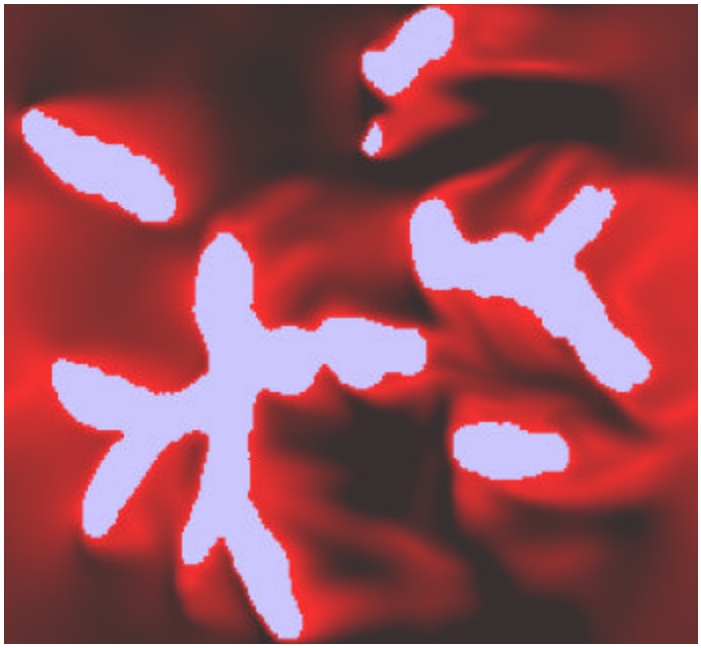
C2 Distributed Computing

Guide to the Practical Work



(simulated flow density pattern about an irregular complex-shaped obstacle)

February 2006

1 Introduction

Dear reader,

This is the guide for the practical work related to the course ``C2 Distributed Computing'', which you will need in addition to the book ``Parallel Scientific Computing in C++ and MPI'' by G.E. Karniadakis & R.M. Kirby (Cambridge University Press, 2003). This guide to the practical work is based, for a major part, on the guide to the practical work for the course ``Parallel Scientific Computing and Simulation'' originally written by Alfons Hoekstra in 2003. The course ``Distributed Computing'' is a new and highly modified version of the ``Parallel Scientific Computing and Simulation'' course, which is especially aimed for the master Computer Science. In this course the emphasis will be on the application of numerical methods in distributed computing.

The lab course is such that you start with an assignment to get hands-on experience with MPI. You will develop and test your parallel programs on the local workstation cluster (i.e. the ow cluster). However, you also get an account on a parallel computer, the 255 node Lisa of the University of Amsterdam. You will run your parallel programs on the Lisa in order to measure the scalability on a tightly coupled machine, and to perform some large simulations.

Please read this syllabus carefully. It explains everything you need to know about this lab course.

Good luck !!

Jaap Kaandorp & Walter Hoffmann
{jaapk, walter}@science.uva.nl

2 What you should know before you start

2.1 Lab report

You write *one* concise report of all the practical work you do for this course. The style of the report is just as you learned previously (so, a clear *introduction* stating the problem at hand, a *theory section* that introduces necessary theory, a *methods section* that describes the methods used, e.g. pseudo code of parallel algorithms, etc, a *results section* that presents the main results that you obtained, e.g. speedup measurements, pictures of the simulated results, and finally a *short discussion* that critically goes through the results in view of the original problem, like did you solve the problem satisfactory). Very important is that you also write a report about the software that you create. That is, you also provide text that explains how you implemented the simulations (e.g. explaining load balancing, or communication patterns). Very important is that any reader of your report must be able to *reproduce* your results. This more or less dictates the amount of detail that will go into your report. I suggest that you write a report for each assignment and then bundle these together into your final report. However, you are free to organize this in different ways.

If you are not sure about how to write the report, you are allowed to write a single small report and hand it in to the teacher after finalizing this first assignment. The teacher will not give a mark for this small report, but will provide feedback on the contents, so that you know what to do when writing the full final report. Please note that there are all time two lab assistants available during the lab sessions, if you have any questions regarding the practical work you should ask them for advice. We strongly advice you, if you have any questions, to attend the lab sessions.

You are allowed to write the report in groups of *maximum* two people. If you wish to collaborate with a fellow student, you should report this to your teacher when you start this practical work (by sending an e-mail).

You will get a single mark for your lab work. Together with the mark you get for the examination, your final mark is will be calculated as the average.

You can hand in your lab report by sending it over the e-mail . It should be either in postscript or PDF. Together with your lab report you must send (in a tar archive) all your source files and Makefiles. Please send your lab report, source files and Makefiles to the lab assistants. We should be able to compile and run your programs on the Lisa parallel computer (and this will be tested). You should also include a README file that explains which parts of the software relate to which assignments.

Deadline to hand in your report

There is just *one single* deadline. For the academic year 2005/2006 this is

Friday April **7,** 2006

No submissions will be accepted after this date..

2.2 MPI

2.2.1 Introduction

You develop your parallel programs using MPI. Both the local cluster of workstations (the ow-cluster) and the Lisa parallel computer have installed MPICH, the public domain version of MPI. You learned about MPI in the previous course *Introduction Parallel Computing*. Below you will find more detailed information about MPI that should be sufficient for the current lab course. For more in-depth information about MPI you may consult `http://www-unix.mcs.anl.gov/mpi/` or the on-line version of the MPI book: `http://www.netlib.org/utk/papers/mpi-book/mpi-book.html`.

You develop and test your programs on the local cluster (or, if you have a Linux system at home, you could get MPICH from the net and install it at home, allowing you to develop your MPI programs at home[1]). Next, you copy your program to the Lisa. On that system you do your performance measurements and you run your large simulations.

2.2.2 A Message Passing Standard: MPI

In principle, a sequential algorithm is portable to any architecture supporting the sequential paradigm. However, programmers require more than this: they want their realization of the algorithm in the form of a particular *program* to be portable, i.e. at the level of source code portability.

The same is true for message passing programs and forms the motivation behind MPI. MPI provides source code portability of message passing programs written in C or Fortran across a variety of architectures. Just as for the sequential case, this has many benefits, including:

- protecting investment in a program;
- allowing development of the code on one architecture (e.g., a network of workstations) before running it on the target machine (e.g., fast specialised parallel hardware).

The primary goal of the MPI specification is to demonstrate that users need not to compromise among efficiency, portability, and functionality. This means that one can write portable programs that can still take advantage of the specialized hardware and software offered by individual vendors. At the same time, advanced features, such as application-oriented process structures and dynamically managed process groups with an extensive set of collective operations, can be expected in every MPI implementation and can be used in every parallel application program where they might be useful.

MPI is not a revolutionary new way of programming parallel computers. Rather, it is an attempt to collect the best features of many existing message-passing systems, improve them where appropriate, and standardize them.

2.2.3 Basic Concepts of MPI

The elementary concepts of the MPI standard are the process and the message. While point-to-point and collective communications are central to MPI; process groups, message contexts, data types, and virtual topologies are other important concepts embedded into the MPI standard.

2.2.3.1 Messages and Processes

An MPI process is an entity that takes part in performing some computational task. In the programming model underlying MPI, each *process* is associated with a unique memory,

[1] There's also a windows version of MPI, so you could also try that. A student from previous year installed MPI under windows, and developed all programs in this environment, ported them (without any problems) to the Lisa computer and ran his experiments there.

called its *local memory*, which it only can directly access and update. A *message* consists of a piece of information, called its *body*, together with some additional data called an *envelope*. The envelope of a message specifies which recipient process can read the information in the body of the message into its local memory. The envelope contains further information that can be used by the recipient process to decide whether and when to open the message as well as how to access the data in the body of the message once it has been received.

The exchange of messages among processes is referred to as *communication* and is the only means by which MPI processes can access data in each other local memories or synchronize with one another. Communication in MPI takes place via explicit calls to C functions or Fortran subroutines specified in the MPI Forum [2].

2.2.3.1.1 Process Groups

A *process group* in MPI is a set of processes whose members are numbered consecutively starting from 0. The number associated with a process in a group is called its *rank* in the group. Groups are formed on the basis of tasks to be performed; that is, an MPI process group typically glues together processes that are working on a common task. A process can belong to any number of groups. MPI groups are said to be *static* because they all derive from the initial group and no new process may be created during the execution of the program. Apart from this limitation, MPI provides an extensive range of process group management facilities.

2.2.3.1.2 Communication Contexts

An individual MPI process can select whether or not to receive a message, depending on the envelope of that message. Receiving a message alters the local memory of the process and is not a reversible operation. A common source of error in programming under the message passing model is for a process to receive a message sent by the wrong process or a message sent by the right process but intended for some other stage of the computation. That issue is of particular relevance when the task assigned to a process involves calls to a parallel library procedure. The point here is that communications initiated directly by the user and communications initiated within the call to a library procedure belong to different *contexts*, or *universes*. Message passing interface that support message contexts make it possible for user and library messages to travel simultaneously without any risk of them being confused.

2.2.3.1.3 Virtual Topologies

The default ranking of processes within a group is not always the most natural for writing application software. For instance, the application might model some phenomenon in three-dimensional space. At other times, more efficient algorithms can be employed if the processes are known to form a two-dimensional mesh. In all these cases it is natural to think of the processes as being indexed according to some Cartesian coordinate system. Such a coordinate system may or may not reflect the actual physical connections among the processors running the processes in the group. Cartesian coordinate systems are called *Cartesian topologies* in MPI and can have any number of dimensions. MPI also provides for *graph topologies* that let the user specify explicitly the nearest-neighbor relations among processes. Processes that are nearest-neighbors according to a process topology need not run on processors that are physically connected; Cartesian and graph topologies are therefore called *virtual topologies*.

2.2.3.1.4 Communicators

The *communicator* construct ties together in MPI the concepts of process group, communication context, and virtual topology. A communicator is always associated with a process group, a communication context and a virtual topology. Groups and communicators are *opaque objects*, they can only be passed as arguments to MPI routines and possibly appear in Boolean expressions. All communications in MPI involve at least one communicator. All the processes involved in a communication must provide the same communicator as an argument in the corresponding MPI procedure calls.

2.2.3.2 MPI Datatypes

Data type systems in programming languages, like C and Fortran, tell the compiler how to interpret data and possibly how data are to be laid out in main memory. That information is of great importance to a congenial message passing interface and the MPI standard defines a message type system on top of the data type systems of the C and Fortran languages. MPI data types allow a programmer to specify where an MPI implementation is to pick-up and then deliver the body of messages in the local memory of processes. The simplest MPI data types are mainly the union of the atomic data types of C and Fortran with a prefix `MPI_` added to their names.

The MPI functionality goes even further. To achieve this the MPI standard provides MPI procedures to define MPI *derived data types*. An MPI derived data type specifies the format of a sequence of values and consists of: the number of elements in the sequence, the MPI data type of each element and the distance in bytes between elements of the sequence.

2.2.3.3 Communications

Calls to MPI communication procedures are classified according to when they can safely be issued, when they will return, by which processes they should be called and whether the MPI implementation is solely responsible for handling message buffers. Not all possible combinations could be included into the MPI Standard (nor are they all needed) but their number has resulted in the same name being used to mean different things.

An MPI communication procedure is *blocking* if a call to it will not return before it is safe for a process to re-use the resources specified in the call. Non-blocking MPI communication procedures are supplemented with procedures to test or to wait for the permission to re-use the resources specified in their call.

2.2.3.3.1 Point-to-Point Communications

A *point-to-point communication* involves only two processes: one process sending a message and one process receiving that message. Every MPI point-to-point communication procedure has a blocking and a non-blocking version.

Each MPI send procedure operates in one of four modes: *synchronous*, *buffered*, *standard* and *ready*. These communication modes specify the completion time of send operations and when a call to the corresponding send procedure can be safely issued. Note that a call to a non-blocking send procedure may return before the send itself is completed. For that reason, the non-blocking send procedures have an additional parameter: a handle to a send request returned by the non-blocking procedure. That handle can be used to test or wait for the completion of the send operation.

A call to a *synchronous* send is complete only after a matching call to a receive procedure has been issued. A send in the *buffered* mode completes whether or not a matching call to a receive procedure has been issued. The MPI standard defines procedures for managing buffer

space and the user is responsible for providing enough buffer space to store the outgoing message. The choice between calling a synchronous or a buffered send procedure involves weighting some trade-offs between the buffer space and execution time. The *standard* mode of communication relays the analysis of these trade-offs to the MPI implementation. Thus a call to a send procedure operating in the standard mode may or may not complete before a call to a matching receive procedure has been issued. A call to any send procedure operating in the standard, buffered or synchronous mode can be safely issued even when no matching receive call has been issued. That is not true for calls to send procedures operating in the *ready* mode. The correct operation of these send procedures relies on the programmer to ensure that the send will be matched by an appropriate receive. There is only one mode of operation for MPI procedures that receive a message.

2.2.3.3.2 Collective Communications

Collective communications are initiated by all the processes in the communicator within which the communication takes place. The MPI Standard provides for the major types of collective communications encountered in current practice. The MPI collective communication procedures are blocking. The *barrier*, described later, operates in synchronous mode; all other MPI collective communication procedures can be thought as operating in standard mode. The major forms of collective communications included in the MPI standard are reviewed in section 2.2.4.3.

2.2.3.4 Is MPI Large or Small?

Perhaps the most fundamental decision for the MPI Forum was whether MPI would be "small and exclusive," incorporating the minimal *intersection* of existing libraries, or "large and inclusive," incorporating the *union* of the functionality of existing systems. In the end, although some ideas were left out, an attempt was made to include a relatively large number of features that had proven useful in various libraries and applications.

2.2.3.4.1 MPI is Large

The many functions described in the last section mean that the MPI standard has many functions in it (about 125). Does this mean that MPI is impossibly complex? The answer is no for two reasons. First, the number of functions in MPI comes from combining a small number of orthogonal concepts. The number of *ideas* in MPI is small. Second, many of the routines represent added functionality that can be ignored until needed.

2.2.3.4.2 MPI is Small

By way of demonstrating just how little one needs to learn to write MPI programs, the number of indispensable functions the programmer really cannot do without is six, i.e.:

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

int MPI_Finalize(void)

int MPI_Comm_size(MPI_Comm comm, int *size)

int MPI_Comm_rank(MPI_Comm comm, int *rank)

int   MPI_Send(void   *buf,   int   count,   MPI_Datatype   datatype,
               int dest, int tag, MPI_Comm comm)
```

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

The above functions will be described, as well as other relevant MPI functions, in the next chapter.

With only these functions a vast number of useful and efficient programs can be written. The other functions all add flexibility (datatypes), robustness (nonblocking send/receive), efficiency ("ready" mode), modularity (groups, communicators), or convenience (collective operations, topologies).

2.2.3.4.3 MPI Is Whatever Size You Like

The designers of MPI attempted to make the features of MPI consistent and orthogonal. This means that users can incrementally add sets of functions to their repertoire as needed without learning everything at once.

2.2.4 The MPI API

2.2.4.1 Preliminaries

MPI comprises a library. An MPI process consists of a C or Fortran77 program which communicates with other MPI processes by calling MPI routines. The MPI routines provide the programmer with a consistent interface across a wide variety of different platforms.

All names of MPI routines and constants in both C and Fortran begin with the prefix `MPI_` to avoid name collisions. Fortran routine names are all upper case but C routine names are mixed case. Following the MPI document [2], when a routine name is used in a language-independent context, the uppercase version is used. All constants are in uppercase in both Fortran and C.

2.2.4.1.1 Initializing MPI

The first MPI routine called in any MPI program *must* be the initialization routine `MPI_Init`. Every MPI program must call this routine *once*, before any other MPI routines. Making multiple calls to `MPI_Init` is erroneous. The C version of the routine accepts the arguments to `main`, i.e. `argc` and `argv` as arguments.

```
int MPI_Init(int *argc, char ***argv);
```

2.2.4.1.2 MPI_COMM_WORLD and Communicators

`MPI_Init` defines something called `MPI_COMM_WORLD` for each process that calls it. `MPI_COMM_WORLD` is a *communicator.* All MPI communication calls require a communicator argument and MPI processes can only communicate if they share a communicator.

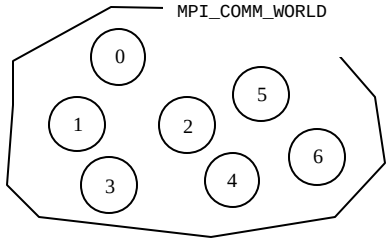


Figure 1: The predefined communicator `MPI_COMM_WORLD` for seven processes. The numbers indicate the rank of each process.

Every communicator contains a *group* which is a list of processes. The processes are ordered and numbered consecutively from 0. The number of each process is known as its *rank*. The rank identifies each process within the communicator. For example, the rank can be used to specify the source or destination of a message. Using `MPI_COMM_WORLD`, every process can communicate with every other. The group of `MPI_COMM_WORLD` is the set of all MPI processes.

An MPI process can query a communicator for information about the group, with `MPI_Comm_rank` and `MPI_Comm_size`.

```
int MPI_Comm_rank (MPI_Comm comm, int *rank)
```

`MPI_Comm_rank` returns in `rank` the rank of the calling process in the group associated with the communicator `comm`.

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

`MPI_Comm_size` returns in `size` the number of processes in the group associated with the communicator `comm`.

2.2.4.1.3 Clean-Up of MPI

An MPI program should call the MPI routine `MPI_Finalize` when all communications have completed. This routine cleans up all MPI data-structures, etc. It does not cancel outstanding communications, so it is the responsibility of the programmer to make sure all communications have completed. Once this routine has been called, no other calls can be made to MPI routines, not even `MPI_Init`, so a process cannot later re-enroll in MPI.

```
int MPI_Finalize(void)
```

2.2.4.1.4 Aborting MPI

```
int MPI_Abort(MPI_Comm comm, int errcode)
```

This routine attempts to abort all processes in the group contained in `comm` so that with `comm` = `MPI_COMM_WORLD` the whole program will terminate.

2.2.4.1.5 Timers and Synchronization

MPI defines a timer. A timer is specified even though it is not "message passing," because timing parallel programs is important in performance debugging and because existing timers are either inconvenient or do not provide adequate access to high-resolution timers.

```
double MPI_Wtime(void)
```

`MPI_Wtime` returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past. The times returned are local to the node that called them. There is no requirement that different nodes return the same time.

```
double MPI_Wtick(void)
```

`MPI_Wtick` returns the resolution of `MPI_Wtime` in seconds. That is, it returns the number of seconds between successive clock ticks.

2.2.4.2 Point-to-Point Communication

A *point-to-point* communication always involves exactly two processes. One process sends a message to the other. This distinguishes it from the other type of communication in MPI, *collective* communication, which involves a whole group of processes at one time.

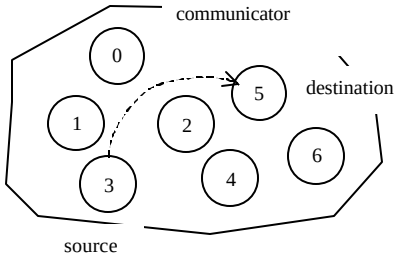


Figure 2: In point-to-point communications a process sends a message to another specific process.

To send a message, a *source* process makes an MPI call which specifies a *destination* process in terms of its rank in the appropriate communicator (e.g., `MPI_COMM_WORLD`). The destination process also has to make an MPI call if it is to receive the message.

There are four *communication modes* provided by MPI: *standard*, *synchronous*, *buffered*, and *ready*. The modes refer to four different types of *send*. It is meaningless to talk of communication mode in the context of a receive. "Completion" of a send means by definition that the send buffer can safely be re-used. The standard, synchronous and buffered send differ only in one respect: how completion of the send depends on the *receipt* of the message (see Table 1).

	Completion Condition
Synchronous send	Only completes when the receive has been issued.
Buffered send	Always completes (unless an error occurs), irrespective of whether the receive has completed
Standard send	Either synchronous or buffered.
Ready send	Always completes (unless an error occurs), irrespective of whether the receive has completed
Receive	Completes when a message has arrived.

Table 1: MPI communication modes.

All four modes exist in both blocking and non-blocking forms. In the blocking forms, return from the routine implies completion. In the non-blocking forms, all modes are tested for completion with the usual routines `MPI_Test`, `MPI_Wait`, etc.).

	Blocking	*Non-Blocking*
Standard send	`MPI_Send`	`MPI_Isend`
Synchronous send	`MPI_Ssend`	`MPI_Issend`
Buffered send	`MPI_Bsend`	`MPI_Ibsend`
Ready send	`MPI_Rsend`	`MPI_Irsend`
Receive	`MPI_Recv`	`MPI_Irecv`

Table 2: MPI communication routines.

2.2.4.2.1 Standard Send

The standard send has the following form:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

where

- `buf` is the address of the data to be sent.
- `count` is the number of elements of the MPI datatype which `buf` contains.
- `datatype` is the MPI datatype.
- `dest` is the destination process for the message. This is specified by the rank of the destination process within the group associated with the communicator `comm`.
- `tag` is a marker used by the sender to distinguish between different types of messages.
- `comm` is the communicator shared by the sending and receiving processes. Only processes that share the same communicator can communicate.

As already said, *completion* of a send means by definition that the send buffer can safely be re-used, i.e., the data has been sent.

2.2.4.2.2 Standard Receive

The format of the standard receive is:

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

where

- `buf` is the address where the data should be placed once received (the receive buffer). For the communication to succeed, the receive buffer *must* be large enough to hold the message without truncation. If it is not, behavior is undefined. The buffer may however be longer than the data received.
- `count` is the number of elements of a certain MPI datatype which `buf` can contain. The number of data elements actually received may be less than this.
- `datatype` is the MPI datatype for the message. This must match the MPI datatype specified in the send routine.

- `source` is the rank of the source of the message in the group associated with the communicator `comm`. Instead of prescribing the source, messages can be received from one of a number of sources by specifying a *wildcard*, `MPI_ANY_SOURCE`, for this argument.
- `tag` is used by the receiving process to prescribe that it should receive only a message with a certain tag. Instead of prescribing the tag, the wildcard `MPI_ANY_TAG` can be specified for this argument.
- `comm` is the communicator specified by both the sending and receiving process. *There is no wildcard option for this argument*.
- If the receiving process has specified wildcards for both or either of `source` or `tag`, then the corresponding information from the message that was actually received may be required. This information is returned in `status`, and can be queried using `status.MPI_SOURCE` and `status.MPI_TAG`.

Completion of a receive means by definition that a message arrived, i.e., the data has been received.

2.2.4.2.3 Shifts and MPI_Sendrecv

A *shift* involves a set of processes passing data to each other in a chain-like fashion (or circular fashion) (see for an example Figure 3). Each process sends a maximum of one message and receives a maximum of one message. A routine called `MPI_Sendrecv` provides a convenient way of expressing this communication pattern in one routine call without causing deadlock and without the complications of "red-black" methods.

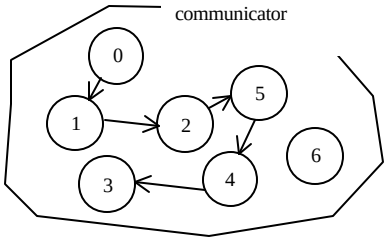


Figure 3: An example of a shift, where `MPI_Sendrecv` could be used.

Note that `MPI_Sendrecv` is just an extension of point-to-point communications. It is completely compatible with point-to-point communications in the sense that messages sent with `MPI_Sendrecv` can be received by a usual point-to-point receive and *vice versa*. In fact, all `MPI_Sendrecv` does is to combine a send and a receive into a single MPI call and make them happen simultaneously to avoid deadlock. It has nothing to do with collective communication and *need not* involve all processes in the communicator. As one might expect, the arguments to `MPI_Sendrecv` are basically the union of the arguments to a send and receive call:

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
                 MPI_Datatype sendtype, int dest, int sendtag,
                 void *recvbuf, int recvcount,
                 MPI_Datatype recvtype, int source, int recvtag,
                 MPI_Comm comm, MPI_Status *status)
```

The `MPI_Sendrecv` is *blocking* – there is no non-blocking form since this would offer nothing over and above two separate non-blocking calls. In Figure 3, process 0 only sends,

process 3 only receives, and process 6 does neither. A nice trick is to use `MPI_PROC_NULL` which makes the code more symmetric.

process	*destination*	*source*
0	1	`MPI_PROC_NULL`
1	2	0
2	5	1
3	`MPI_PROC_NULL`	4
4	3	5
5	4	2
6	`MPI_PROC_NULL`	`MPI_PROC_NULL`

Table 3: Communications from Figure 3

2.2.4.2.4 Datatype-Matching Rules

All MPI messages are *typed* in the sense that the type of the contents must be specified in the send and receive. The basic datatypes in MPI correspond to the basic C datatypes as shown in Table 4.

There are rules for datatype-matching and, with certain exceptions, the datatype specified in the receive must match the datatype specified in the send. The great advantage of this is that MPI can support *heterogeneous* parallel architectures, i.e., parallel machines built from different processors, because type conversion can be performed when necessary. Thus two processors may represent, say, an integer in different ways, but MPI processes on these processors can use MPI to send integer messages without being aware of the heterogeneity.

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

Table 4: Basic C datatypes in MPI.

More complex datatypes can be constructed at run-time. These are called *derived* datatypes and are built from the basic datatypes. They can be used for sending strided vectors, C structs, etc. The construction of new datatypes is described later.

2.2.4.3 Collective Communication

MPI provides a variety of routines for distributing and re-distributing data, gathering data, performing global sums, etc. This class of routines comprises what are termed the “collective communication” routines, although a better term might be “collective operations.” What

distinguishes collective communication from point-to-point communication is that it always involves *every* process in the specified communicator (by which we mean every process in the group associated with the communicator). To perform a collective communication on a subset of the processes in a communicator, a new communicator has to be created. The characteristics of collective communication are:

- Collective communications cannot interfere with point-to-point communications and *vice versa* – collective and point-to-point communication are transparent to one another. For example, a collective communication cannot be picked up by a point-to-point receive. It is as if each communicator had two sub-communicators, one for point-to-point and one for collective communication.
- A collective communication may or may not synchronize the processes involved.
- As usual, completion implies the buffer can be used or re-used. However, there is no such thing as a non-blocking collective communication in MPI.
- All processes in the communicator must call the collective communication. However, some of the routine arguments are not significant for some processes and can be specified as "dummy" values (which makes some of the calls look a little unwieldy!).
- Similarities with point-to-point communication include:
  - A message is an array of a particular datatype.
  - Datatypes must match between send and receive.
- Differences include:
  - There is no concept of tags.
  - The sent message must exactly match the specified receive buffer.

2.2.4.3.1 Barrier Synchronization

This is the simplest of all the collective operations and involves no data at all.

```
int MPI_Barrier(MPI_Comm comm)
```

`MPI_Barrier` blocks the calling process until all other group members have called it. A typical use of `MPI_Barrier` occurs if in one phase of a computation, all processes participate in writing a file, the file being used as input data for the next phase of the computation. Therefore, no process should proceed to the second phase until all processes have completed phase one.

2.2.4.3.2 Broadcast, Scatter, Gather, etc.

This set of routines distributes and re-distributes data without performing any operation on the data. The routines are shown schematically in Figure 4. A broadcast has a specified root process and every process receives one copy of the message from the root. All processes must specify the same root (and communicator).

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

The `root` argument is the rank of the root process. The `buffer`, `count`, and `datatype` arguments are treated as in a point-to-point send on the root and as in a point-to-point receive elsewhere.

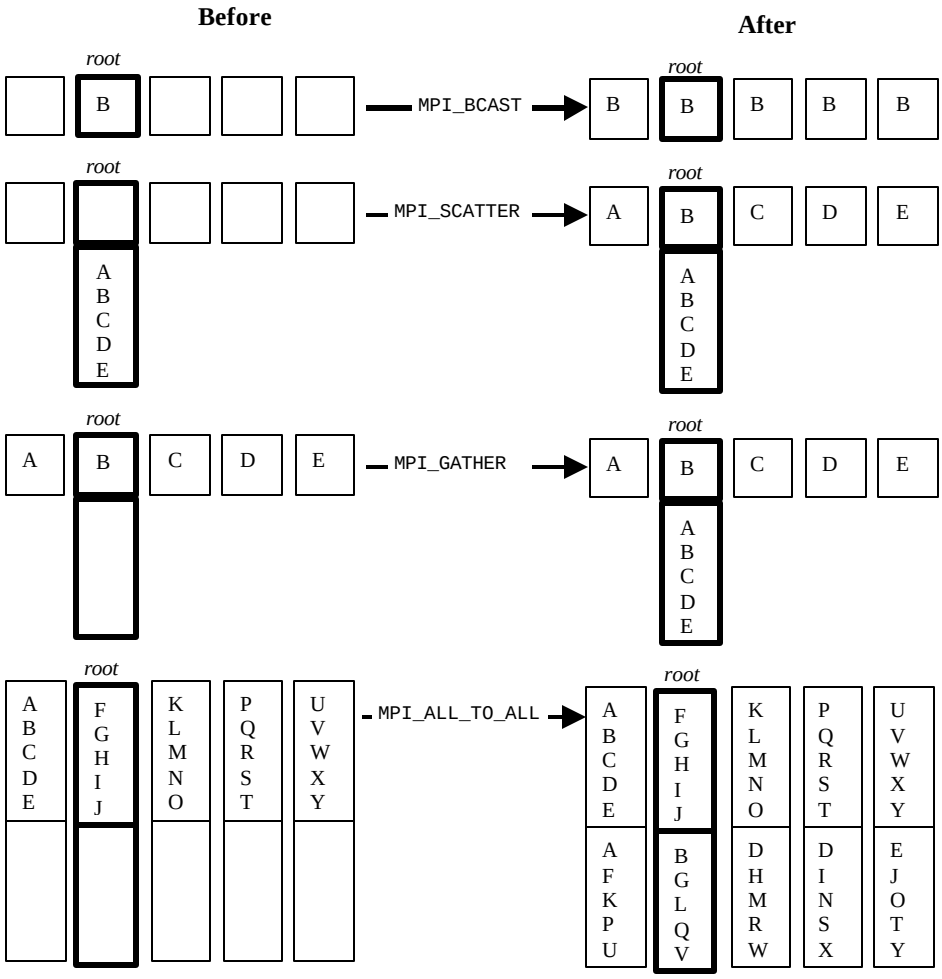


Figure 4: Schematic illustration of broadcast/scatter/gather operations. The boxes represent processes, with ranks 0 to 4, and buffer space and the letters represent data items. Receive buffers are represented by the empty boxes on the "before" side, send buffers by the full boxes.

2.2.4.3.3 Global Reduction Operations

You should use global reduction routines when you have to compute a result which involves data distributed across a whole group of processes. For example, if every process holds one integer, global reduction can be used to find the total sum or product, the maximum value or the rank of the process with the maximum value. The user can also define his or her arbitrarily complex operators. One can find a complete description of global reduction operators in the MPI document [2].

2.2.4.4 Derived Datatypes

One of MPI's main features is its use of a datatype associated with every message. Specifying the length of a message as a given count of occurrences of a given datatype is more portable than using length in bytes, since length of given types may vary from one machine to another. It also allows MPI to provide translations between machine formats.

Derived datatypes are created at run-time. Before a derived datatype can be used in a communication, the program must create it. This is done in two stages.

- **Construct the datatype**. New datatype definitions are built up from existing datatypes (either derived or basic) using a call, or recursive series of calls, to the following routines: `MPI_Type_contiguous`, `MPI_Type_vector`, `MPI_Type_struct`, etc. (This list is not complete.)
- **Commit the datatype**. The new datatype is "committed" with a call to `MPI_Type_commit`. It can then be used in any number of communications. There is a complementary routine to `MPI_Type_commit`, namely `MPI_Type_free`, which marks a datatype for de-allocation.

2.2.4.4.1 Construction of Derived Datatypes

Any datatype is specified by its *type map*, that is a list of the form:

basic datatype 0	displacement of datatype 0
basic datatype 0	displacement of datatype 0
…	…
basic datatype *n*-1	displacement of datatype *n*-1

The displacements may be positive, zero or negative, and when a communication call is made with the datatype, these displacements are taken as offsets from the start of the communication buffer, i.e., they are added to the specified buffer address, in order to determine the addresses of the data elements to be sent. A derived datatype can therefore be thought of as a kind of *stencil* laid over memory.

Of all the datatype-construction routines, here we will only describe `MPI_Type_vector` and `MPI_Type_struct`. The others are broadly similar and the interested programmer is referred to the MPI document [2] or to [1,3]. Appendix A provides an example.

2.2.4.4.2 MPI_Type_vector

`MPI_Type_vector` is a constructor that allows replication of a datatype into locations that consist of equally spaced blocks. Each block is obtained by concatenating the same number of copies of the old datatype. The spacing between blocks is a multiple of the extent of the old datatype.

```
int  MPI_Type_vector(int  count,  int  blocklength,  int  stride,
                     MPI_Datatype oldtype, MPI_Datatype *newtype)
```

The new datatype `newtype` consists of `count` blocks, where each block consists of `blocklength` copies of `oldtype`. The elements within each block have contiguous displacements, but the displacement between every block is `stride`. Figure 5 illustrates a call to `MPI_Type_vector` with:

```
count = 2
stride = 5
blocklength = 3
```

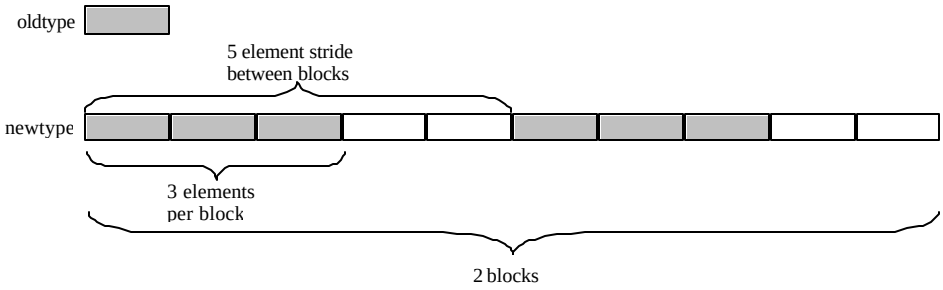


Figure 5: Illustration of a call to `MPI_Type_vector`.

2.2.4.4.3 MPI_Type_struct

`MPI_Type_struct` is very general. It allows us to describe a collection of data items of various elementary and derived types as a single datatype. It considers the data to be composed of a set of "blocks" of data, each of which has a count and datatype associated with it, and a location given as a displacement.

```
int MPI_Type_struct(int count, int *array_of_blocklengths,
                    MPI_Aint *array_of_displacements,
                    MPI_Datatype *array_of_types,
                    MPI_Datatype *newtype)
```

The new datatype `newtype` consists of a list of count blocks, where the *i* th block in the list consists of `array_of_blocklengths[i]` copies of the type `array_of_types[i]`. The displacement of the *i* th block is in units of *bytes* and is given by `array_of_displacements[i]`. Figure 6 illustrates a call to `MPI_Type_struct` with:

```
count = 2
array_of_blocklengths[0]  = 1
array_of_types[0] = MPI_CHAR
array_of_displacements[0] = 0
array_of_blocklengths[1]  = 3
array_of_types[1] = MPI_DOUBLE
array_of_displacements[1] = 1
```

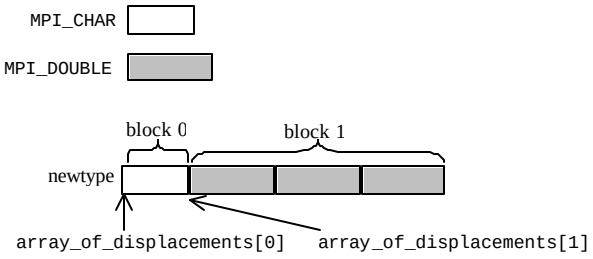


Figure 6: Illustration of a call to `MPI_Type_struct`.

2.2.4.5 Virtual Topologies

A virtual topology is a mechanism for naming the processes in a communicator in a way that fits the communication pattern better. The main aim of this is to make subsequent code simpler. It may also provide hints to the run-time system, which allow it to optimize the

communication or even hints to the loader on how to configure the processes. However, any specification for this is outside the scope of MPI. For example, if your processes will communicate mainly with nearest neighbors in the fashion of a two-dimensional grid (see Figure 7), you could create a virtual topology to reflect this fact. What this gives you is the access to convenient routines which, for example, compute the rank of any process given its coordinates in the grid, taking proper account of boundary conditions, i.e., returning `MPI_PROC_NULL` if you go outside the grid. In particular, there are routines to compute the ranks of your nearest neighbors. The rank can then be used as an argument to `MPI_Send`, `MPI_Recv`, `MPI_Sendrecv`, etc. The virtual topology might also give you some performance benefit, but if we ignore the possibilities for optimization, it should be stressed that nothing complex is going on here. The mapping between process ranks and coordinates in the grid is simply a matter of integer arithmetic and could be implemented simply by the programmer, but virtual topologies may be simpler still.

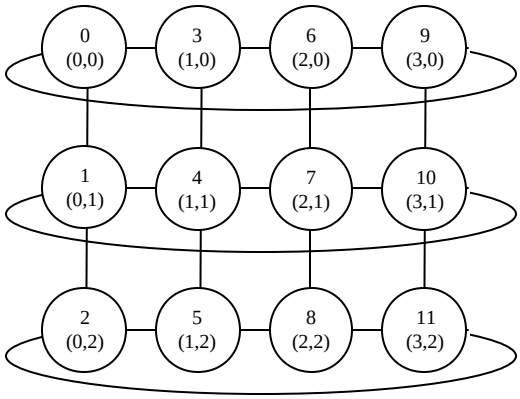


Figure 7: A virtual topology of twelve processes. The lines denote the main communication patterns, namely between neighbours. This grid has a cyclic boundary condition in one direction, e.g., process 0 and 9 are connected.

Although the virtual topology highlights the main communication patterns in a communicator by a “connection,” any process within the communicator can still communicate with any other.

As with everything else in MPI, a virtual topology is associated with a communicator. When a virtual topology is created on an existing communicator, a new communicator is automatically created and returned to the user. The user must use the new communicator rather than the old one to use the virtual topology.

You will only need *Cartesian* virtual topologies, suitable for grid-like lattices (with or without cyclic boundaries), in which each process is “connected” to its neighbors in a virtual grid. MPI also allows completely general graph virtual topologies, in which a process may be “connected" to any number of other processes and the numbering is arbitrary. These are used in a similar way to Cartesian topologies, although of course there is no concept of coordinates. The reader is referred to the MPI document [2] or [1,3] for details.

2.2.4.5.1 Creating a Cartesian Virtual Topology

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int
*periods, int reorder, MPI_Comm *comm_cart)
```

MPI_Cart_create takes an existing communicator comm_old and returns a new communicator comm_cart with the virtual topology associated with it. The Cartesian grid can be of any dimension and may be periodic or not in any dimension, so a torus, rings, three-dimensional grids, etc., are all supported. The ndims argument contains the number of dimensions. The number of processes in each dimension is specified in the array dims and the array periods is an array of TRUE or FALSE values specifying whether that dimension has cyclic boundaries or not. The reorder argument is an interesting one. It can be TRUE or FALSE:

- FALSE is the value to use if your data are already distributed to the processes. In this case the process ranks remain exactly as in old_comm and what you gain is access to the rank-coordinate mapping functions.
- TRUE is the value to use if your data are not yet distributed. In this case it is open to MPI to renumber the process ranks. MPI may choose to match the virtual topology to a physical topology to optimize communication. The new communicator can then be used to scatter the data.

MPI_Cart_create creates a new communicator and therefore like all communicator creating routines it may (or may not) synchronize the processes involved.

2.2.4.5.2 Cartesian Mapping Functions

MPI provides a simple way to find the neighbors of a Cartesian mesh. The most direct way is to use the routine MPI_Cart_get. This routine returns both values of dims and periods argument used in MPI_Cart_create as well as an array coords that contains the Cartesian coordinates of the calling process.

```
int   MPI_Cart_get(MPI_Comm   comm,   int   maxdims,   int   *dims,
                   int *periods, int *coords)
```

The MPI_Cart_rank routine converts process grid coordinates to process rank. It might be used to determine the rank of a particular process whose grid coordinates are known, in order to send a message to it or receive a message from it (but if the process lies in the same row, column, etc. as the calling process, MPI_Cart_shift might be more appropriate). If the coordinates are off the grid, the value will be MPI_PROC_NULL for non-periodic dimensions, and will automatically be wrapped correctly for periodic. (The rank of the process itself within the communicator can of course be obtained with MPI_Comm_rank, see section 2.2.4.1.2).

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
```

The inverse function MPI_Cart_coords routine converts process rank to process grid coordinates. It might be used to determine the grid coordinates of a particular process from which a message has just been received.

```
int   MPI_Cart_coords(MPI_Comm   comm,   int   rank,   int   maxdims,
                      int *coords)
```

The maxdims argument is needed to specify the length of the array coords, usually ndims.

```
int   MPI_Cart_shift(MPI_Comm   comm,   int   direction,   int   disp,
                     int *rank_source, int *rank_dest)
```

This routine does not actually perform a “shift”. What it does do is to return the correct ranks for a shift that can then be included directly as arguments to MPI_Send, MPI_Recv,

MPI_Sendrecv, etc. to perform the shift. The user specifies the dimension in which the shift should be made in the direction argument (a value between 0 and ndims-1). The displacement disp is the number of process coordinates in that direction in which to shift (a positive or negative number). The routine returns *two* results: rank_source is where the calling process should receive a message *from* during the shift, while rank_dest is the process to send a message *to*. The value will be MPI_PROC_NULL if the respective coordinates are off the grid (see Figure 8). Unfortunately, there is no provision for a diagonal "shift," although MPI_Cart_rank can be used instead.

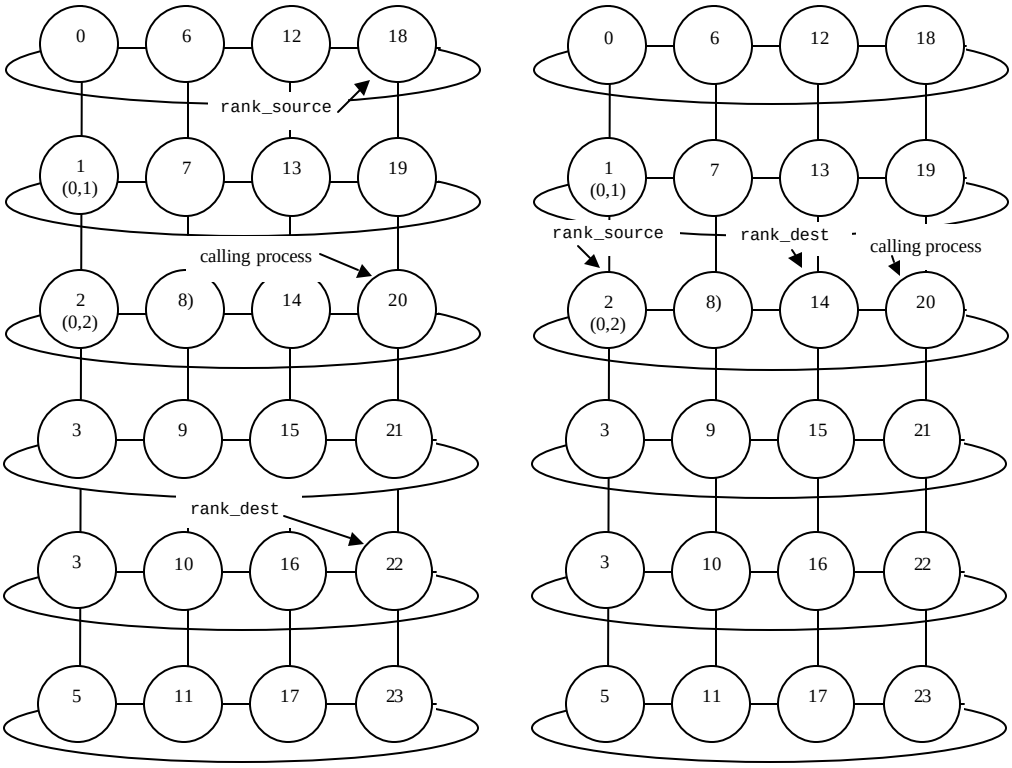


Figure 8: MPI_Cart_shift is called on process 20 with a virtual topology as shown. In the left picture MPI_Cart_shift was called with direction = 0 and with disp = 2 and in the right picture with direction = 1 and disp = -1. Note the effect of periodic boundary condition.

2.3 The OW cluster

For the lab course you will use the message passing interface (MPI) and develop programs using the SPMD paradigm. The development of your MPI programs will take place on the Euclides workstation cluster (edu-domain).

2.3.1 Shell Environment

On the Sun workstation cluster you simply append the following line to the .pkgrc file (located in your home directory):

```
mpich
```

to be able to use the MPI libraries and course software. Also make sure lamb daemon is up and running by executing lamboot command or add it to your .bashrc file or equivalent.

After you have set up your environment you need to log out and in again.

2.3.2 Compiling MPI Programs

The Message Passing Interface is implemented as a library that you link with your program. Definitions given for this library (constants and function prototypes) are included in the `mpi.h` header file. Furthermore, a number of functions and mis cellaneous definitions specific for assignment 1 of this course are provided in the apr library (`libapr.so`) and header file (`apr.h`).

In the practical assignments we provide so-called makefiles, which you can copy from the directory given in the assignment section. These makefiles specify the rules and commands, which should be used to compile a program. By only issuing the command `make`, which reads the `Makefile`, the proper executable is generated. We advise you to use these makefiles. In principle, the `Makefile` that is provided for assignment 1 will also work on the Lisa computer, and is easily adapted for the other assignments.

However if you prefer to do this yourself then you need to direct the compiler to the proper paths for the include files and libraries. The easiest and recommended way to do this is by using the `mpicc` script, which simply invokes the compiler, taking care of all MPI-specific arguments. The command line to compile the `wave.c` of assignment 1 on the ow-cluster would be

```
mpicc -g -I/home/<username> -R/home/<username> -L/home/<username>\
       -o wave wave.c –lapr -lm
```

The three paths `/home/<username>` and the library `-lapr` are specific for assignment 1 of this lab course and point to the location of the dynamically linked library and include file. The full command line, generated by `mpicc`, including all MPI specifics, is shown below:

```
cc -g -I/home/<username> -L/home/<username> -L$MPIHOME/lib \
       -I$MPIHOME/include -o wave wave.c –lapr -lmpich –lsocket \
       -lnsl -lm
```

We advise you, as best practice, to use `mpicc`. In this way compiling MPI programs can be done transparently on different platforms. For instance, the libraries `-lsocket -lnsl` are specific for the Sun workstations running Solaris, and can be different on other systems. The paths with the `$MPIHOME` component point to where to find the MPI library `-lmpich` and include file. The option `-g` includes debugging symbols in the executable.

2.3.3 Running MPI Jobs

An MPI job is normally started using the `mpirun` command. This command starts a number of processes with the SPMD paradigm. On a cluster of workstations you normally create a file named `.mpirun.machines` in your home directory in which you specify on which machines the MPI program should run by putting the hostnames of the machines on separate lines. A typical `.mpirun.machines` file, specifying e.g. 6 machines, could look like this:

```
ow102
ow103
ow104
ow105
ow106
ow107
```

To add this automatically you can by the command

```
$ rsh sremote /opt/etc/netgroup edu-linux \ cut –f 1
–d.>~/.mpirun.machines
```

The command:

```
mpirun -np 3 –machinefile $HOME/.mpirun.machines executable
        argument1 argument2
```

would start one copy of the program `executable` on the local machine and two other copies on the first two machines specified in the `.mpirun.machines` file and pass the arguments argument1 and `argument2` to each of them. You can force `mpirun` not to start a copy on the local machine by using the option `–nolocal`. The number after the option `-np` specifies the number of processes to be started. Note that MPI assumes the SPMD paradigm; each process runs the same executable program and receives the same arguments.[2]

It is recommended to add a shell alias that will automatically append the `–machinefile` argument. Users of `csh` or `tcsh` should add to their `.aliases` file:

```
alias mpirun ‘mpirun –machinefile $HOME/.mpirun.machines’
```

Users of bash should add to their `.aliases.sh` file (create if necessary):

```
alias mpirun=‘mpirun –machinefile $HOME/.mpirun.machines’
```

From now on we assume that you created this and that invoking `mpirun` automatically included the `–machinefile $HOME/.mpirun.machines` option.

There are two other options to `mpirun` that are of possible interest, namely the option `-gdb` or `-dbx` by which you indicate that the first process (with rank id 0 in the world communicator) must be started under the supervision of the `gdb` or `dbx` debugger respectively.

2.3.4 Testing your installation:

You can check whether your environment is set up correctly on the cluster of workstations in the edu-network by compiling and running a test program. First copy the file from `/usr/local/stud/apr/mpi` named `mpitestprg.c` to your own directory. Compile this file using the command:

```
mpicc -g -o mpitestprg mpitestprg.c
```

And now run the program using

```
mpirun -np 4 mpitestprg
```

The output should look something like:

```
This is process 0 on ow30.edu.wins.uva.nl
This is process 1 on ow100.edu.wins.uva.nl
This is process 2 on ow101.edu.wins.uva.nl
This is process 3 on ow102.edu.wins.uva.nl
```

2.4 The Lisa computer

In the autumn of 2004 the University of Amsterdam realized a 255 node Lisa cluster, which is installed at SARA in the Watergraafsmeer in Amsterdam. The cluster currently consists of 255 compute nodes. The nodes run Linux (Debian) and MPICH is available. Furthermore, a large suite of software is available. Finally, a batch processing systems, PBS, has been

[2] The arguments to the program are not valid before the `MPI_Init` function call.

installed. You must use this system to start your parallel jobs. For much more information please consult the web pages, http://www.sara.nl/userinfo/Lisa/.

You get an account on the Lisa cluster. If you have assigned for this course you should have, or will soon receive an e-mail announcing your account and initial password on the Lisa.

NOTE that the Lisa blue cluster is a production machine, used by researchers of the Computer Science department. Therefore, keep the load on the machine to an absolute minimum! Develop and test your programs on the local workstations, and only use the Lisa for real measurements and simulations. Also, *only* use the Lisa for this specific course. Your account will be disabled if you try to use the system for other jobs.

Although MPI is exactly the same on the Lisa, you do need to know some peculiarities. For details of MPI on the Lisa, look in detail at the web page. Here are some of the most important issues.

(1) *Change your system to tcshell by following commands*

$ ypchsh

Enter your password

Login shell[/bin/bash]: /bin/tcsh

(2) *Configure your ssh settings*

Following the instructions on

http://www.sara.nl/userinfo/lisa/usage/ssh-settings/index.html

(3) *Configuring your environment on the Lisa*

Add the following lines to your .cshrc file

source /etc/csh.login

source /usr/share/module/init/csh

module load gnu-mpich-ib

module load intel-ifort

Add the following lines to your .pkgrc file

mpich

(4) *Compiling*

Compile as usual, using `make`, or `mpicc`.

(5) *Executing*

Only batch jobs may be executed on the Lisa cluster. For this purpose you use the PBS system. Detailed information can again be found on the web pages. To run a job you need to prepare a batch file. An example of such a batch file is shown in Appendix B. You submit your job using qsub

```
$ qsub batch
```

Manually:

mpirun -local -np <number of processors> -hostfile $PBS_NODEFILE <program> <parameters>

The batch file contains all the information that the system has to know in order to run your job. You have to edit this file in order to change parameter values. For example, the number of processors is set with the line:

Request nodes

#PBS -l nodes=4

In this case, you will get 4 nodes. If you want to change this value, change 4 with the new value. The other parameter lines are equally self-explanatory. If your program has a different name, or if it has command line parameters, you can change these by editing the last line in the batch file.

You may check if your program has correctly been submitted by typing

```
$ qstat
```

You may remove jobs from the queue by invoking

```
$ qdel my_job_id
```

If you want to see which nodes of the cluster are currently in use, type

```
$ xpbsmon
```

In order to see the monitor, you must redirect the graphical output to the display in front of you. You may find it convenient to execute (from your local UNIX session):

```
% rxcmd lisa.sara.nl xpbsmon
```

You should use either `rxcmd` or ssh for your interactive sessions (ssh is strongly recommended, if you use ssh you must however set the DISPLAY variable to redirect the graphical output in a correct way to your local machine).

2.5 Data analysis

During the assignments you will have to do many kinds of data analysis, ranging from plotting simple one-dimensional graphs, to plotting 2-dimensional concentration fields around an object. Furthermore, you may need other ways of data analysis, depending on how you exactly go about in working out the assignments. You are completely free to choose any environment that you like. However, we advise you to invest some time in this, as it is an important tool for your lab work.

A very powerful tool that many of you may know is `gnuplot`. You may want to use this tool for your analysis. It is available on all Unix systems. For information on `gnuplot` you may

want to consult the files in `/opt/stud/pwrs/NEW/info` on `gene`. The first assignment, on the vibrating string, will show an example of using `gnuplot` (see Appendix C). Another powerful environment that is available on `gene` and `remote` is `Mathematica`. Most of you will have previous experience with this system. This system is also very handy in creating all kinds of plots and analyzing your data. In fact, the picture on the front page was created with `Mathematica`.

3 Assignments

The first assignment, the parallel matrix vector product, is meant to refresh your memory with respect to parallel computing, to learn the basics of MPI, and to get you going on the Lisa system. Just take some time for this assignment, and make sure that you really master all difficulties before moving to the next assignments. While doing this first assignment, you will learn all necessary theory for the following assignment during the lectures

As an indication (!), you should try to stick to the following schedule (in 2006):

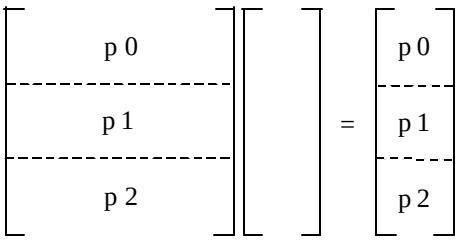
The parallel matrix vector product	week 1
Vibrating string	week 2, 3
Time dependent diffusion	week 4, 5
Jacobi iteration	week 6
Gauss-Seidel iteration	week 7

3.1 The Parallel Matrix Vector Product

Consider the matrix vector product

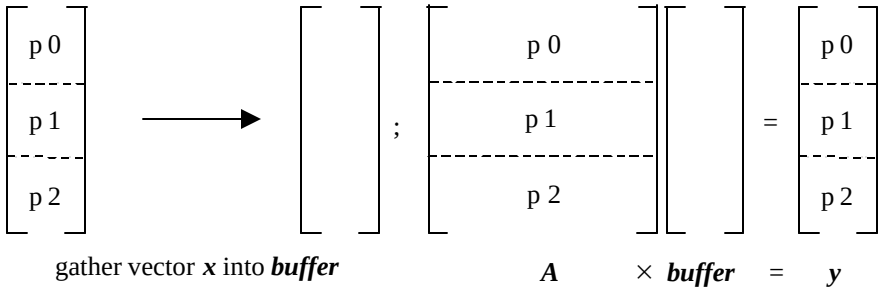
$$\boldsymbol{y} = \boldsymbol{A}\,\boldsymbol{x},$$

where $\boldsymbol{A}$ is an $n \times n$ matrix, and $\boldsymbol{x}$ and $\boldsymbol{y}$ are vectors of length n. A parallel version of this matrix vector product can be accomplished in many ways. The most straightforward way is by row-block decomposition. Here each processor receives a block of consecutive rows of the matrix and each processor has a copy of the complete argument vector $\boldsymbol{x}$ in memory. Using this data decomposition, each processor can, completely in parallel, calculate a part of the vector $\boldsymbol{y}$. This is drawn schematically in Fig. 1.



The parallel matrix vector product.

The dashing schematically shows the data decomposition of the matrix and both vectors. Note that the argument vector ***x*** must be available in the memory of each vector (in the picture schematically shown as a vector without dashed lines). The answer vector ***y*** is again distributed over all processors. Usually, the matrix vector product is part of a larger parallel program. Let us assume that in this program all vectors are distributed, like the vector ***y***. This means that the argument vector ***x*** is also distributed over all processors. Therefore, the full parallel matrix vector product should also contain a routine that first gathers the argument vector into memory of all processors (using an MPI global communication routine), and next the parallel computation is performed. The full parallel matrix vector product is drawn in Fig. 2.



the full parallel matrix vector product.

(A) Write down a time complexity formula for this parallel computation. Investigate the scalability. What is the influence of the grain size?

Hint : Use a simple linear model for point-point communication (as in Introduction to Parallel Computing) and assume that all floating point operations require the same time. You should also make an assumption about the gather operation. Clearly you don't know how this one is implemented in MPICH. Just assume that each processor sends chunks of data to all other processors one by one, and that every processor can do this completely in parallel. This is not a very realistic assumption, but the main conclusion from your time complexity model will apply to the real results. If you have time left you could investigate the behavior of the global communication routine in more detail.

(B) Write a parallel MPI program for the matrix vector product. You start with both the matrix and vectors initially distributed over all processors (that is, you generate this yourself). Write your program in such a way that you can easily change the matrix dimension n and the number of processors p. So, don't hard code the number of processors into your program. Use an MPI call to find out how many processors you have. That means that you write your program in such a way that the data decomposition is done completely automatic. Take special precautions in case n/p is not an integer. Device a load-balancing algorithm that enforces that the maximum difference in the number of rows that each processor receives in just one, i.e. processors receive $\lceil n/p \rceil$ or $\lceil n/p \rceil$ - 1 rows, with $\lceil\ \rceil$ the ceiling function. Remember that you had to solve a comparable load-balancing problem for the parallel wave equation in the Introduction Parallel Computing course. Spend some time to get all this absolutely right, as you will need it again in the next assignments.

Devise a method to test the correctness of your parallel program and explicitly report about this.

(C) Measure the scalability of your parallel matrix vector product (on the Lisa). Discuss your results, using the theory from (A).

(D) You may devise other data decompositions of the matrix. Theoretically analyze them and compare with the row-block decomposition. How do they compare, especially in the limit of large n?

3.2 The Vibrating string

The problem at hand is the numerical solution of a one-dimensional wave equation, which might describe the vibrations of a uniform string, the transport of voltage and current along a lossless transmission line, the pressure and flow rate of a compressible liquid or gas in a pipe, sound waves in gases or liquids, and optical waves. In our conceptual view, the domain of the wave equation is the physical extent of a string, and the displacement $\Psi(x,t)$ is a function of the distance x and the time t representing the unknown solution.

The one-dimensional wave equation is

$$\frac{\partial^2 \Psi}{\partial t^2} = c^2 \frac{\partial^2 \Psi}{\partial x^2}$$

The solution $\Psi(x,t)$ is the vibration amplitude expressed as a function of position and time. The problem becomes fully posed with the addition of boundary conditions on the spatial domain, and initial position and velocity distributions.

Numerical solution results in a uniform discretization of the spatial and temporal domains, and approximates the partial differential equation by finite difference expressions. The grid points of this problem consist of discrete nodal points at which Ψ is to be evaluated. By analogy with the example of Hadrian's Wall, discussed during the lectures, the string is decomposed into contiguous sections. We note the importance of the domain topology (which is one-dimensional for both Hadrian's Wall and the string) and the grain size of the decomposition.

3.2.1 Assignments

Be sure to first follow the instructions in the previous chapters!

1. *Complete* the vibrating string program. The template skeleton `wave.c` can be found blackboard. You can also find a handy `Makefile` there. Additional instructions about the usage of the apr library that has to be used and accompanying software, such as the visualization of the string, can be found in Appendix (C). Also refer to this appendix how to run the wave program.

   (A1) The code for part A1 deals with the function `getparms()` You can use this function only to get started but have to implement this function yourself eventually.

   The function getparms has to be called in the following way:

   ```
   getparms(argc, argv);
   ```

You have to write a *function* which does the same as the provided function `getparms()`.

In this function the first task within the MPI world communicator (which has rank-id zero in that communicator) gets the arguments provided at the command line. The function `parseargs()` that is available in the apr library and which you may use, parses these command line options and fills the parameter structure `parms` with the appropriate values. This first process must send the values from the structure stored in the `parms` variable to the other tasks which receive them into the `parms` variable.

The `parseargs()` function is used in the following manner:

```
if(parseargs(argc, argv)) {
    MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    exit(EXIT_FAILURE);
}
```

Explain why `MPI_Abort` should be used and not `MPI_Finalize`.

There are two ways by which the root process can send the structure to all tasks, using broadcasting or by sending the structure to each process separately. Explain which method in SPMD is the cleanest method, and which method may be the faster one.

(A2) Create the necessary program code in order to create a new communicator which uses a ring or line topology.

For this you have to find out how many MPI tasks are participating in the world communicator. Store this number in the `nprocs` variable. You can now initialize the ring/line topology using a call to `MPI_Cart_create`. Store the resulting topology in the variable `ring`. Then retrieve the rank of the MPI task and store this in the variable rank. Also find out the rank id's of the left and right processor in the topology and assign these values to the variables left and right.

(B) Complete the code for the assignment to the variables `npoints`, `nvalues` and `offset`. The number of data grid points on the total string is given in the parms structure. These grid data points have to be decomposed over the number of processing tasks, each processor will get a proportional amount of data points (*taking into account a possible load imbalance*). Since each MPI task only holds a part of the global string, it needs to know which part it has for initialization and visualization purposes.

(C) Write the code for the initialization of the string. The parameters in the `parms` structure dictate how the string should be initialized, which is either with a number of sine waves or plucked at a position in the string. After initialization the grid data points that are needed for the first computation need to be initialized.

(D) Complete the function compute() which should perform a single iteration over the grid data points, calculating the position of the string at time $t + Dt$ given the position of the string at time $t - Dt$ and t.

Ideally you could use the function `MPI_Sendrecv()` for exchanging the boundary grid data points. However, it is a good exercise to use separate `MPI_Send()` and `MPI_Recv()` functions. So, do both, and discuss the difference.

2. Show the correct working of the wave programs using a number of runs. Vary the parameters and use the waveplot program to create a number of dumps at some intervals. Which behaviour do you observe? Is it in agreement with your expectations?

3. The parallelism in the string problem is exploited using *spatial* domain decomposition, where the border points need to be communicated. However, the finite difference equation is put in the *temporal* domain; explain why parallelization of the application is best done in the spatial domain instead of the temporal domain.

4. It is known that the stability of the computation is limited for $0 \leq t \leq 1$, with t as defined in the syllabus *Introduction Parallel Computing*. Show experimentally that this is true. For different values of *Dt*, what is the effect in time of the non-stable behavior and what does this mean.

5. Scalability is a characteristic of a program as a function of both the number of processors as well as the data size. Measure the scalability of the vibrating string program on the network of Sun workstations. Interpret your results in terms of speedup and efficiency.

3.3 The Time Dependent Diffusion Equation

Consider the two-dimensional time dependent diffusion equation

$$\frac{\partial c}{\partial t} = D\nabla^2 c\,, \tag{1}$$

where $c(x,y;t)$ is the concentration as a function of the coordinates x and y and time t, and D is the diffusion constant. In all the assignments we will consider a square domain, and without loss of generality we assume that

$$0 \leq x \leq 1 \text{and} \; 0 \leq y \leq 1\,. \tag{2}$$

Furthermore, we always assume the following boundary conditions:

$$c(x, y=1; t) = 1 \text{and} \; c(x, y=0; t) = 0\,. \tag{3}$$

So, on the top of the domain the concentration is always equal to one, and on the bottom of the domain the concentration is always equal to zero. Furthermore, in the x-direction we will always assume period boundary conditions:

$$c(x=0, y; t) = c(x=1, y; t)\,, \tag{4}$$

or in general

$$c(x=x, y; t) = c(x=x+1, y; t)\,. \tag{5}$$

These boundary conditions are once more drawn in Figure 9.

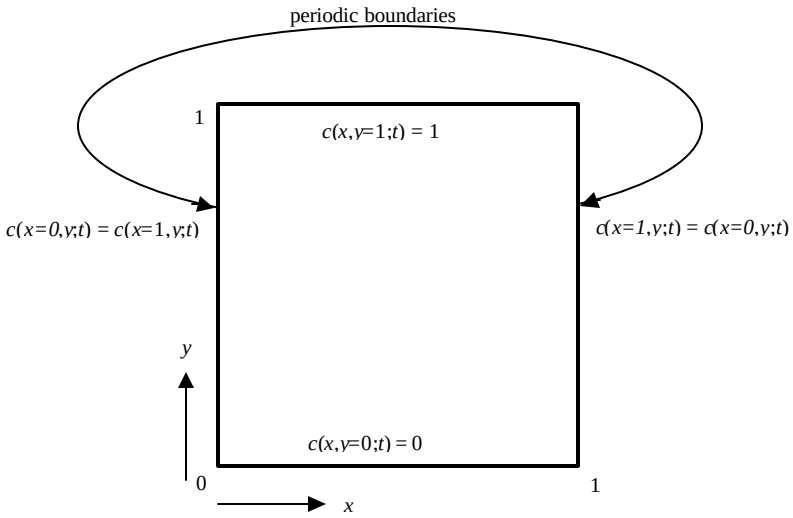


Figure 9: The computational domain and boundary conditions.

As an initial condition we take

$$c(x, y; t = 0) = 0 \text{ for } 0 \le x \le 1, 0 \le y < 1 . \tag{6}$$

Note that for this specific set of initial and boundary conditions that the solution only depends on the *y*-coordinate (because of symmetry) and on the time. This is a very useful feature to test the correctness of your (parallel) code. Furthermore, for this specific set of initial and boundary conditions the diffusion equation can also be solved analytically. The analytical solution, assuming $D = 1$, as a function of the *y*-coordinate are shown Figure 10 at a number of time stamps.

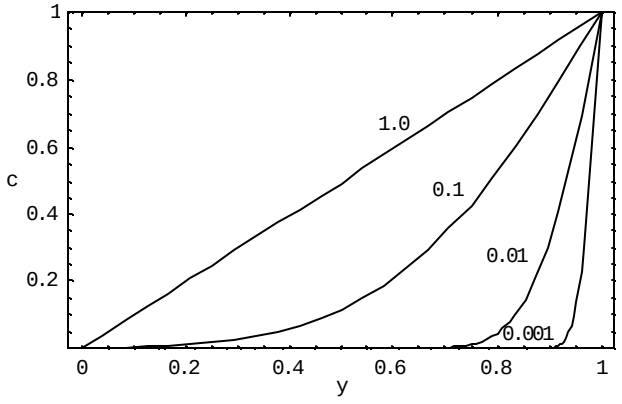


Figure 10: The analytical solution of the concentration, as a function of the *y*-coordinate, for *t* equal to 0.001, 0.01, 0.1, and 1 respectively. Here, $D = 1$ and the initial and boundary conditions are as in Eqs. 2 – 6.

Note that for $t \to \infty$ the concentration profile is simply a straight line, in this case

$$\lim_{t \to \infty} c(y, t) = y . \tag{7}$$

You can easily derive this yourself from the time-independent diffusion equation (try it). This is also a powerful check of the correctness of your simulation. For finite times you can also compare the simulation results with the exact solution.

The directory `/opt/stud/pwrs/NEW/datafiles` on `gene` contains the data files of the exact solutions of Figure 10. You will need these data to compare with your simulations later on.

Next the spatial and temporal domain are discretized. The x- and y-axis are divided in intervals with length dx. Assuming that we have N interval in the x- and y-direction we have

$$dx = \frac{1}{N}, \text{ and } x = idx,\ y = jdx \text{ where } i, j \in (0,1,2,\cdots N). \tag{8}$$

Furthermore we assume a small time interval dt and

$$t = kdt \text{ where } k \in (0,1,2,\cdots). \tag{9}$$

We now only want to find solution to the concentration at these discretized space – and time points. With the definition

$$c(idx,\ jdx;\ kdt) \equiv c_{i,j}^{k}. \tag{9}$$

and discretizing the diffusion equation using standard finite difference methods the following explicit scheme is easily derived.

$$c_{i,j}^{k+1} = c_{i,j}^{k} + \frac{dtD}{dx^2}\left[c_{i+1,j}^{k} + c_{i-1,j}^{k} + c_{i,j+1}^{k} + c_{i,j-1}^{k} - 4c_{i,j}^{k}\right]. \tag{10}$$

This scheme is stable if

$$\frac{4dtD}{dx^2} \le 1. \tag{11}$$

This determines the maximum time step δt that you can take.

In this assignment you implement a parallel finite difference simulation to solve the time dependent diffusion equation, using the finite difference scheme in Eq. 10 and subject to the boundary and initial conditions in Eqs. 2 - 6.

Due to the five point stencil every grid point only needs local information to update for a next time step. Therefore, as with the parallel wave simulation, a straightforward decomposition of the spatial domain is possible. However, in this case we have a two-dimensional spatial domain, and therefore many different decompositions are possible, such as a strip-wise or block-wise decomposition, see Figure 11.

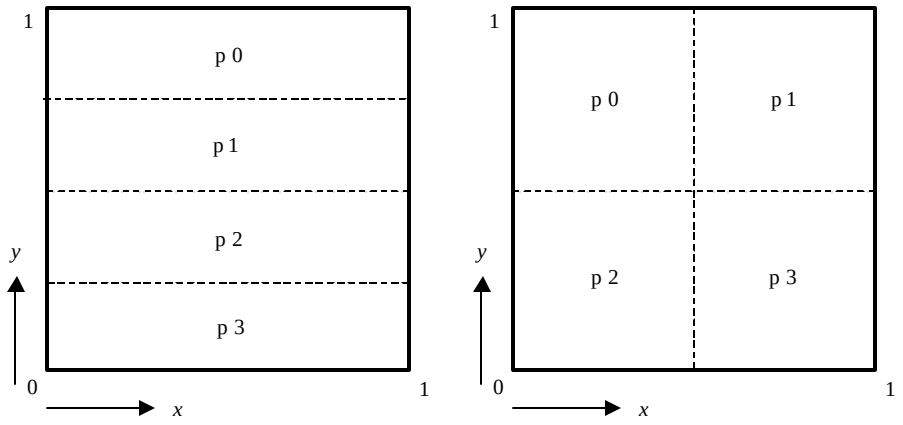


Figure 11: A strip wise and block wise decomposition of the square spatial domain, using in this case 4 processors.

(A) Derive theoretical time complexity formulas for a single iteration of the finite difference scheme for both a strip-wise and block-wise decomposition of the spatial domain. Discuss the differences that you observe. Which one is the best, and why ?

(B) Write a parallel program for the simulation of the two-dimensional time dependent diffusion equation discretized using the explicit finite difference formulation from Eq. 10. As the strip wise decomposition is easiest to implement, you can take that one. However, if you want to take the square decomposition, that's also allowed (this will require a bit more programming, but it *is* a good experience). In both cases, use virtual topologies (the MPI_Cart_ routines). Explain in detail how you handle the dependencies between the processors (both in terms of the data structures that you use, and in terms of communication between processors). The program much be such that the number N and the number of processors p is not hard-coded, and that you take precautions for good load-balancing.

Another issue is your output. You want to write your data to file (e.g. after every iteration, of maybe after every 100 iterations) such that you can analyze the data later on. The easiest way is to let every processor write data to a unique file (using `fwrite` or `fprintf`) and later on to join these files using e.g. `cat`. More sophisticated methods are off course possible, but are not necessary. In any case, this I/O operation will induce overhead that may (or may not) have negative effects of the efficiency. Try to measure this.

(C) Test the correctness of your parallel simulation

Hint: Compare to the analytic solutions.

(D) Measure the scalability of your parallel simulation. Decide exactly what you will measure, just a single iteration, the total program, or both! How does this compare to the theoretical analysis of (A) ?

In the final applications you will typically run your simulation on grids containing 256^2 up to 512^2 grid points. Therefore, when measuring scalability (on the Lisa!), the number N should be in the range from e.g. 32 to 1024 (and increasing by a factor of 2). Furthermore, try to measure the scalability using as much processors as you can get. You will however notice that getting more and more processors from the batch system is increasingly difficult. Therefore, try at least to measure the scalability using $p = 2, 4, 8, 16$. If you are able to get more processors, that's great, do it !

3.4 The Time Independent Diffusion Equation

Now assume that we are not very much interested in the time development of the concentration profile (i.e. the transient behavior), but only in the steady state. In that case it is possibly more effective to directly solve the time independent diffusion equation:

$$\nabla^2 c = 0 . \tag{12}$$

This is the famous Laplace equation. We again assume the same boundary conditions as in the previous section.

Taking the same spatial discretization as before, and applying again the same 5-points stencil for the second order derivatives, Eq. 12 transforms to the following set of finite difference equations:

$$c_{i,j} = \frac{1}{4}\left[c_{i+1,j} + c_{i-1,j} + c_{i,j+1} + c_{i,j-1}\right], \tag{13}$$

for all values of (i, j). Note that the superscript k is no longer present, because the time behavior is now suppressed. Many methods exist to solve the equations. We will concentrate on iterative methods, because they are potentially efficient and allow for relative easy parallelism.

3.4.1 The Jacobi Iteration

Using now the superscript (k) denoting the k-th iteration, the Jacobi iteration is immediately suggested from Eq. 13:

$$c_{i,j}^{(k+1)} = \frac{1}{4}\left[c_{i+1,j}^{(k)} + c_{i-1,j}^{(k)} + c_{i,j+1}^{(k)} + c_{i,j-1}^{(k)}\right]. \tag{14}$$

Note that Eq. 14 is easily derived from Eq. 10 by putting

$$\frac{dtD}{dx^2} = \frac{1}{4}, \tag{15}$$

i.e. the Jacobi iteration is nothing but the solution of the time-dependent equation using the scheme from Eq. 10 with the maximum allowed time step. This suggests that more efficient iterative methods are needed.

There is one noticeable difference between the Jacobi iteration and the solution of the time dependent equation, especially for the parallel implementation. In the time dependent case one defines the time step and the total time that should be simulated. In an iterative method one needs a *stopping condition*. This stopping condition typically is some global measure, that is, will typically induce some form of global communication in the parallel program. Assume that the stopping condition is such that the solution is assumed to be converged if for all values of (i, j)

$$\left|c_{i,j}^{(k+1)} - c_{i,j}^{(k)}\right| < d , \tag{16}$$

where d is some small number (say 10^{-5}).

(A) Develop and implement an efficient parallel stopping condition. Try to minimize the amount of communication as much as possible. Measure the overhead induced by the communication in the parallel stopping condition. Is it wise to perform the stopping

condition calculation every iteration, or should one define an interval (say after e.g. every 5 or 10 iterations) that one checks for convergence? Explain why, and propose a clever algorithm yourself.

Hint : this depends of course heavily on the grain size of your parallel computation and the relative cost of the stopping condition as compared to the cost of a parallel iteration. So, take that into account in your discussion. Clearly, for this you need the results from assignment (A) and (D) of the previous section.

(B) Using your parallel stopping condition, implement the parallel Jacobi iteration (re-using the software for the parallel time dependent simulation) and test it. Especially test the influence of the number d on the number of iterations and the quality of the solution (the linear dependence of the concentration on the *y*-coordinate, see Eq. 7).

3.4.2 The Gauss-Seidel Iteration

An improvement over the Jacobi iteration is the Gauss-Seidel iteration, where during the iteration a new value is used as soon as it has been calculated. Assuming that the iteration proceeds along the rows (i.e. incrementing i for fixed j), the Gauss-Seidel iteration reads

$$c_{i,j}^{(k+1)} = \frac{1}{4}\left[c_{i+1,j}^{(k)} + c_{i-1,j}^{(k+1)} + c_{i,j+1}^{(k)} + c_{i,j-1}^{(k+1)} \right]. \qquad (17)$$

Although the Gauss-Seidel iteration is not a big improvement over the Jacobi iteration (in terms of the amount of iterations needed for convergence) and only is a first step in introducing the Successive Over Relaxation method (SOR, see next section), the Gauss-Seidel iteration *does* change an enormous amount in the parallel algorithm. The inherent locality in the Jacobi iteration is presumably lost in Eq. 15. A re-ordering of the calculations is needed to regain the necessary data locality.

(A) Describe this new re-ordering, the Red/Black order that allows a parallel implementation of the Gauss-Seidel iteration. What is the consequence of this ordering in terms of the communication overhead per iteration? Adapt your time complexity model for this new situation. What happens?

(B) Implement the parallel Gauss-Seidel iteration using Red/Black ordering, test the correctness, scalability, and convergence behavior.

Appendix A: Example Use of Derived Datatypes in C

```
/* ********************************************************** *
* It is required to send resultPacket
* ********************************************************** */
struct {
   int nResults;
   double results[RMAX];
} resultPacket;
/* ********************************************************** *
* Set up the description of the struct prior to
* constructing a new type
* Note that all the following variables are constants
* and depend only on the format of the struct.  They
* could be declared `const'
* ********************************************************** */
#define RESULT_PACKET_NBLOCKS 2
MPI_Datatype resultPacketType;
int array_of_blocklengths[RESULT_PACKET_NBLOCKS] = {1, RMAX};
MPI_Datatype array_of_types[RESULT_PACKET_NBLOCKS] =
                                    {MPI_INT, MPI_DOUBLE};
MPI_Aint array_of_displacements[RESULT_PACKET_NBLOCKS];

array_of_displacements[0] = 0;
MPI_Type_extent(MPI_INT, &extent);
array_of_displacements[1] = extent;
/* ********************************************************
* Use the description of the struct to construct a new
* type, and commit.
* ********************************************************** */
MPI_Type_struct (RESULT_PACKET_NBLOCKS,
                 array_of_blocklengths,
                 array_of_displacements,
                 array_of_types,
                 &resultPacketType);
MPI_Type_commit (&resultPacketType);
/**********************************************************
* The new datatype can be used to send any number of
* variables of type `resultPacket'
**********************************************************/
count = 1;
MPI_Ssend(&resultPacket, count, resultPacketType, dest, tag, comm);
```

Appendix B: An example PBS batch file

Below is an example of a PBS batch file, to be used to submit job on the Lisa.

```
#####
# An example PBS batch file. We assume that we want to start
# an MPI program called simulate.
#####
#
# job name (default is name of pbs script file)
#PBS -N per_10
#
# request nodes
#PBS -l nodes=8
#
# request 2GB memory
#PBS -l mem=2gb
#
# shell to be used for this script
#PBS -S /bin/sh
#
# max. wall clock time (just get a reasonable amount of time)
#PBS -l walltime=1:00:00
#
# Request that regular output and terminal output go to
# the same file
#PBS -j oe
#
# send me mail when job begins
##PBS -m b
# send me mail when job end
##PBS -m e
# send me mail when job aborts (with an error)
##PBS -m a
# if you want more than one message, you must group flags on one
# line, otherwise, only the last flag selected executes:
#PBS -mbea
#
# do not rerun this job if it fails
#PBS -r n
#
# export all my environment variables to the job
#PBS -V
#
#
# Using PBS - Environment Variables
# When a batch job starts execution, a number of
# environment variables are predefined, which include:
#
# Variables defined on the execution host.
# Variables exported from the submission host with
#  -v (selected variables) and -V (all variables).
# Variables defined by PBS.
#
# The following reflect the environment where the user ran qsub:
# PBS_O_HOST    The host where you ran the qsub command.
# PBS_O_LOGNAME Your user ID where you ran qsub.
# PBS_O_HOME    Your home directory where you ran qsub.
# PBS_O_WORKDIR The working directory where you ran qsub.
# These reflect the environment where the job is executing:
```

```
# PBS_ENVIRONMENT
#Set to PBS_BATCH to indicate the job is a batch job, or
# to PBS_INTERACTIVE to indicate the job is a PBS interactive job.
# PBS_O_QUEUE   The original queue you submitted to.
# PBS_QUEUE     The queue the job is executing from.
# PBS_JOBID     The job's PBS identifier.
# PBS_JOBNAME   The job's name.
# PBS_NODEFILE  the name of the file contain the
#               list of nodes assigned
#               to the job (for parallel and cluster systems).
####
cd $PBS_O_WORKDIR
n=`wc -l < $PBS_NODEFILE`
echo job running on $n processors
echo nodes:
cat $PBS_NODEFILE
mpirun -np $n -machinefile $PBS_NODEFILE ./simulate
```

Appendix C : Libraries and Utilities for Assignment 1

The APR Library

The APR library, needed for the vibrating string assignment, contains some functions that are needed for the vibrating string program. The library is called `libapr.so` or `libapr.a` depending on the platform. It can simply be used by including the header file `apr.h` in user program code and linking using the `-lapr` flag. Current installation of the APR library is on blackboard. Download it and instruct your compiler using the `-I/home/<username>`, `-L/home/<username>`, and `-R/home/<username>` options to use these library.

The header file contains the `struct parm parms`; declaration, which defines the structure containing the runtime parameters for the program. The function `parseargs()` parses the command line arguments given to the wave program and fills in the structure. Only task 0 of an MPI program will parse the arguments. The function `getparms` uses the `parseargs` function to get the arguments passed to the program and distributes them to all other MPI processes. The functions `getparms` and `parseargs` rely on the definition of the `parms` structure, so this structure may not be redefined.

Running the Wave Program

The simulation parameters such as length of time to simulate are passed to the program using the command line. The following parameters can be passed to the wave program in random order:

- -s *n* initialize the string with *n* sine waves
- -p *p* initialize the string as a plucked string at position *p* ($0 < p < 1$)
- -n *n* the number of grid data points in the global string should be *n*
- -d *d* set the *Dt* parameter at *d*
- -t *t* set the time to simulation time at *t*
- -v *n* write every *n* iterations a representation of the string to a file which can be visualized

The -s and -p option are mutually exclusive and either one is required. The options -n, -d, and -t are required but option -v is optional. Option -v must not be used when doing timing measurements because of the large overhead.

The `mpirun` command can be used to run the wave program (or other MPI programs) on the Euclides workstation cluster (edu domain). The first argument to the `mpirun` command is the number of MPI tasks to start. The remaining arguments give the executable name and the parameters to pass to the program to run. The command:

```
mpirun -np 4 wave -p 0.1 -n 100 -t 10 -d 0.5 -v 5
```

runs the wave program on 4 workstations. The vibrating string with 100 grid data points is simulated for 10 simulated seconds using a time step of 0.5 seconds and every 5 of the 20 iterations the string is visualized. The string is initial a plucked string at grid point 10.

The Waveplot Utility

When the -v option is given to the wave program, a file called `string.<procid>` (where `<procid>` is the rank of the processor) is produced during runtime in directory `~/tmp/wave`. This directory will be created automatically if it does not exist yet. `string.<procid>` holds the position of the string points at the selected intervals. To visualize the output data, first run the `waveplot` utility (type `waveplot`). This will generate output files that can be visualized as follows:

go to the directory `~/tmp/wave`

type: `gnuplot`

type: `load 'wave.gnu'`

Postscript files are generated in `gnuplot` as follows:

type: `set output 'file-name.ps'`

type: `set terminal postscript`

type: `plot 'wave<n>.dat'` where `<n>` is the corresponding time step

Before each new simulation remove the old string files in `~/tmp/wave` *directory to avoid a wrong visualisation.*

The Fit Function

The fit function in the APR library can be used to fit a straight line to a set of data points. The parameters a and b in the formula $y(x)=a\ x+\ b$ are set to such values that they describe the data set as accurately as possible by minimizing the difference between the observed value and value from the function for all measured values using ζ squared method.

Wave Source Code Skeleton

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include <apr.h>

static double *curval;  /* current values of grid points on string */
static double *oldval;  /* previous time step values of data grid points */
static double *newval;  /* data grid point values for next time step */
static MPI_Comm ring;  /* mpi communicator for ring/line topology*/
static int nvalues;  /* number of values in array curval, oldval and newval */
static int npoints;  /* number of grid data points for local process */
static int nprocs;  /* number of processes/mpi-tasks in spmd implementation */
static int rank;  /* rank of this mpi process in ring/line topology */
static int left;  /* rank of left mpi process in ring/line topology */
static int right;  /* rank of right mpi process in ring/line topology */

struct parm parms; /* structure with parameters which describe initial
                    * string configuration and computation parameters;
```

```
                    * see further apr.h file for fields.
                    */

FILE   *fp;         /* file pointer for visualization                        */

static void
compute(void)
{
    double tau2; /* squared tau value, where tau equals c * dt / dx */
    tau2 = 1.0 * parms.delta / 1.0;
    tau2 = tau2 * tau2;

    ...
    ...    EXCERCISE PART D, EXCHANGE BOUNDARY POINTS, COMPUTE NEW GRID
    ...    DATA POINTS AND DO TIME ADVANCE
    ...
}

int
main(int argc, char *argv[])
{
    int dims[1], periods[1], coords[1], neighbour[1];
    int i, iter, niters, offset;

    /* initialize mpi */
    MPI_Init(&argc, &argv);
    /* rank process 0 in world communicator receives parameters on
     * command line, parses them and pass them to other processes.
     * the other mpi processes receive the parameters.
     */

    ...
    ...   EXCERCISE PART A1, for COMPUTER SCIENCE AND AI STUDENTS ONLY
    ...   BIS STUDENTS CAN SIMPLY PUT IN A CALL TO
    ...                   getparms(argc, argv);
    ...   FOR THIS PART
    ...

    /* get number of mpi processes (nprocs) and initialize ring/line
     * topology (the communicator stored in ring variable).  Find
     * out own rank and left and right processes ranks on ring.
     */
    ...
    ...   EXCERCISE PART A2, PUT IN INITIALIZATION CODE FOR RING/LINE
    ...   TOPOLOGY AND RELATED CODE.
    ...

    /* compute data decomposition;
     * - npoints should hold the number of grid data points on the string
     *   for the local computing node.
     * - nvalues is the number of elements in the array (which is npoints
     *   plus the neighbourhood.
     * - offset is the starting position of the local grid data points
     *   in the global grid data points.
```

```
 */
...
...   EXCERCISE PART B, COMPLETE ASSIGNMENTS FOR VARIABLES npoints
...   nvalues AND offset
...
npoints = ... ;
nvalues = ... ;
offset  = ... ;

/* allocation of dataset and initialize visualisation */
if(!(curval = malloc(sizeof(double) * nvalues)) ||
   !(oldval = malloc(sizeof(double) * nvalues)) ||
   !(newval = malloc(sizeof(double) * nvalues))) {
    fprintf(stderr,"%s: out of memory\n",argv[0]);
    MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    exit(EXIT_FAILURE);
}
if(parms.freq > 0)
    /* if frequency is zero, don't create visualization file */
fp = visualize_init(rank);

/* initialization of string, initialize all grid data points in the
 * array according to the parameters set in parms. I.e.; either a
 * sine or a plucked string with eithers a number of periods or
 * plucked position.
 * use variables offset and parms.ntotal to determin the local x
 * position.
 */
...
...   EXCERCISE PART C, INITIALIZE GRID DATA POINTS IN ARRAYS
...

/* computation loop, determin number of iteration based on simulation
 * time and time advance per iteration.  call visualization at determined
 * number of iterations and when loop is done.  don't call visualization
 * if not requested (parameter is equal to zero).
 */
niters = (int) ceil((double) parms.stime / parms.delta);
for(iter=0; iter<niters; iter++) {
    /* visualize string at the first iteration and then every t.freq
     * time steps.
     */
    if(parms.freq > 0 && iter % parms.freq == 0)
        visualize(fp, npoints, &curval[1], iter*parms.delta);
    compute();
}
if(parms.freq > 0)
/* visualize end */
    visualize(fp, npoints, &curval[1], iter*parms.delta);

/* finish visualization, deallocate memory and quit from mpi */
if(parms.freq > 0)
    fclose(fp);
```

```
    free(curval);
    free(oldval);
    free(newval);
    MPI_Finalize();
    exit(EXIT_SUCCESS);
}
```

Bibliograhpy

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

[2] Message Passing Interface Forum, “MPI: a message-passing interface standard”, June 1995. `http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html`.

[3] M. Snir, S.W. Otto, S. Huss-Ledermann, D.W. Walker, and J. Dongarra, *MPI: The Complete Reference*. Cambrigde, MA: MIT Press, 1995