed);
}

void initcands() /* give each candidate a random no of coins */ {
  long i,j;
  for(i=0;i<CANDS;i++)
```

```
    for(j=0;j<MAXCOIN;j++)
      cand[i].value[j]=random()%MAXNO;
}

lswap(long *i,
      long *j) {  long t;
   t=*i;
   *i=*j;
   *j=t;
}

void crossover(Cand cands[],
       long crossover) {
  long i,j,k,l;
  for(k=0;k<crossover;k++) {
    i=random()%CANDS;   /* select 2 random candidates */
    j=random()%CANDS;
    l=random()%MAXCOIN; /* select random crossover point */
    lswap(&(cands[i].value[l]),&(cands[j].value[l]));
  }
}

void mutate(Cand cands[],
    long mutate) {
  /* change rand cand at rand code position to rand coin no */
  long i,j;
  for(i=0;i<mutate;i++)
    cands[random()%CANDS].value[random()%MAXCOIN]=random()%MAXNO;
}

void reproduce(Cand cands[],
       long sel) {
  /* copy 1st sel cands repeatedly to replace less fit cands */
  long i,j,k,c;
  for(i=0;i<CANDS/sel;i++)
    for(j=0;j<sel;j++)
      for(k=0;k<MAXCOIN;k++)
        cands[i*sel+j].value[k]=cands[j].value[k];
  c=sel*(CANDS/sel);
  for(j=c;j<CANDS;j++)
    for(k=0;k<MAXCOIN;k++)
      cands[j].value[k]=cands[j-c].value[k];
}

#include "coins.h"
```

```
void calcfit (Cand cand[],
      long num) {
  Cand* current;
  int i,j;

  for(i=0;i<num;i++)  {
    current = &cand[i];
    current->total=0;
    current->count=0;
    for(j=0;j<MAXCOIN;j++) {
        current->total=current->total+current->value[j]*coins[j];
        current->count=current->count+current->value[j];
      }
      current->fit=abs(sum-current->total);
      if(current->count<min)
       current->fit=current->fit+(min-current->count)*(min-current->count);
      else
      if(current->count>max)
       current->fit=current->fit+(current->count-max)*(current->count-max);
   }
}
```

E.3 Sort & Merge Functions

A bubble sort was used in the examples presented in this text. This is an extremely inefficient algorithm but made the computation involved in the main algorithm significant and therefore provided motivation for parallelisation. A more efficient algorithm would have meant that the cost of communication between processes would have greater than the cost of computation.

```
#include "coins.h"

swap(Cand** i,
     Cand** j) {
   Cand* t;
   t=*i;
   *i=*j;
   *j=t;
}

void sort (Cand cands[],
   long length) {
  int i,j;
  Cand sorted[length];
  Cand* local_I[length];
```

```
  /* create array of pointers */
  for (i=0; i<length; i++)
    local_I[i] = &(cands[i]);

  for(i=0;i<length-1;i++)
    for(j=i;j>=0;j--)
      if(local_I[j]->fit>local_I[j+1]->fit)
        swap(&(local_I[j]),&(local_I[j+1]));

   /* dereference pointer array */
   for (i=0; i<length; i++)
     sorted[i] = *(local_I[i]);

   /* copy new back to old */
   for (i=0; i<length; i++)
     cands[i] = sorted[i];
}

#include "coins.h"
#define MAXNUM 32000000

void getmin (long nums[],
     int n,
     long* min,
     int* index) {
  int i;
  *min = nums[0];
  *index = 0;
  for (i=1; i<n; i++)
    if (nums[i] < *min) {
      *min = (nums[i]);
      *index = i;
    }
}

void merge(Cand** parts,
   int lists,
   int items,
   Cand total[]) {
  int countp[lists];
  int sub[lists];
  int n = 0;
  int p = 0;
  long nums[lists];
  int i, k, l;
```

```
  long amin;
  int index;

  for (l=0; l<lists; l++)
    countp[l] = 0;
  for (k=0; k<(lists*items); k++) {
    for (i=0; i<lists; i++)
      if (countp[i] < items)
        nums[i] = parts[i][countp[i]].fit;
      else (nums[i] = MAXNUM);
    getmin(nums,lists,&amin,&index);
    total[p] = parts[index][countp[index]];
    countp[index]++;
    p++;
  }
}
```

E.4 Input & Output Functions

```
#include "coins.h"

void pcand (Cand* a) {
  printf("values(%d, %d, %d, %d, %d, %d, %d) fit(%d), count(%d), total(%d)\n",
         a->value[0],a->value[1],a->value[2],a->value[3],a->value[4],
         a->value[5],a->value[5], a->fit,a->count,a->total);
  fflush(stdout);
}

void printcand(long i) {
  long j;
  for(j=0;j<MAXCOIN;j++)
    printf("%4d ",cand[i].value[j]);
}

void printcands(long MAXPRINT) {
  long i;
  printf("cand ");
  for(i=0;i<MAXCOIN;i++)
    printf("%3dp ",coins[i]);
  printf("  sum  count  fit\n");
  for(i=0;i<MAXPRINT;i++) {
    printf("%4d ",i);
    printcand(i);
    printf("%5d %5d %5d\n",cand[i].total,cand[i].count,cand[i].fit);
  }
```

```
  printf("\n");
}

void readint(int p,
     char* m,
     long* i,
     long mm) /* read integer - limit check */ {
  if (p) printf("%s ",m);
  scanf("%d",i);
  if(*i>mm) {
    (*i)=mm;
    if (p) printf("restricted to %d\n",mm);
  }
}

void readi(int p,
   char * m,
   long * i) /* read integer - no limit check */ {
  if (p) printf("%s ",m);
  scanf("%d",i);
}

void readyn(int p,
    char * m,
    long * a) /* check for yes/no answer */ {
  char c;
  if (p)  printf("%s ",m);
  scanf("%c",&c);
  (*a)=c=='y'?1:0;
}

void get_data(int PROMPT,
      long *sum,        long *max,
      long *min,        long *CANDS,
      long *SELECT,     long *CROSS,
      long *MUTATE,     long *MAXPRINT,
      long *STEP,       long *CHANGE) {
  readi(PROMPT,"how much money?",sum);
  readi(PROMPT,"maximum number of coins?",max);
  readi(PROMPT,"minimum number of coins?",min);
  readint(PROMPT,"how many candidates? ",CANDS,MAXC);
  readint(PROMPT,"select how many fittest candidates?",SELECT,*CANDS);
  readint(PROMPT,"how many crossovers?",CROSS,*CANDS);
  readint(PROMPT,"how many mutations?",MUTATE,*CANDS);
  readint(PROMPT,"print how many fit candidates?",MAXPRINT,*CANDS);
  getchar();
```

```
  readyn(PROMPT,"stop after each generation?",STEP);
  getchar();
  readyn(PROMPT,"only show generation after change in most fit?",CHANGE);
}
```

Bibliography

[1] G. A. Geist, J. A. Kohl, P. A. Papadopoulos: *PVM and MPI: A Comparison of Features* 1996

[2] Peter S. Pacheco: *Parallel Programming with MPI* 1997.

[3] Message Passing Interface Forum: *A Message-Passing Interface Standard*, version 1.1 1995.

[4] William Gropp, Ewing Lusk: *User's Guide for mpich - a Portable Implementation of MPI 1.2.0*

[5] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiamg, Robert Mancheck, Vaidy Sunderam: *PVM: Parallel Virtual Machine* 1994.

Index