UNIVERSITY OF BURGOS

Area of Electronic Technology

MPI Parallel Programming Fundamentals.

José María Cámara Nebreda, César Represa Pérez, Pedro Luis Sánchez Ortega
MPI Parallel Programming Fundamentals. 2015
Area of Electronic Technology
Electromechanical Engineering Department
University of Burgos

Introduction ... 1
Activity 0: A Simple Example ... 3
OBJECTIVES ... 3
THEORETICAL CONCEPTS ... 3
PRACTICAL CASE ... 5
QUESTIONS ... 6
FLOWCHART ... 6
Activity 1: Point to Point Communications ... 7
OBJETIVES ... 7
THEORETICAL CONCEPTS ... 7
PRACTICAL EXERCISE ... 9
QUESTIONS ... 9
FLOWCHART ... 10
Exercise 2: Collective Communications ... 11
OBJETIVES ... 11
THEORETICAL CONCEPTS ... 11
PRACTICAL CASE ... 13
QUESTIONS ... 13
FLOWCHART ... 14
Activity 3: Scattering & Reduction ... 15
OBJETIVES ... 15
THEORETICAL CONCEPTS ... 15
PRACTICAL CASE ... 17
QUESTIONS ... 17
FLOWCHART ... 18
Activity 4: Virtual Topologies ... 19
OBJETIVES ... 19
THEORETICAL CONCEPTS ... 19
PRACTICAL EXERCISE ... 20
QUESTIONS ... 20
FLOWCHART ... 21
Activity 5: Parallel I/O ... 22
OBJETIVES ... 22
THEORETICAL CONCEPTS ... 22
PRACTICAL EXERCISE ... 24
QUESTIONS ... 24
FLOWCHART ... 25
Activity 6: New communication modes ... 27
OBJETIVES ... 27
THEORETICAL CONCEPTS ... 27
RPRACTICAL EXERCISE ... 28
QUESTIONS ... 28
FLOWCHART ... 28
Activity 7: Derived data types ... 29
OBJECTIVES ... 29

THEORETICAL CONCEPTS 29
PRACTICAL EXERCISE 31
QUESTIONS 31
FLOWCHART 32
Activity 8: Dynamic Process Management 33
OBJETIVES 33
THEORETICAL CONCEPTS 33
PRACTIAL EXERCISE 34
QUESTIONS 34
FLOWCHART 34
Activity 9: Example of real application 35
OBJETIVES 35
THEORETICAL CONCEPTS 35
PRACTICAL EXERCISE 35
QUESTIONS 36
Activity 10: Performance Assessment 37
OBJECTIVES 37
THEORETICAL CONCEPTS 37
PRACTICAL EXERCISE 40
QUESTIONS 41
Appendix A: Installing DeinoMPI 42
Installation 42
Configuration 42
Launching Jobs 42
Graphic Environment 42
Deino MPI manual. Available at: http://mpi.deino.net/manual.htm 46
Appendix B: Project Configuration in Visual Studio 2010 47
Appendix C: Configuration of MS-MPI. 52

Introduction

Our interest will be focused on parallel programing for multicomputer MIMD machines. Our application programs will split into several processes and each one will have the potential capability to be executed on a different node of our cluster.

The processes created by the user will cooperate to achieve a common computational objective. The collaboration will be possible due to communication and synchronization tools provided by the programming environment. Communication is implemented in the form of message exchanging.

Most of the scenarios proposed admit a number of different parallel solutions. We should try to come up with the most advantageous in terms of system performance. To do so we must take into account:

- We will try to increase performance (execution time). To do so, we will try to squish the application's potential locality, that is, its capability to work with local data avoiding the need for much information exchange between processes.
- Another important point is "scalability". In a hardware environment, where the amount of available resources is unknown at programming time, the application must scale to make the most of the available resources at any time.

Parallel programing is not an easy job. The theory around the development of concurrent and parallel software is beyond the scope of this course but, we will provide some hints. Parallel programming, as well as sequential programming is a creative task; what is about to be exposed is nothing more than a series of steps we recommend to follow when facing a parallelization. Let's split up the process in 4 steps:

- **Fragmentation**: this initial step is meant to find potential parallel structures within the problem to be solved. As a first approach, we may try to decompose the job in as many small parallel tasks as possible. Two criteria can be followed to carry out this decomposition:
  - The functional way: seeks for possible divisions in the job to be carried out by paying attention to its nature.
  - The data way: pays attention to the nature of the data to be processed trying to decompose them into the smallest chunks.
- **Communication**: once identified potential parallel tasks, communication needs between them must be analyzed.
- **Binding**: given that the cost of communications is high in terms of global execution time, the formerly identified tasks have to merge partially in order to balance computation and communication.
- **Mapping**: once the program's structure is settled, the recently generated processes have to be spread across the computers available. The strategy to be adopted differs according to

the fragmentation way. As a rule of thumb, there should be at least as many processes as computers are available in order to prevent anyone being unused. If all computers are equal, it would be recommendable to make as create as many processes as computers. If not, the most powerful computers can host a higher number of processes. It is also possible to assign processes to nodes on the go, thus balancing processors' load dynamically.

Activity 0: A Simple Example

OBJECTIVES

- Understand the structure of a parallel program.
- Learn the main concepts associated to MPI through a simple practical case.
- Conduct a first try in parallel programming and to understand its compilation and execution process.

THEORETICAL CONCEPTS

The structure of a MPI program.

The structure of a parallel MPI program is not different from any other C program. To make the MPI function set available we only need to include the corresponding header file: `<mpi.h>`. Now we are going to introduce the most basic functions and the order in which some of them have to be called.

There is a conflict of names between `stdio.h` and `mpi.h` for C++ affecting `SEEK_SET`, `SEEK_CUR`, y `SEEK_END` functions. MPI generates them within its name space but, `stdio.h` defines then as integer. Using `#undef` by bring about some conflicts with other included libraries, such as `iostream`, thus triggering a fatal error. To sort this out `#undef` can be introduced before `#include<mpi.h>`:

```
#include <stdio.h>
#undef SEEK_SET
#undef SEEK_CUR
#undef SEEK_END
#include<mpi.h>
```

also `#include<mpi.h>` before `#include<stdio.h>` or `iostream`:

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

In `<mpi.h>` all the functions we use are referenced but some of them must be called according to a certain sequence.

The first MPI function must be:

```
int MPI_Init(&argc, &argv[])
```

Command line initialization parameters are included as function parameters. They need to be declared in function `main( )`.

The last MPI function on the program must be:

```
int MPI_Finalize()
```

It has no parameters. All other MPI functions will be placed between these two.

As a last remark we have to mention that LINUX systems require a `exit (0)` or `return 0` ending of the code so function `main` must be declared as `int`. This is not mandatory in Windows systems but be keep it for the sake of compatibility.

Communicators

A communicator is a virtual entity that includes a number of processes capable of communicating within the communicator. It is mandatory for two processes to communicate that they belong to a common communicator. However, any process may belong to several communicators.

Within the communicator a process is identified by an integer number called “rank”.

The communicator itself is identified by a name. In every MPI program a default communicator including all processes is generated: MPI_COMM_WORLD. Additional communicators can be created by the user but this is out of the scope of this early exercise.

Some useful information can be retrieved from the communicator:

- **Rank**: in most cases, launched processes need to know their own ranks in order to identify the specific tasks they have to carry out. Imagine there is a process in charge of delivering workload to others and gathering results from them. Since the program’s code is seen the same by all processes they at least need to know if they are the one to deliver or working rest. Usually is the 0 ranked processes the one to behave as a master; the rest (and maybe the 0 as well) do the hard work. The MPI function that returns the local rank is:

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

  The input parameter `comm` is the communicator’s name. It is a `MPI_Comm`, variable type (MPI exclusive). The output parameter `*rank` points to the integer variable holding the communicator’s rank.

- **Size**: it is usually useful to know the communicator’s size, that is, the number of processes belonging to it. It is commonly used to split up workload between processes. The corresponding MPI function is:

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

  Output parameter `*size` returns the requested value.

The following function may also be useful. It returns the computer’s ID:

```
int MPI_Get_processor_name(char *name, int *resultlen)
```

The first output parameter points to a string containing the computer’s name. The second one, `resultlen`, returns the number of characters in the string.

PRACTICAL CASE

In this introductory exercise we provide the whole program's code. Its purpose is to fully understand the use of the elementary MPI functions and the process to write, compile and execute a MPI program.

It is the typical "Hello world" program but in a parallel manner this time:

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

int main (int argc,char *argv[])
{
      int myrank,size;
      int length;
      char name[10];

      MPI_Init (&argc,&argv);
      MPI_Comm_rank (MPI_COMM_WORLD,&myrank);
      MPI_Comm_size (MPI_COMM_WORLD,&size);
      MPI_Get_processor_name (name,&lenght);

      printf ("[Computer %s]> Process %d out of %d: Hello World\n", name,
myrank, size);
      fflush(stdout);

      MPI_Finalize();
      return 0;
}
```

REMARKS:

1. *Remember to write* `#include<mpi.h>` *before* `#include<stdio.h>` *to prevent the name conflicts already explained.*
2. *Function* `fflush(stdout)` *intends to clean up the standard output (display results) before resuming program execution.*

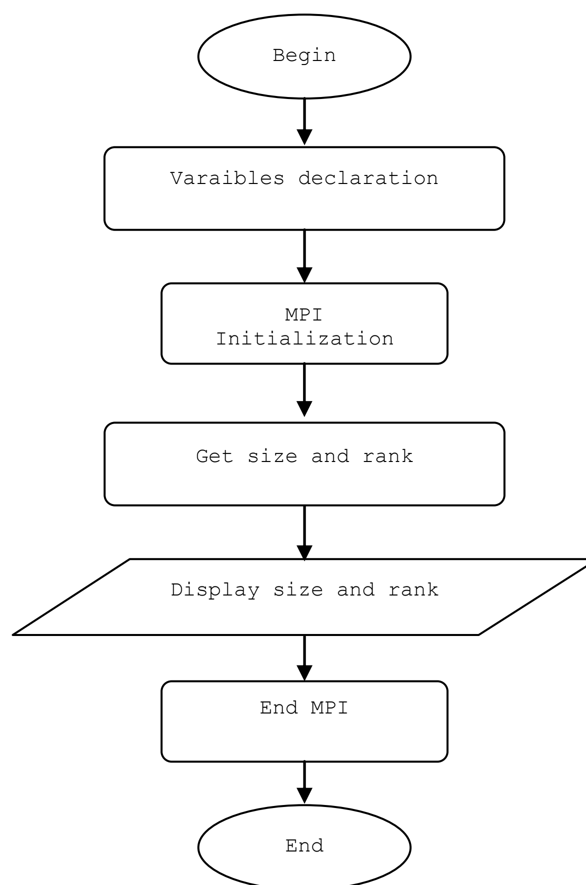
In this course we use Ms Visual Studio to write and compile programs. Appendices B & C provide detailed configuration information for Visual Studio 2005 & 2010 respectively.

Executable application programs must be launched from the DeinoMPI environment (or any other MPI launcher) already installed at the lab. Appendix A describes the use of this graphical launcher.

QUESTIONS

- After seeing the results, is it possible to guess how Rank is assigned to processes within the Communicator?
- Explain how system reacts when MPI is not initiated or finalized. Explain also the reaction to MPI function calls out of the space between these two events.

FLOWCHART



Activity 1: Point to Point Communications

OBJECTIVES

- To learn the different communication modes available in MPI and the structure of messages.
- Get to know the basic message passing functions.
- Make a first application program to deliver workload by means of message exchanging functions.

THEORETICAL CONCEPTS

Communication between processes

We can classify communication modes according to different criteria. These criteria are not alternative but complementary:

- According to the number of processes that generate and receive information we have communications:
  - *Point to point*: one process sends a message to another.
  - *Point to multipoint (broadcast)*: one process sends information to several others.
  - *Multipoint to point*: various processes send a message to the same one. This is physically not possible as messages would collide but MPI provides a function that virtualizes this possibility.
- According to the synchronization between emitter and receiver:
  - *Synchronous exchanges*: the emitter is idled until the receiver takes the message.
  - *Asynchronous exchanges:* the emitter saves the message in an internal buffer to be sent in background.
- According to processes blocking, it may occur:
  - Both emitter and receiver are idled until the send and receive operations have completed. This doesn’t mean that the message has been collected by the receiver. In both cases (send and receive) it is just necessary to have a copy of the message in a local buffer. Processes only stop in case the buffer has no more space.
  - Processes never stop regardless of the situation of the message. Unless outdated data is used by later instruction, there is no issue on the use of this mode.

Messages

MPI messages are composed of two main parts: envelope and body.

The envelope is integrated by:

- **Source**: Sender's ID (rank).
- **Destination**: Receiver's ID (rank).
- **Communicator**: Name of the communicator to which emitter and receiver belong.
- **Tag**: Number used by both processes to classify messages.

The body is integrated by:

- **Buffer**: Memory area where the information is temporarily stored, either for send and receive operations.
- **Data type**: Can be a simple data type: `int`, `float`, etc; or complex data types previously defined by the user.
- **Count**: Number of pieces of "Data type" to be exchanged.

MPI uses its own data types. These types are equivalent to the standard C types (see table below) but independent of the computer's system. Therefore it is not necessary to deal with different data formats when the hardware is heterogeneous.

Send & Receive functions.

MPI permits all communication modes already explained and some additional modes such as the "ready" and "buffered" ones. At this point we are going to introduce only the functions implementing the asynchronous blocking mode. They are:

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

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

`MPI_Send` requires a number of input parameters:

- `*buf`: points to the buffer (variable) representing the data to be sent.
- `count`: number of data units to be sent.
- `dtype`: MPI datatype.
- `dest`: receiver's rank.
- `tag`: message's tag.
- `comm`: communicator to which both sender and receiver belong.

`MPI_Recv` manages the following input parameters:

- `count`: number of data to be received.

- `dtype`: MPI datatype.
- `source`: emitter's rank.
- `tag`: message's tag.
- `comm`: communicator to which both sender and receiver belong.

Output parameters are:

- `*buf`: points to the buffer (variable) where the received data are to be stored.
- `*status`: returns some relevant information such as message's tag and source. It has to be declared as `MPI_Status` variable type.

Both functions return an error code in case they are not successfully completed.

Below we can find a quick reference table showing equivalent C and MPI data types:

MPI type	C type
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	Ninguno
MPI_PACKED	Ninguno

Table 1. MPI datatypes.

PRACTICAL EXERCISE

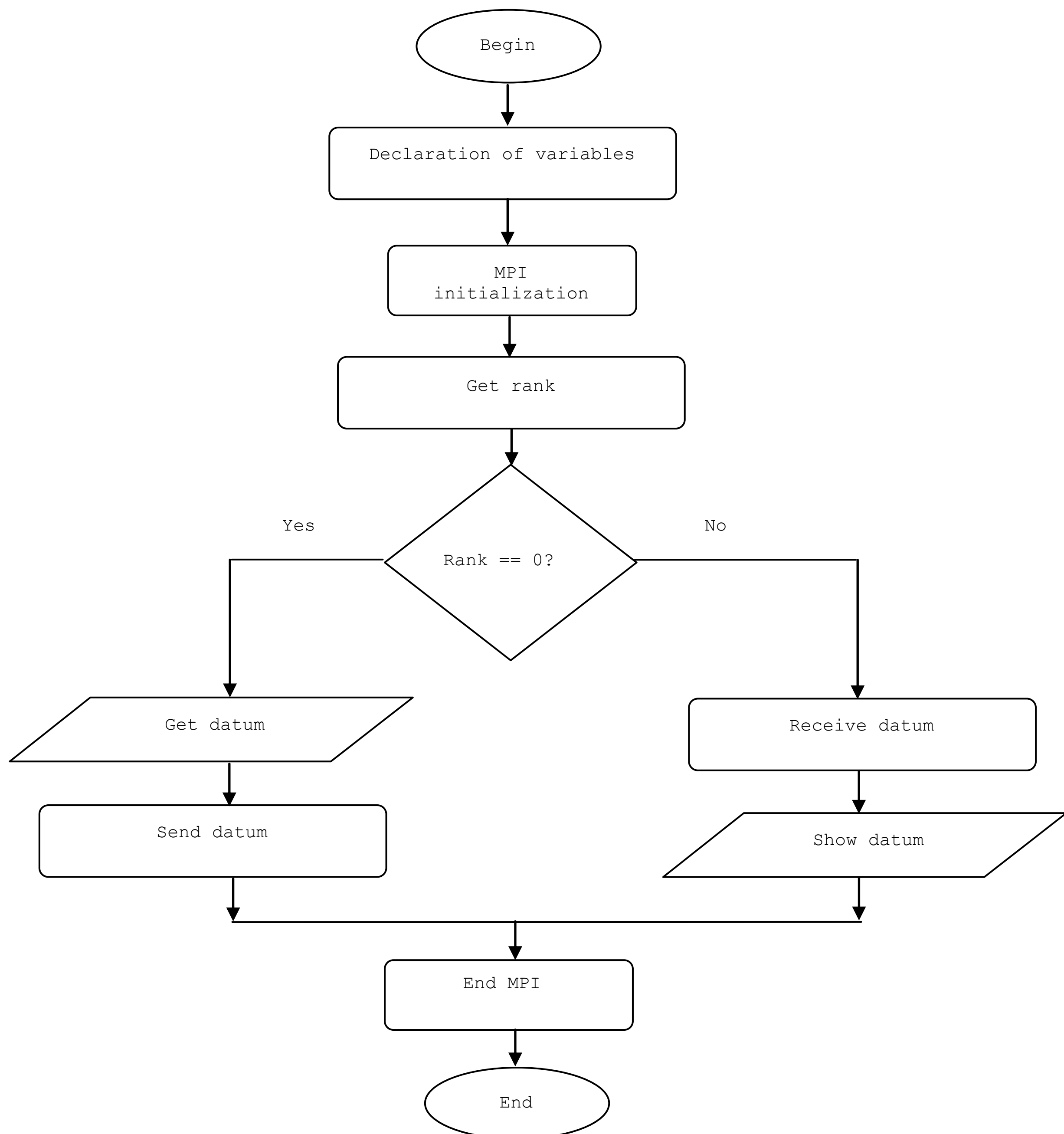
We will generate a simple program to test asynchronous blocking mode. 0 ranked process will request an integer through the standard input, then it will be sent to process 1 who will display the integer through the standard output.

QUESTIONS

- How could the same data be sent to various processes?

- How would be the receptions sorted in this case: by rank, by geographic proximity, by program sequence...?

FLOWCHART



Exercise 2: Collective Communications

OBJECTIVES

- ❖ Extend communication possibilities towards collective exchanges meant to make communication programing simpler.
- ❖ Think about different applications of collective communications.

THEORETICAL CONCEPTS

Collective Communications

On the previous exercise we exchanged messages between processes in the simplest possible way. In spite of its flexibility and simplicity it isn't in many cases the most adequate option. In most computing scenarios a single master process delivers workload to several slave processes and then gathers the results they all generate. It is possible to manage all necessary communication events via point to point functions, but if higher level functions are available there it is pointless.

Collective communication functions allow one process to send the same data to many others and to collect results from all of them in one step. This results in an extreme simplification of most parallel applications.

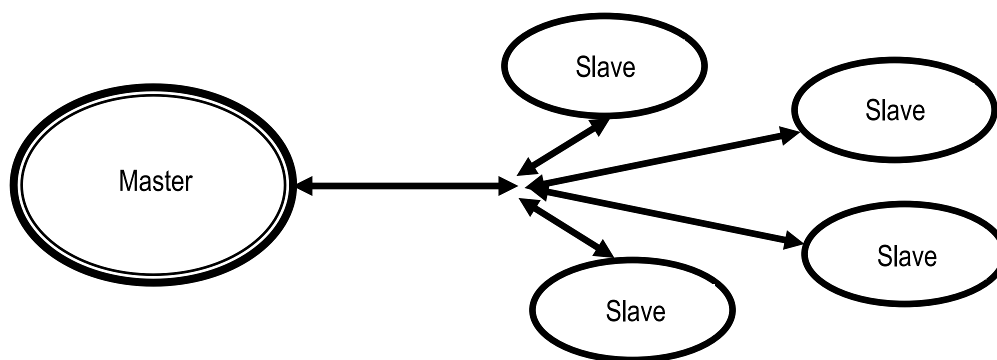


Figure 2.1. Master Slave scheme.

MPI implementation

Broadcast is a word used to describe information deliveries from one sender to all possible receivers. MPI provides a single function to manage this type of exchange:

```
int MPI_Bcast(void *buf, int count, MPI_Datatype dtype, int source,
    MPI_Comm comm)
```

We already know the meaning and use of all its parameters. Just notice that there is a *source* but no destination. It is important to remark that all processes see the function in their code exactly the

same, but only the one whose rank is equal to *source* is the sender and the rest will consider the function as a receive. Figure 2.2 shows how **`MPI_Bcast`** works. It copies the content of sender's `*buf` to the rest of the processes.

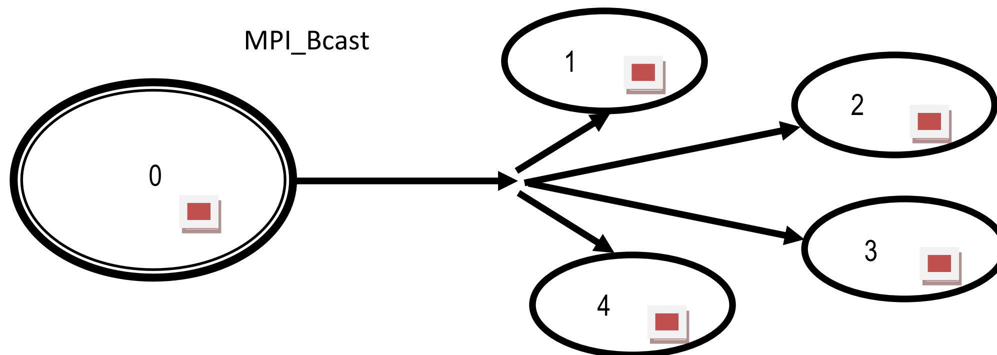


Figure 2.2. MPI_Bcast behavior.

Opposite to broadcasting is gathering. The results generated by all processes are reported to the master. All this data is arranged in the master's memory space according to the slave's Rank. It is also possible to collect results in all participating processes instead of only one. MPI functions providing this functionality are:

```
int MPI_Gather(void *sendbuf, int count, MPI_Datatype dtype, void
    *recbuf, int count, MPI_Datatype dtype, int dest, MPI_Comm comm)

int MPI_Allgather(void *sendbuf, int count, MPI_Datatype dtype, void
    *recbuf, int count, MPI_Datatype dtype, MPI_Comm comm)
```

The first one puts results only in receiver's *recbuf*. The second one leaves results in all processes' *recbuf*. In this case no *dest* parameter is needed. In all cases, data are arranged according to the sender's rank.

The value of *count* is the same for both the sender and the receiver, remarkably. It is the amount of date sent and the amount of data received from each process respectively.

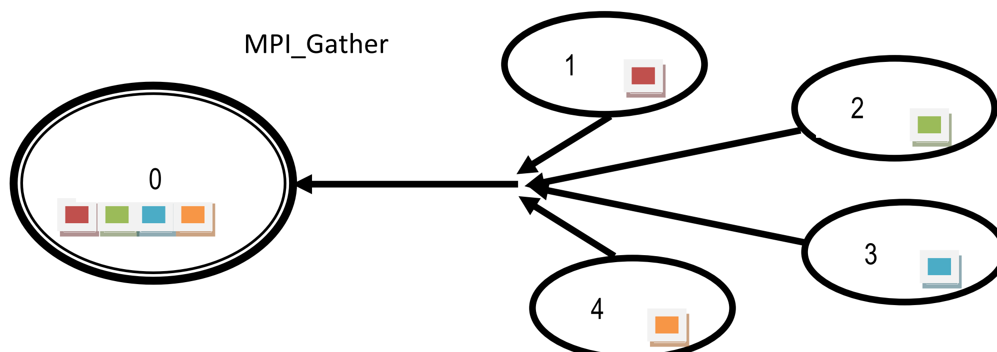


Figure 2.3. MPI_Gather behavior.

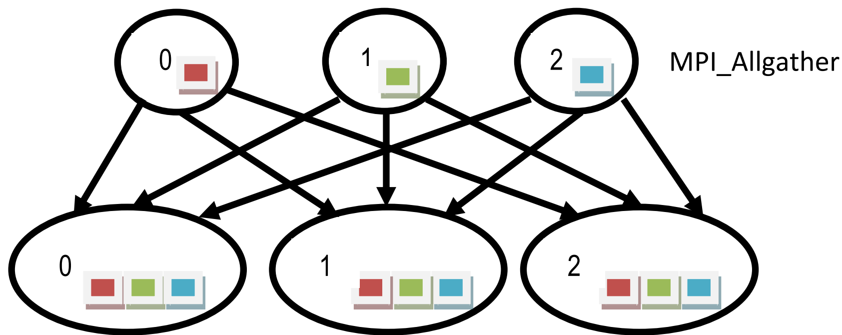


Figure 2.4. MPI_Allgather behavior.

PRACTICAL CASE

The program to be carried out will sum up two matrices (NxN). Process 0 will initialize both and broadcast them to the rest. Each process will sum the columns coinciding with its own rank. Process 0 will then collect all results, construct the final matrix and display it. Launch N process to avoid caring about size mismatches.

In addition to that we will measure execution time. MPI provides this functionality by means of the following function:

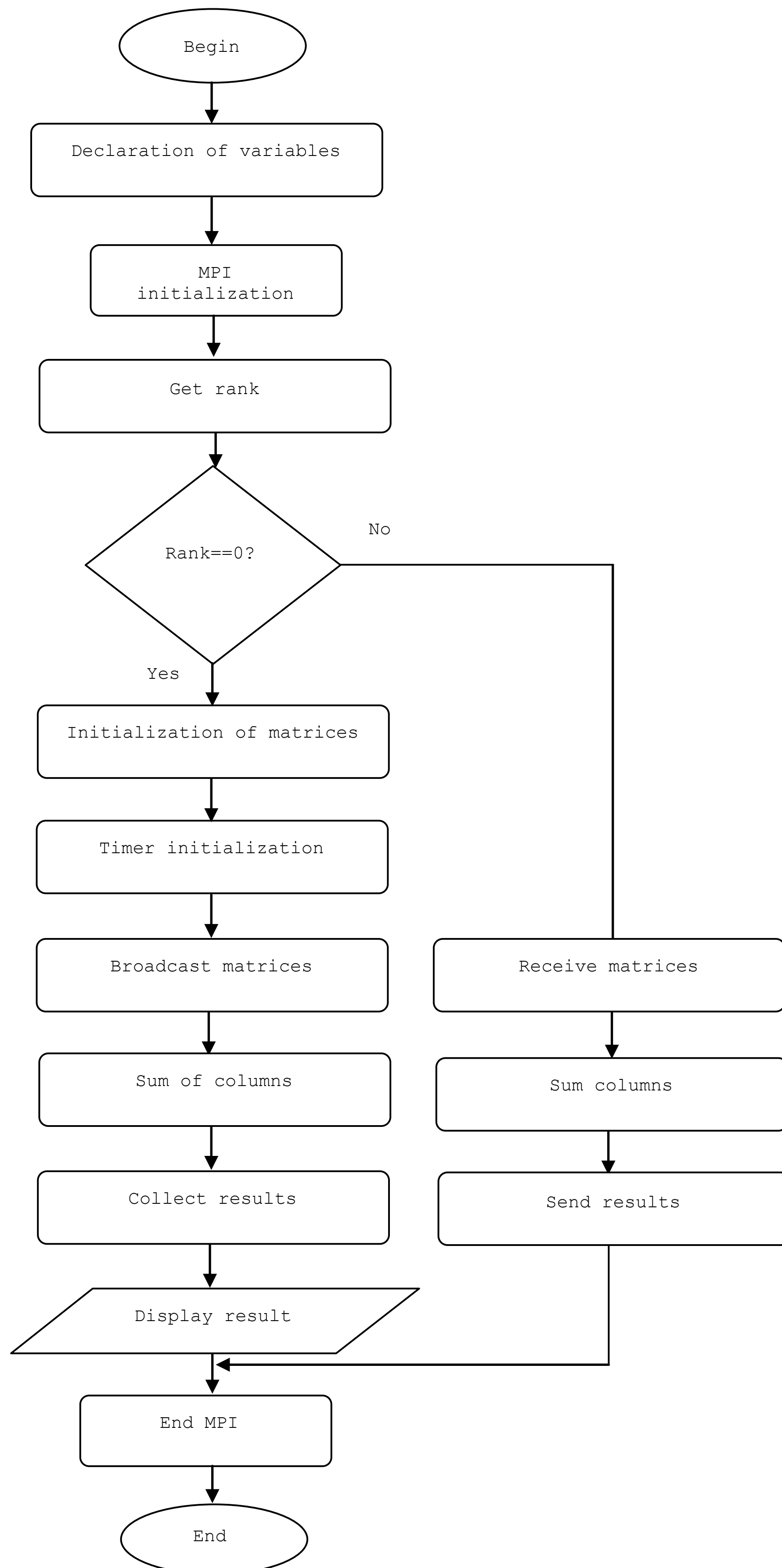
```
double MPI_Wtime (void)
```

it returns an absolute time so it has to be called at the beginning and end of the program and then subtract both times.

QUESTIONS

- Collective communications ease programming and make the code more straightforward. Would it be wide to believe that they shorten program's execution time?
- Explain what time is measured by the MPI function.
- Think of other time measurements that make possible to differentiate between communication and calculation times.

FLOWCHART



Activity 3: Scattering & Reduction

OBJECTIVES

- Try higher level collective functions.

THEORETICAL CONCEPTS

Scatter and reduction operations

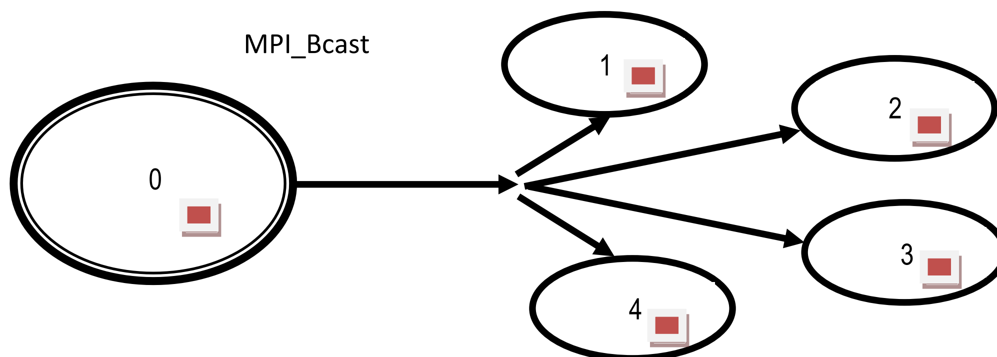
Collective communication functions provide noticeable advantages over point to point functions in many cases. Nevertheless it is still possible to increase the level of abstraction by means of the above mentioned operations. Unlike broadcasting, scattering splits up the workload between the available workers. Each process receives a chunk of the data to be processed. Reduction processes all results generated by the slaves performing a certain operation on them and thus generating a final result.

MPI implementation

MPI function providing scattering capabilities is:

```
int MPI_Scatter(void *bufsend, int count, MPI_Datatype dtype, void
   *bufrecv, int count, MPI_Datatype dtype, int source, MPI_Comm comm)
```

It works similarly to other previous functions. In this particular case, data from master process (`*bufsend`) are sent to slaves' memory (*bufrecv). Parameter "`count`" indicates how many are sent to each slave and its value is the same in both cases. Unlike **`MPI_Bcast`**, **`MPI_Scatter`** sends only a subset of data to each receiving process. As depicted in figure 3.1, **`MPI_Scatter`** this subset is distributed to the receivers according to their rank: first element (red) to process 0, second (green) to process 1, etc. It is important to notice that the first element is copied to process 0 even though it comes from the same process.



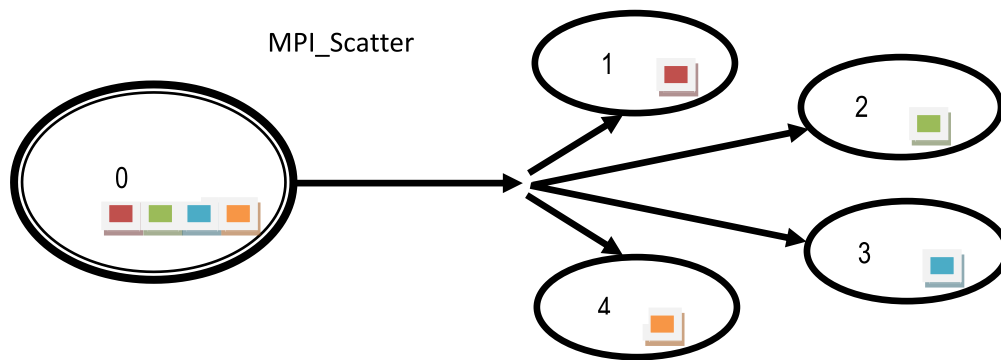


Figure 3.1. Diferences between functions MPI_Bcast & MPI_Scatter.

Reduction capability is provided by:

```
int MPI_Reduce(void *bufsend, void *bufrecv, int count, MPI_Datatype
   dtype, MPI_op op, int dest, MPI_Comm comm)
```

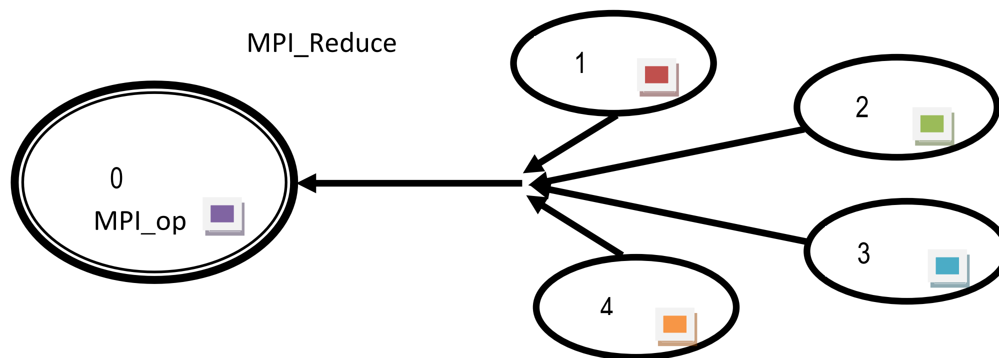


Figure 3.2. Reduction operation with MPI_Reduce.

Like **`MPI_Gather`**, this function collects information from slave processes. Each one of them sends `count` data stored in buffer `*bufsend` to the process whose Rank equals parameter `dest`. This process doesn't store the data in its buffer `*bufrecv`, but the result of an operation performed on these data (fig. 3.2). Parameter `MPI_op`, tells what the operation must be. Most common operations in MPI are described in Table 2.

Operación	Descripción
MPI_MAX	Highest value
MPI_MIN	Lowest value
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	AND
MPI_LOR	OR
MPI_LXOR	XOR
MPI_BXOR	Bit level XOR

MPI_MINLOC	Determines the Rank of the process providing the lowest value.
MPI_MAXLOC	Determines the Rank of the process providing the highest value.

Table 2. MPI operations.

PRACTICAL CASE

In this case, the program must calculate the scalar product of two vectors of whatever size:

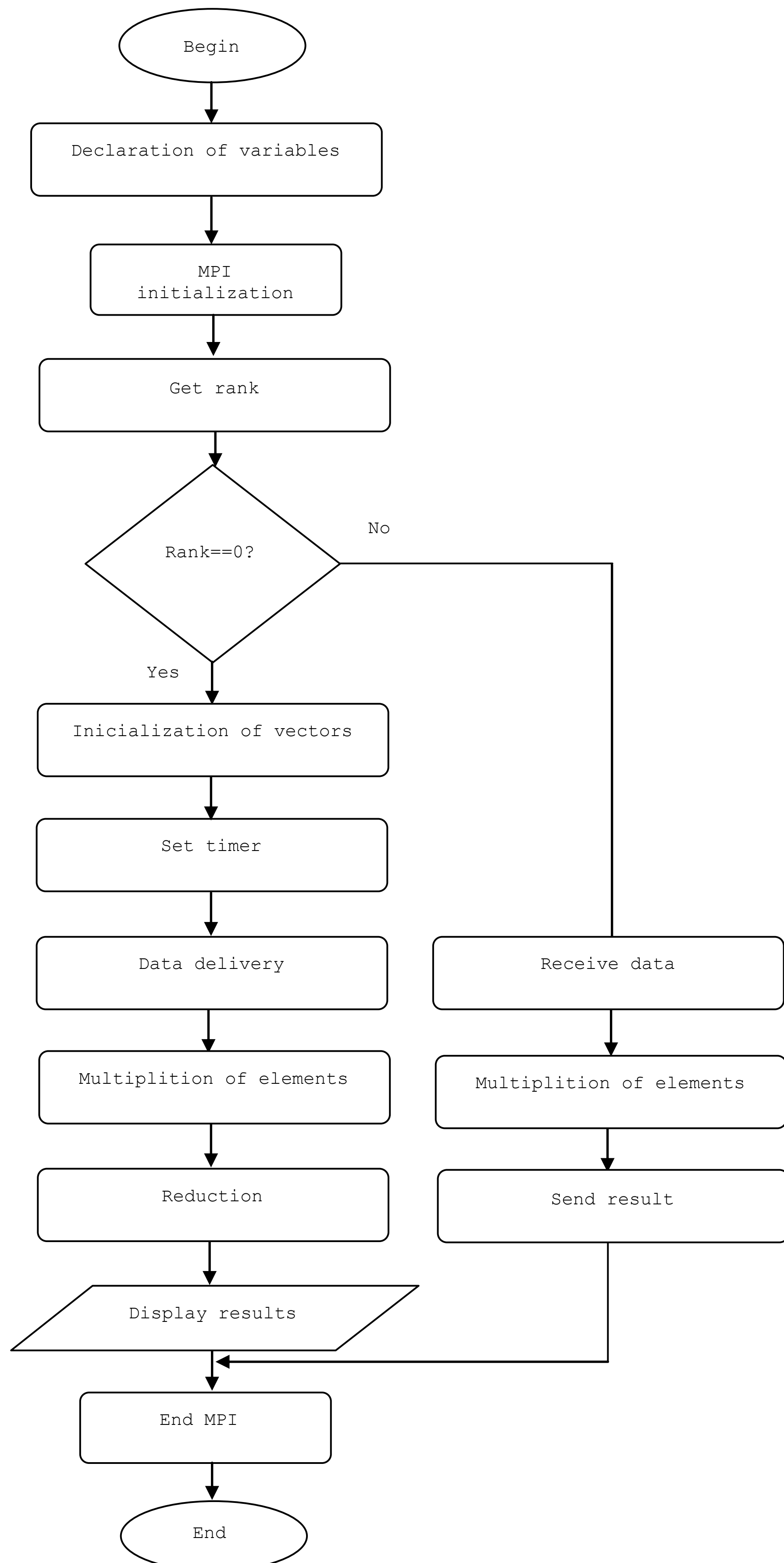
$$(x0, x1, \ldots, xn) \bullet (y0, y1, \ldots, yn) = x0 \bullet y0 + x1 \bullet y1 + \ldots + xn \bullet yn$$

It is suggested to build up the program in a way that each process calculates one term of the whole sum. The number of processes must be equal to the vector size in this scenario. Process 0 will initialize the vectors **x** & **y** and it will also dispatch their elements to the rest. Process 0 will eventually gather all individual terms of the sum and display the final result.

QUESTIONS

- Provided that the higher Rank processes have to perform more complex calculations, try to propose how to optimize system performance delivering more work to lower Rank processes.
- Is it possible to keep using the reduction function in this scenario?

FLOWCHART



Activity 4: Virtual Topologies

OBJETIVES

- A first approach to virtual topologies as a fundamental tool for the resolution of certain computing problems.

THEORETICAL CONCEPTS

MPI Cartesian topology

So far we've realized that all processes willing to Exchange messages must belong to a common communicator. Since the complexity of the previous exercises has been low, only the default communicator `MPI_COMM_WORLD` has been used. What we are going to do in this exercise is to modify the virtual distribution of the processes within a new communicator. By doing so we intend to make communicator's virtual shape more similar to the structure of the problem to be solved.

We will work on the Cartesian topology, that is, processes will be identified according to n-dim coordinates instead of a linear rank number. The number of dimensions and the size of each dimension are configurable. After the creation of the virtual topology, processes keep their linear rank ID within `MPI_COMM_WORLD` but they will be given coordinates in the virtual matrix as well.

Various functions implement this functionality. Here we present the main ones:

```
int MPI_Cart_create(MPI_Comm comm1, int ndims, int *dim_size, int
    *periods, int reorder, MPI_Comm *comm2)
```

This function creates a new communicator `comm2` to which all processes belonging to `comm1` `(MPI_COMM_WORLD)` are included. In `comm2` processes are identified by `ndims` coordinates. Each dimension's size is set through `*dim_size` poiting to an `ndims` size vector. Each vector component set the size of the corresponding dimension of the topology. For instance, if we intend to arrange 12 processes in 4x2 matrix, we have to initialize `ndims` = 2, declare a two components vector; the first one has two be initialized to 4 and the second one to 3.

Slightly more complex is the behavior of `*periods` and `reorder`. The first one, `(*periods)` point to a `ndims` components vector setting the periodicity of each dimension's numbering. Don't worry too much about this at this point.

Parameter (`reorder`) allows MPI to modify (**1**) the order of the processes in respect to their order in `MPI_COMM_WORLD`. Again it is not an issue for the moment.

In order to be able to assign workload to the processes according to their brand new coordinates, these must be known first. The following function returns the local coordinates:

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

Coordinates of process `rank` are returned in `coords` vector. `Comm` is the new communicator although `rank` is the process ID in `MPI_COMM_WORLD`.

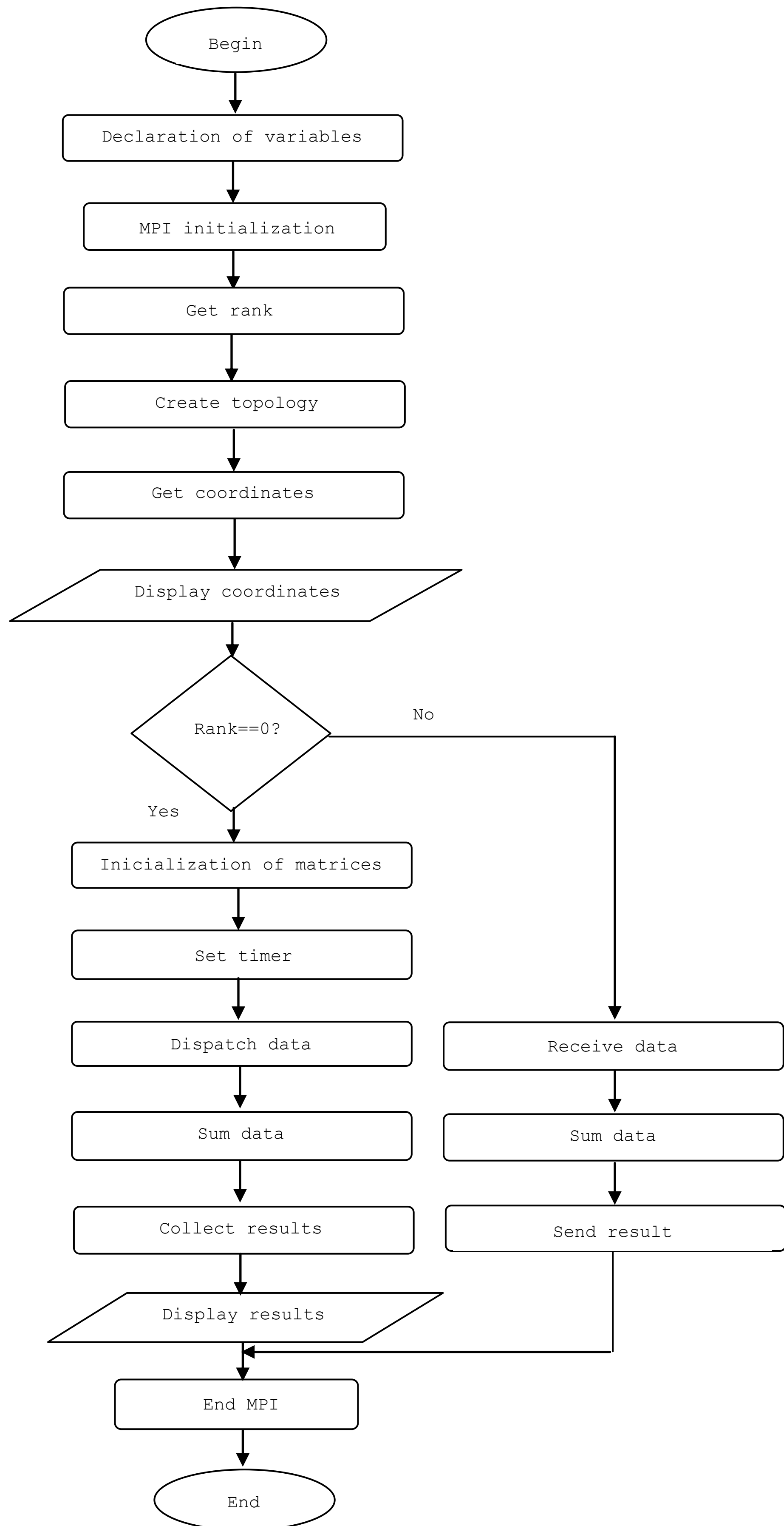
PRACTICAL EXERCISE

A matrix sum has to be programmed. Processes will be arranged according to the matrices' structure so each process will sum the elements whose position within the matrix corresponds to the process' coordinates. Once again, process 0 will display the result and the elapsed time.

QUESTIONS

- How are matrices stores in memory in the C language?
- Think of other situations where this topology can also be exploited.
- Think of other different topologies that could be of interest.

FLOWCHART



Activity 5: Parallel I/O

OBJECTIVES

- Get to know parallel input/output techniques.

THEORETICAL CONCEPTS

Serial I/O

Traditional application programs manage input/output. This usually implies reading input data at the beginning and writing results at the end. It is all conducted by the only process at work. Parallel application can easily keep this serial approach; so have we done so far. Process 0 has always been in charge of obtaining input data, deliver then to the rest and collecting and displaying results. Although feasible in many cases, this serial i/o can become a bottleneck in certain situations since process 0 has to manage all transactions. If all processes are granted access to input data and can send results to output channels, the bottleneck is cleared. This is what we know as parallel input/output.

Parallel I/O

Parallel input/output requires accesses to the corresponding channels from all processes. This is achieved by the use of shared files. All processes may have read and write access rights over these files. It is undoubtedly useful but some constraints have to be imposed to make it feasible.

First, the view each process has of the file has to be set. Obviously processes cannot have arbitrary access to the contents of the file, otherwise race conditions may occur. Each process will have a particular view or the file, meaning that its default read and writes will take place on a specific area of the file different from the rest. This area is not private; in fact, processes can access each other's area whenever they need to do so to Exchange information.

The second issue it the sort of privileges each process has over the file. MPI manages this in a way that every process establishes its own access rights.

MPI parallel I/O

MPI implements this functionality through a set of functions. The first one in the program must be the one to open the common file:

```
int MPI_File_open(MPI_Comm com, char *fichero, int mode, MPI_Info info,
    MPI_File *fh)
```

The file pointed by `*fichero` is opened to perform operation `mode` on it (see table below). The function returns the file handler `*fh`. Parameter `info` refers to a process information handler whose content depends on the MPI distribution. At this point we will give it a null value (`MPI_INFO_NULL`). The following table shows all possible operation modes on the file:

Access mode	Description
MPI_MODE_RDONLY	Read only
MPI_MODE_WRONLY	Write only
MPI_MODE_RDWR	Read & Write
MPI_MODE_CREATE	Create file in case it doesn't yet exists
MPI_MODE_EXCL	Return error when trying to créate an already existing file.
MPI_MODE_DELETE_ON_CLOSE	Delete file when closed.
MPI_MODE_UNIQUE_OPEN	Concurrent access to the file is not allowed
MPI_MODE_SECUENTIAL	Only sequential access to the file is permitted
MPI_MODE_APPEND	All pointers are initially set to the end of file

Table 3. MPI file access modes.

All these modes are not alternative but complementary. In fact, it is quite common to combine several ones. For instance, we can create it in case it's necessary and open it for read and write:

```
MPI_MODE_CREATE | MPI_MODE_RDWR
```

Next operation to be performed is the definition of file view for each process. Each one has to set where exactly will it set the beginning of the file, what its internal structure will be and what sort of data are to be stored. The MPI function to do it all is:

```
int MPI_File_set_view(MPI_File fh, MPI_Offset offset, MPI_Datatype
   dtipo, MPI_Datatype ftype, char *datarep, MPI_Info info)
```

The file to be configured is set by the handler obtained from the previous function. Current process will see the file as if it began at position `offset`. From there on `dtype` data are to be stored. Parameter `ftype` sets the file's structure, that is, the way data introduced by different processes are going to interleave. This is actually a powerful tool as it allows to interleave different data types and sized introduced from different processes. To do so it is a MPI derived data type has to be created. We will ignore this for the moment thus making `ftype` and `dtype` equal. Our files will present a homogeneous structure. Parameter `*datarep` sets how data are represented in the file when transferring them from memory. There are three possibilities: "`native`", "`internal`" or "`external32`". The "`native`" data are moved from memory to the file unchanged. Their format differs according to the specific MPI distribution, so different computers might see the file contents differently. Format "`external32`" is common to all MPI implementations. The downside of this is that a format conversion is required. Midway between these two is the "`internal`" format. It only performs data conversions when necessary. Since all the nodes in our system are equal, we'll use the "`native`" format in our experiments.

After initializing the file view it is already possible to read and write data. There are several functions available, let's see the most simple:

```
int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int
   count, MPI_Datatype dtype, MPI_Status *estatus)
```

File `fh` will be read from position `offset.` As usual, `count` is the number of data to be read and to be stored in `buf`. These pieces of information will be of `dtype` type.

```
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf, int
   count, MPI_Datatype dtipo, MPI_Status *estado)
```

File `fh` will be written from position `offset.` Again, `count` is the number of data to be read and to be stored in `buf`. These pieces of information will be of `dtype` type.

Once the read/write operation is performed, the file must be closed:

```
int MPI_File_close(MPI_File *fh)
```

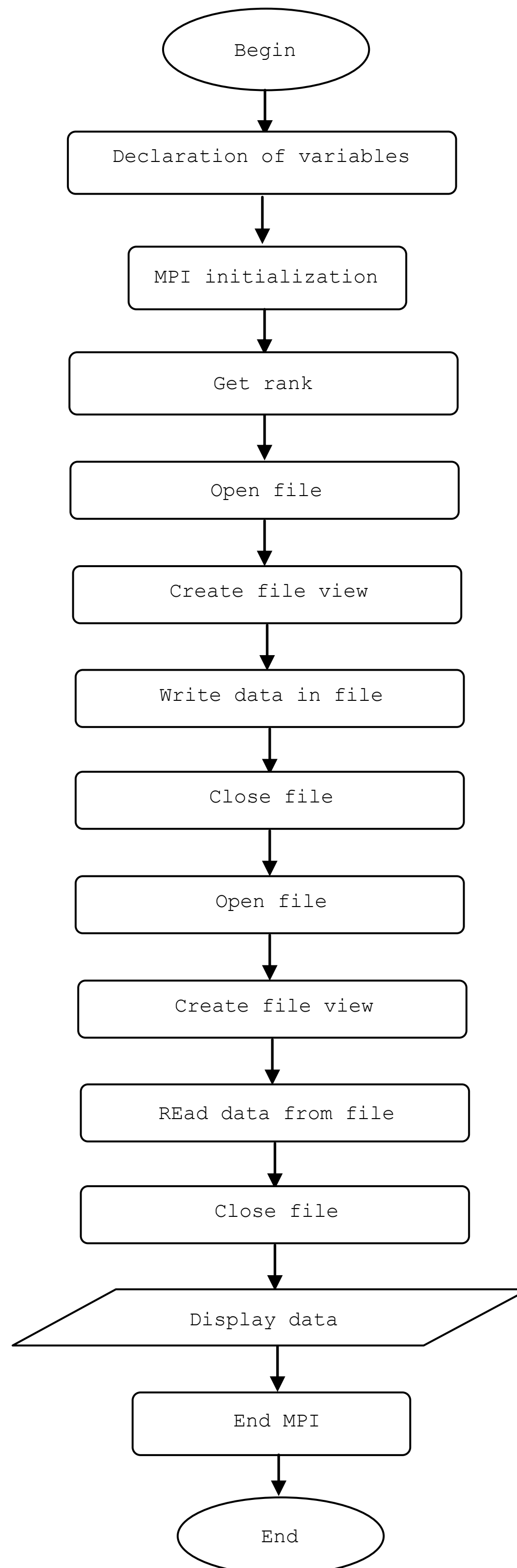
PRACTICAL EXERCISE

We will carry out a simple program to check the use of parallel input/output. All launched processes must store in a file their rank a certain number of times (configurable). The writings will be sorted according again to the rank. Afterwards, each process will read and display the data introduced by it. In order to make the file readable, 48 value can be added to the rank. This will transform the integer value of the rank into its ASCII equivalent. Alternatively, a char could be written.

QUESTIONS

- Is it possible to use parallel input output as an alternative way of scattering information to other processes?
- What are the setbacks of this procedure?
- Is there any advantage on its use?

FLOWCHART



Activity 6: New communication modes

OBJETIVES

- Increase our knowledge on MPI communication modes.

THEORETICAL CONCEPTS

Other communication modes in MPI

In exercise 1 various different way of communicating were introduced but only the standard mode was then used under the functions **`MPI_Send`** y **`MPI_Recive`**. We are now in a position to take further steps.

Previous communication functions implement the so called blocking mode, that is, they don't allow process progression until the communication event is completed. It is important to remember that this completion means nothing else than copying the message in a local buffer.

Nevertheless, this process blocking leads to a performance penalty. So as to sort out these negative implications, MPI provides an alternative, included within the standard communication mode as well. In the non-blocking mode the communication event is Split in two parts: in the first one the operation is initiated, whereas in the second one it is finished. Meanwhile instructions not depending on communication results can be processed. In general terms, all operations that don't depend on the data to be transferred can proceed in that interval.

Let's have a look at the functions that trigger the communication event:

```
int MPI_Isend(void *buf, int count, MPI_Datatype dtype, int dest, int
   tag, MPI_Comm comm, MPI_Request *request)

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

As we can see, the syntax doesn't change much. The main difference is the presence of parameter `*request`. It returns a pointer to the communication task itself so it can be addressed in the finalization operation.

Communication operations may end up in two forms: *wait* or *test*. If we choose *wait*, processing will stop until the communication is completed. Non-blocking send plus wait does not differ much from a blocking send. In case a *test* operation is launched the processes will be aware of the finalization status of the operation and behave consequently.

Wait function for both send and receive operations is:

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

Its input parameter `*request` is the pointer returned by the starting functions. Test function is:

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

Unlike the previous one, this one returns a flag indicating whether the operation has finished “**1**” or not “**0**”.

RPRACTICAL EXERCISE

Let’s imagine a scenario where the factorial of a certain number is to be calculated. Process 0 will obtain the numbers (integers) from the user. Calculations will be performed by process 1 (suppose it resides in a powerful machine where calculations are faster). In order to avoid collapsing calculation by an excessive number of requests from the user, input data will be requested only when previous operation has finished. Otherwise a wait message will be displayed. The message will be followed by an increasing number of dots as the time goes on. This scenario is quite unlikely to happen since factorial calculations are actually quite short for modern computers. To force waits to happen we can repeat calculation at process one as many times as necessary to make its work harder.

Process 0 will display the result of the factorial calculation. In order to provide the user a proper way to end up this program the number 0 will be treated as a “scape” condition.

QUESTIONS

- How can the power of non-blocking communications be exploited to avoid working with obsolete data?
- Is it possible to generate a deadlock (processes waiting for one another) when using non-blocking functions?
- Make a brief dissertation about the concept of deadlock and how it affects the different communication modes.

FLOWCHART

Activity 7: Derived data types

OBJECTIVES

- Improve data Exchange capabilities with the use of derived data types.

THEORETICAL CONCEPTS

MPI derived data types

So far, very simple data types have been exchanged: integer, float or vectors. It is enough in many cases but, in certain situations it’s worth using more potent capabilities. Let’s imagine we could exchange data structures defined by the user. Structures don’t exist as primary data types in MPI so its components would have to be delivered separately. This is a feasible solution but there is a better option though. The question is, Can data structures be defined in MPI? The answer is yes but… This possibility is available through the creation of the derived data types. This is a bit more complex than the use of data structures in C since not only different data types can be interleaved but also their relative location in memory has to be specified. This means that the user can create gaps between the data. This has some relevant uses. If we think about the way matrices are stored in memory, where they are saved in row order, if we need to deal with sub-matrices, we can introduce the mentioned gaps to skip the parts of a row that are not to be processed.

Let’s have a look at the tools provided by MPI:

```
int MPI_Type_struct(int count, int *vector_block_length, MPI_Aint
   *vector_offset, MPI_Datatype *vector_types, MPI_Datatype *new_type)
```

Parameter `count` describes the number of elements in the data type. Parameter `vector_block_length` contains each element’s length (the elements could be arrays). Parameter `vector_offset` specifies the position of the element in memory related to the beginning of the message. Parameter `vector_types` describes the MPI data type of each element. Finally `new_type` is a pointer to the just created data type. Note that it points to a data type not to a real data structure.

Once the new type has been defined, the following function has to be called to make it available:

```
int MPI_Type_commit(MPI_Datatype *new_type)
```

The following example intends to make the use of these functions more straightforward:

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


int main (int argc, char *argv[])
{
       int mirango;
       int vector_long[3];
```

```
	MPI_Aint vector_despl[3];
	MPI_Datatype vector_tipos[3];
	MPI_Datatype nuevo_tipo;

	typedef struct{
		float a;
		int b;
		char c;
	}viejo_tipo;

	viejo_tipo datos;

	vector_long[0] = vector_long[1] = vector_long[2] = 1;
	vector_despl[0] = 0;
	vector_despl[1] = sizeof(float);
	vector_despl[2] = vector_despl[1]+sizeof(int);
	vector_tipos[0] = MPI_FLOAT;
	vector_tipos[1] = MPI_INT;
	vector_tipos[2] = MPI_CHAR;

	MPI_Init (&argc, &argv);
	MPI_Comm_rank (MPI_COMM_WORLD, &mirango);
	MPI_Type_struct(3,vector_long,vector_despl,vector_tipos,
	               &nuevo_tipo);
	MPI_Type_commit(&nuevo_tipo);
	if(mirango==0)
	{
		datos.a = 1.5;
		datos.b = 10;
		datos.c = '2';
	}
	MPI_Bcast(&datos, 1, nuevo_tipo, 0, MPI_COMM_WORLD);

	if(mirango!=0)
	{
		printf("a=%f b=%d c=%c",datos.a,datos.b,datos.c);
	}
	MPI_Finalize();
	return 0;
}
```

The function described before is the most flexible option but, in case our data type is to be quite simple some more functions can be considered:

```
int MPI_Type_contiguous(int count, MPI_Datatype old_type, MPI_Datatype
   *new_type)

int MPI_Type_vector(int count, int block_length, int hop, MPI_Datatype
   element_type, MPI_Datatype *new_type)

int MPI_Type_indexed(int count,int *vector_length, int *vector_offset,
   MPI_Datatype element_type, MPI_Datatype *new_type)
```

The first one converts `count` contiguous elements from the old type, into one element of the new one.

The second one converts `count` blocks with `block_length` contiguous elements from the old type separated `hop` elements from one another, into the new type.

The third one takes `count` blocks, where the ith block is integrated by a number of elements given by the nth component of `vector_length` and are `element_type` typed. This block is shifted from the beginning of the new type a distance equal to the size of `element_type` multiplied by the ith component of `vector_offset`. It is a bit complicated but can be useful in certain cases.

In all cases function **`MPI_Type_commit`** has to be called afterwards.

PRACTICAL EXERCISE

Our program will have three processes. Process 0 will initialize an NxN matrix containing random numbers. Processes 1 and 2 will initialize their matrices to null. Then process 0 will send process 1 the upper triangular matrix whereas process 2 will be given the lower triangular in turn (figure 7.1). Both receiving processes will then display their matrices.

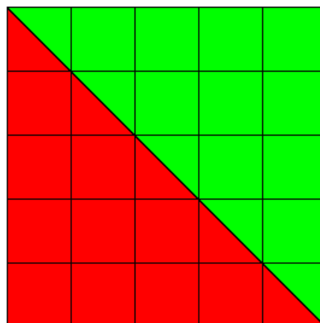


Figure 7.1. Square matrix with the main diagonal in blue.

QUESTIONS

- In this exercise, what sort of problems would we find if we used dynamic memory allocation to generate space for the matrices? How does this affect the creation of the new data types?

- There are some applications where, in order to perform a certain operation with a sub-matrix, the surrounding data are necessary (a row and a column for instance) although they will remain unchanged. How could this application be modified to make it possible?

- Think of other situations where derived data types may be of help.

FLOWCHART

Activity 8: Dynamic Process Management

OBJETIVES

- Try the dynamic cluster configuration tools as an approach to the virtual machine model.

THEORETICAL CONCEPTS

Dynamic cluster management

A complete dynamic management tool should make possible the creation and deletion of remote processes already in progress without restriction. Esto, que está muy bien logrado en PVM, en MPI está aún al principio del camino. Given the fact that MPI is a cluster oriented environment rather than to a virtual machine related done, it is not so concerned about the dynamic creation and removal of processes at run time. This not only restricts the capabilities of the message passing system but also may lead to full system failures in case one of its nodes vanishes.

The relentless expansion of MPI makes its developers try to overcome these constraints thus nearing MPI to the concept of virtual machine. A first step has been the introduction of some functions that permit the dynamic management of processes. In this form, an already running process is able to launch child processes to carry out some tasks for him.

Process management functions in MPI-2

The main and almost only function so far is:

```
int MPI_Comm_spawn(char *command, char *argv[], int maxprocs, MPI_Info
   info, int root, MPI_Comm comm., MPI_Comm *intercom, int
   array_of_errcodes[])
```

With this function, `maxprocs` copies of the MPI program pointed by `command`, are launched. The communicator pointed by `intercomm` is created to facilitate data exchange with them. A second `MPI_COMM_WORLD` is automatically created to provide communication between child processes. Initialization data can be passed to child processes through `argv`. Parameter `root` is parent's ID within communicator `comm`. Parameter `info` provides a series of values related to where and how to start child processes. Finally, an array of possible error codes is returned, one for each launched process.

The name of the child process can be directly written between quotation marks. Since `info` parameter is not to be used in this exercise it can be ignored using `MPI_INFO_NULL` and so can be done to ignore error codes: `MPI_ERRCODES_IGNORE`. If no command line arguments are not to be used `MPI_ARGV_NULL` is the value for `argv`.

Another useful function is:

```
int MPI_Comm_get_parent(MPI_Comm *comm)
```

It returns an inter-communicator to connect parent and child processes.

Another one:

```
int MPI_Attr_get(MPI_Comm comm, MPI_UNIVERSE_SIZE, int *universe_sizep,
    int *flag)
```

Parameter `universe_sizep` returns the expected number of processes. It is automatically set and depends on MPI distribution. LAM MPI initializes it to the number of machines in the cluster. Its value can be used as `maxprocs` in **`MPI_Comm_spawn`** function. Should this functionality not be available `flag` will be set to false.

In order to make communication between parents and child processes possible the inter-communicator must be transformed into intra-communicator at both sides:

```
int MPI_Intercomm_merge(MPI_Comm intercom, int order, MPI_Comm
    *intracom)
```

The intra-communicator is integrated by both parents and child processes. Parameter `order` sets the order of the processes within the new communicator. This parameter set to false in the parent processes means that it will have rank 0; likewise, child processes musk set it to true.

PRACTIAL EXERCISE

We will launch a single process which, at run time will launch a number of child processes. To do so, two ".exe" files have to be created, one for the parent and a different one for the child/children. Yet the parent is the one that is launched by MPI. Greeting messages will be exchanged and displayed between parent and child processes and between child processes as well.

QUESTIONS

- It is possible for a parent process to launch several children. Would it be possible for a child process to have several parents?
- Could a child process be launched by several parents alternatively?
- Can a child process become a parent and launch other ones?
- Try to think of more opportunities opened by these tools.

FLOWCHART

Activity 9: Example of real application

OBJETIVES

- Apply previously acquired knowledge to develop a bit more complex program intended to be used as a benchmark to measure system performance.

THEORETICAL CONCEPTS

No new concepts will be introduced in this chapter since it is meant to exploit those already learned. As obvious, not all aspects of MPI development environment have been exposed and nor our application program is expected to find the most optimal solution but quite a good job is possible though.

However, it may be helpful to introduce some additional information about the functions we already know. Function **MPI_Recv** returns a MPI_Status type parameter that we haven't used so far. It is a structure integrated by 3 elements: MPI_SOURCE, MPI_TAG & MPI_ERROR. The first one contains the Rank of the sender process. If the message was received under MPI_ANY_SOURCE it can be necessary to find out who sent it later on in the program. The second one returns the message's tag. If it was received under MPI_ANY_TAG, it could be interesting to get to know the tag's value as well. The third one returns an error code. We won't deal with error codes in this exercise.

PRACTICAL EXERCISE

We will program a parallel matrix multiply. It is the student's decision how to scatter calculations among all the processes. The size of the matrices (square) must be configurable. Dynamic memory allocation is strongly recommended so no limits to the size of the matrices are imposed.

Process 0 will initialize the operand matrices with any value (random, loop, etc). Data type will be float. In a first stage, multiplication results will be displayed to check correctness. Once the program has been validated, result printing must be removed to allow matrix size to grow. Execution time has to be displayed in all cases.

REMARK:

To combine double indexing with dynamic memory allocation for matrices, we must use double pointers. Each pointer within an array will give access to a row in a matrix:

```
// Declare a double poiter for the matrix
// This will let us refer to the elements in a [row][column] manner
float **Matrix;
// Initialize the double poiter to store poiters to each and every row in the matrix.
Matrix = (float **) malloc(ROWS*sizeof(float *));
// We initialize each poiter to the starting poit of each row
for (i=0; i< ROWS; i++)
{
        MatriX[i] = (float *) malloc(COLUMNS*sizeof(float));
}
// Now we can us [row][column] format for our matrix:
```

```
for (int i=0; i<ROWS; i++)
{
        for (int j=0; j<COLUMNS; j++)
        {
                Matrix[i][j] = 0.0;
        }
}
```

However, this dynamic allocation procedure does not guarantee that rows in the matrix are contiguous in memory. This can be necessary for sending functions in our program. We should send data row by row in that scenario. If we want to keep double indexing while adding contiguity, we will have to proceed as follows:

```
// Declare a double poiter for the matrix
// This will let us refer to the elements in a [row][column] manner
float **Matrix;
// Initialize the double poiter to store poiters to each and every row in the matrix.
MatriX = (float **) malloc(ROWS*sizeof(float *));
// Declare a new pointer to allocate memory space for the whole matrix.
float *Mf;
// Initialize the pointer that will guarantee consecutive location of all rows
Mf = (float *) malloc(ROWS*COLUMNS*sizeof(float));
// We initialize each poiter to the starting poit of each row.
for (i=0; i< ROWS; i++)
{
        MatriX[i] = Mf + i* COLUMNS;
}
// Now we can us [row][column] format for our matrix:
for (int i=0; i<ROWS; i++)
{
        for (int j=0; j<COLUMNS; j++)
        {
                Matrix[i][j] = 0.0;
        }
}
```

It is now important to notice that this alternative leads to the use of `Matrix[0]` as the starting address of the data stored in the matrix.

QUESTIONS

- In order to multiply A×B matrix A can be delivered to all processes whilst matrix B se can be distributed in columns. Think of a different option.
- Would it be possible to avail of the power of Cartesian topology to facilitate the resolution of this exercise?
- The need to broadcast one of the matrices slows program execution. Think of a different solution to avoid delivering so much information. Try to guess what the performance of this new option would be compared with the current program.

Activity 10: Performance Assessment

OBJECTIVES

- To measure system's performance in various circumstances.
- To learn how to estimate system's power and how to exploit it. A compromise between learning effort and code optimization must be obtained.

THEORETICAL CONCEPTS

In this chapter some common performance related concepts are presented:

- **Degree of parallelism (DOP)***:* Number of processors used to run a program in a precise moment on time. The curve, **DOP** = *P(t)*, representing the degree of parallelism as a function of time is called **parallelism profile** of the program. It doesn't need to match the number of processors available (*n*). For the following definitions we will assume that there are more processors than necessary to reach the maximum degree of parallelism admitted by a program: **máx**{*P(t)*} = *m* < *n*.

- **Total amount of work**: Being Δ the computation capacity of a single processor, given either in MIPS or MFLOPS, and assuming all processors to be equal, it is possible to measure the amount of work carried out between time instant t_A and t_B from the area under the parallelism profile as:

$$W = \Delta \cdot \int_{t_A}^{t_B} P(t) \cdot dt \,.$$

  Usually the parallelism profile is a discrete graph (figure 3), so the total amount of work can be computed as:

$$W = \Delta \cdot \sum_{i=1}^{m} i \cdot t_i \,.$$

  Where t_i is the time span when the degree of parallelism is *i*, being *m* the maximum degree of parallelism all over the program's execution time.

  According to this, the sum of the different time intervals is equal to the program's execution time:

$$\sum_{i=1}^{m} t_i = t_B - t_A \,.$$

- **Average parallelism**: Is the arithmetic mean of the degree of parallelism along time:

$$\overline{P} = \frac{1}{t_B - t_A}\int_{t_A}^{t_B} P(t)\cdot dt = \frac{\sum_{i=1}^{m} i\cdot t_i}{\sum_{i=1}^{m} t_i}.$$

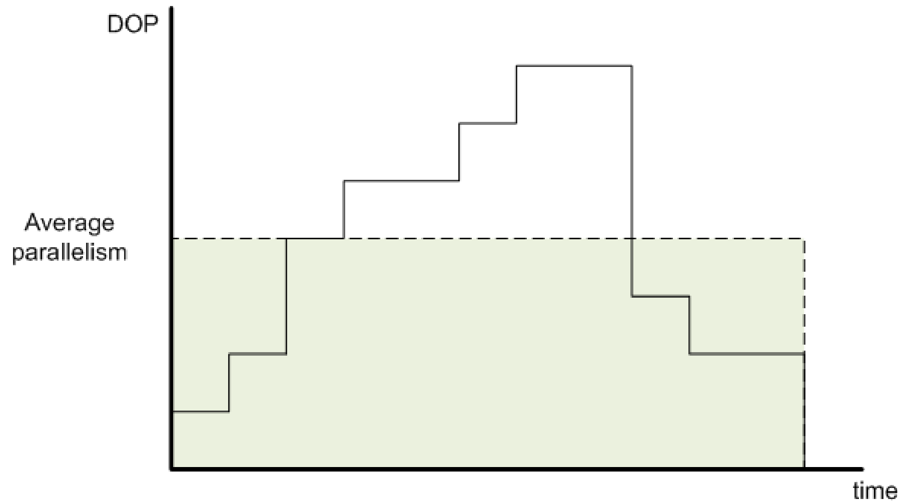


Figure 3.Parallelism profile and average parallelism.

- **Available parallelism**: Maximum degree of parallelism that can be extracted from a program, regardless of hardware constraints.

- **Asymptotic speedup**: Let $W_i = i\cdot\Delta\cdot t_i$ be the work done when **DOP** = *i*, hence $W = \sum_{i=1}^{m} W_i$.

  In this situation, the time employed by a single processor to carry out the work W_i is $t_i(1) = \frac{W_i}{\Delta}$; for *k* processors it is $t_i(k) = \frac{W_i}{k\cdot\Delta}$, and for an infinite number of processors it is $t_i(\infty) = \frac{W_i}{i\cdot\Delta}$.

  Hence, the **response time** is defined as:

$$T(1) = \sum_{i=1}^{m} t_i(1) = \sum_{i=1}^{m}\frac{W_i}{\Delta}$$

$$T(\infty) = \sum_{i=1}^{m} t_i(\infty) = \sum_{i=1}^{m}\frac{W_i}{i\cdot\Delta}.$$

  The **maximum speed-up** on a parallel system is reached when the number of processors is unlimited. It will be determined by the quotient of both:

$$S_\infty = \frac{T(1)}{T(\infty)} = \frac{\sum_{i=1}^{m} \frac{W_i}{\Delta}}{\sum_{i=1}^{m} \frac{W_i}{i \cdot \Delta}} = \frac{\sum_{i=1}^{m} \frac{i \cdot \Delta \cdot t_i}{\Delta}}{\sum_{i=1}^{m} \frac{i \cdot \Delta \cdot t_i}{i \cdot \Delta}} = \frac{\sum_{i=1}^{m} i \cdot t_i}{\sum_{i=1}^{m} t_i} = \overline{P}$$

It can be stated that the maximum speed-up for a parallel system with an unlimited number of processors is equal to the intrinsic average parallelism of the program to be parallelized. Obviously what is difficult is to figure out this intrinsic parallelism and make the program be as parallel as that.

A different way to calculate speed-up assumes that a job (being it either a single program or a group of them), is to be run in "*i*" mode if "*i*" processors are to be employed. In this scenario, R_i represents the collective speed of them all in either MIPS or MFLOPS; R_1 would be the speed of a single processor and $T_1 = 1/R_1$ the execution time. Let's suppose the job is conducted in "*n*" different modes, with different workload for each one, which results in a different weight f_i assigned to each mode. In this scenario, speed-up is defined as:

$$S = \frac{T_1}{T^*} = \frac{1/R_1}{\sum_{i=1}^{n} f_i/R_i}$$

Where T^* is the weighted harmonic mean of the execution time for the "*n*" execution modes.

In an ideal scenario, no delays are introduced by communications or lack of resources, so R_1 = 1, $R_i = i$:

$$S = \frac{1}{\sum_{i=1}^{n} f_i/i}$$

This expression is equivalent to the previous one.

From the previous case, the Amdahl's law is derived. $R_i = i$ and it is assumed that $W_1 = \alpha$ and $W_n = 1 - \alpha$, which implies that part of the work is to be done in sequential mode and the rest will exploit all system power. In this scenario:

$$S_n = \frac{1}{\frac{\alpha}{1} + \frac{1-\alpha}{n}} = \frac{n}{1 + (n-1)\alpha}$$

Hence:

$$n \to \infty \Rightarrow S \to 1/\alpha$$

In other words, system performance is upper bounded by the sequential part of the job.

- **System efficiency**: Determines the degree of exploitation of the resources available:

$$E = \frac{S}{n} = \frac{T(1)}{n \cdot T(n)} \leq 1$$

- **Redundancy**: Is the ratio between the number of operations performed by the system and those performed by a single processor to carry out the same job:

$$R = \frac{O(n)}{O(1)}$$

- **System utilization**:

$$U = R \cdot E = \frac{O(n)}{n \cdot T(n)}$$

- **Quality of parallelism**:

$$Q = \frac{S \cdot E}{R} = \frac{T^3(1)}{n \cdot T^2(n) \cdot O(n)}$$ assuming T(1) = O(1).

PRACTICAL EXERCISE

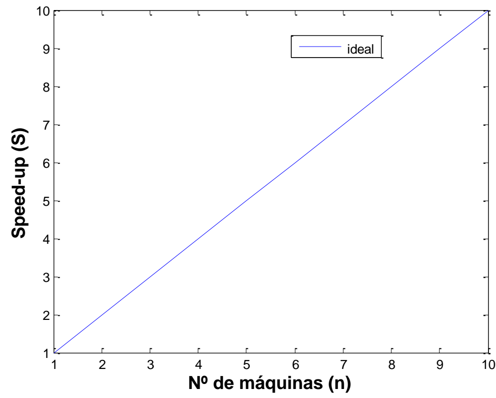
The program developed in the previous exercise (matrix multiply) is to be used as a benchmark to measure system performance. Matrix multiply is a cubic order problem that involves a significant calculation increase for a small increase in matrix size. In this exercise we will explore the influence of both system size and computation on execution time.

Concerning the amount of calculation, we must choose some precise values for matrix size. The first figure is intended to result in a similar execution time regardless of the amount of resources available. It will depend on the capabilities of the computers available. In our case we will start from matrices 3000×3000 in size. This leads to 27×10^9 multiplication operations.

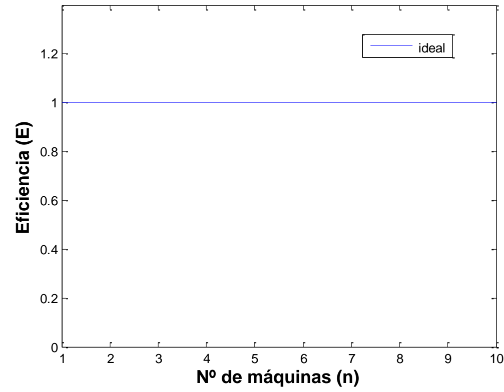
Starting from this size, we will increase matrix size to 4000 and 5000. For each of this values we will launch from 1 (2 in case process 0 doesn't perform calculations) to 6 (7) processes to be executed on the same number of computers. A graph representing execution time as a function of the number of computers (processes) should demonstrate that, when the workload is high, execution time is reduced proportionally to the number of resources deployed.

Speed-up graphs

Make a graph of the evolution of speed-up (*S*) and efficiency (*E*) as a function of the number of Computers and compare it with the ideal scenarios (figure a and figure b, respectively).



(a)



(b)

QUESTIONS

- For our experiments, determine: efficiency, utilization, redundancy and system quality.
- Compare the speed-up obtained with the one that should be achieved according to the amount of resources utilized.
- Try to figure out the reasons for the deviation.
- Describe which aspects should be improved to obtain a higher speed-up.

Appendix A: Installing DeinoMPI

DeinoMPI in an implementation of the standard MPI-2 for Microsoft Windows derived from Argonne Nacional Laboratory's MPICH2.

System requirements:

- Windows 2000/XP/Server 2003/Windows 7
- .NET Framework 2.0

Installation

DeinoMPI has to be downloaded and then installed in all computers in the cluster. The installation process is the same in all nodes. It requires administrator privileges for installation but all users can execute it afterwards.

Once it is installed folder \bin has to be added to the path.

Note: make sure Deino's version matches the operating systems requirements (32 or 64 bits).

Configuration

Once the software has been installed, each user will need to create a "Credential Store". It is used to launch routines in a secure manner. Mpiexec will not execute any of them without this "Credential Store". The graphic environment will show the user this option in the first execution.

Launching Jobs

Once again, both the graphic environment and the command line are valid.

Graphic Environment

This tool can be used to launch MPI processes, manage the "Credential Store", search for computers within the local network that have MPI installed, verify mpiexec entries to diagnose common problems, and go to the DeinoMPI web site to look for help and documentation.

Mpiexec tab

It is the main page and is used to launch and manage MPI processes.

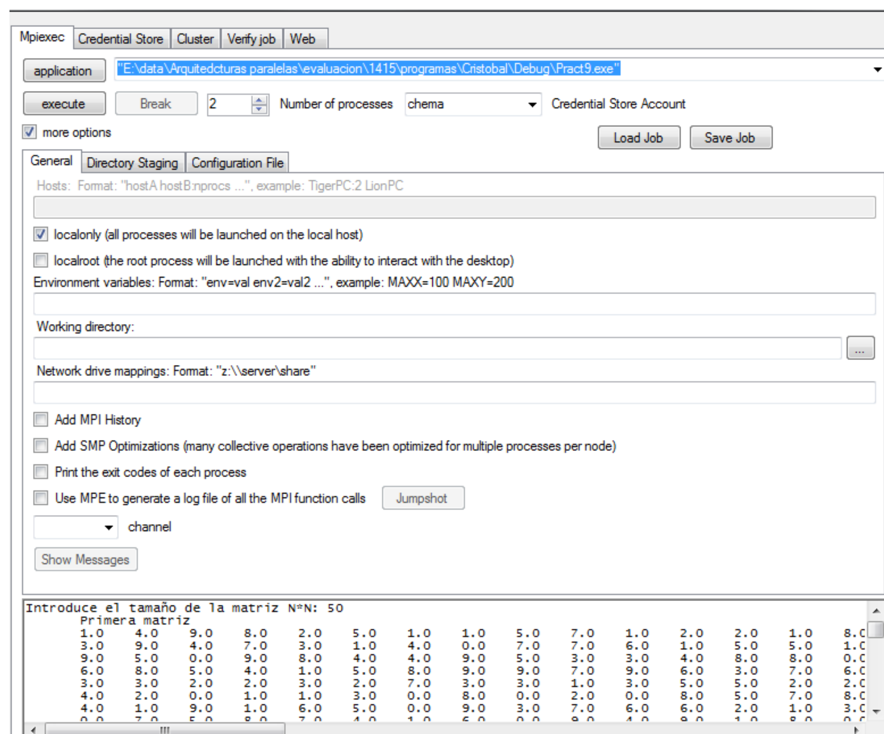


Figure A1. Mpiexec tab.

These are the main elements of this tab:

- **Application:**
  - The MPI application's path is introduced here. The same path will be taken by default in all nodes within the cluster so it is recommendable to copy the .exe file in the same folder in all of them.
  - If a network folder is specified, it is necessary to have sufficient privileges in the server.
  - The "application" button can be used to locate the .exe file.
- **Execute:** the program selected in the application dialog is launched when this button is pressed..
- **Break:** aborts program execution.
- **Number of processes:** Sets the number of processes to be launched.
- **Credential Store Account:** Sets the active user of the Credential Store.
- **Check box "more options":** It expands/contracts the options area.
- **Hosts:** Introduce here the list of hosts where you want the processes to run. Host names are separated by blanks. To execute the program in the local machine only, keep the default option "localonly" active or write down its name on this list

Credential Store Tab.

This tab is used to manage user's credential store. If no credential store has been created so far, select "enable create store options" check box to make remaining options available. They are hidden by default since they are only used the first time Deino is initiated.

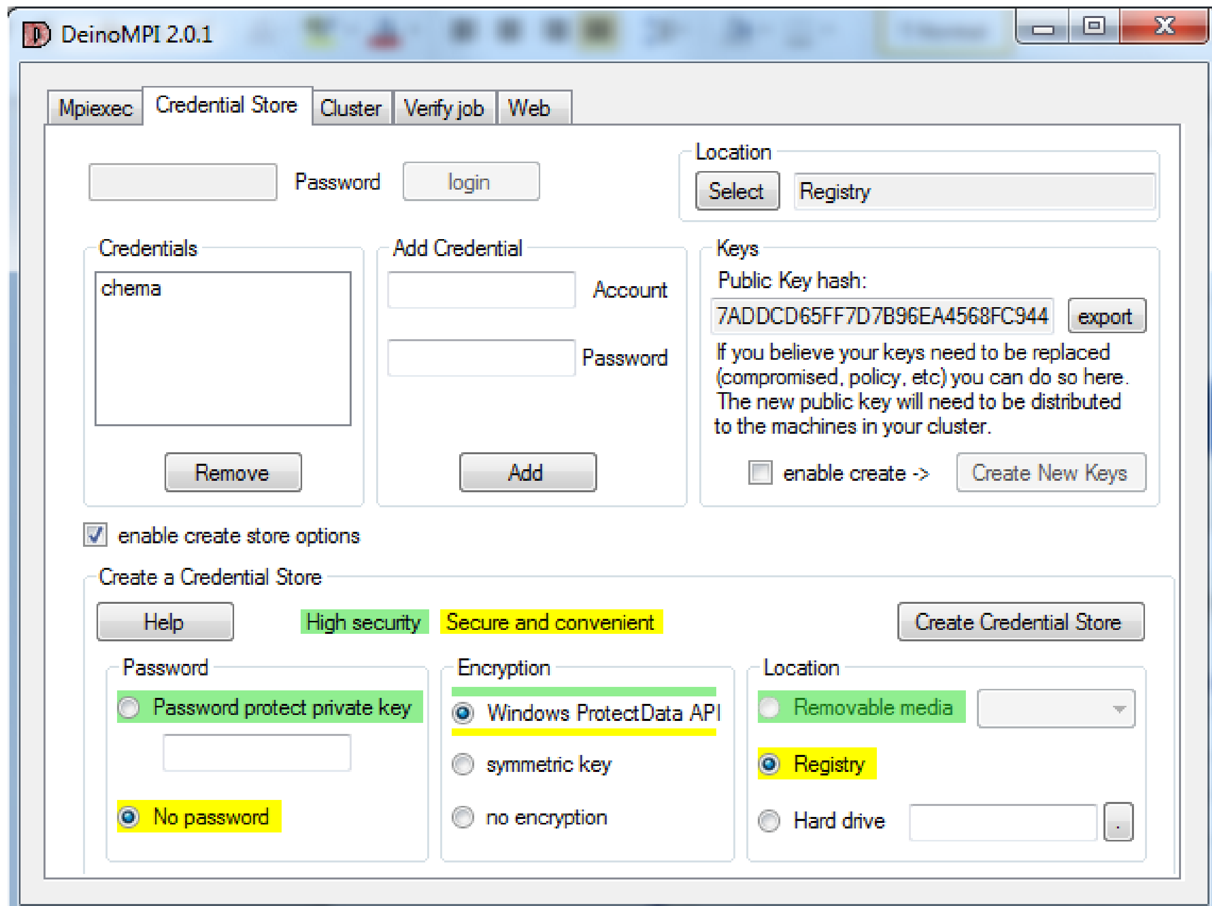


Figure A2. Credential Store tab including all options.

In order to create a credential store, the "enable create store options" check box must be selected. Three possibilities arise:

- **"Password":**
  - If this option is selected, the credential store will be protected from access by a password. It is the most secure option but forces the user to introduce the password any time a job has to be launched.
  - If "No password" is selected, the use of MPI is easier but more vulnerable. Without a password any program launched by the user can access the credential store which is not really a problem provided no malicious software is being used.
  - Even with this "No password" option active, the credential store is not available to other users if the encryption option is selected.
- **"Encryption":**

- "Windows ProtectData API" allows encryption of the credential store using the encryption scheme used by Windows for the current user. This ensures the credential store will only be available when the user is validated.
- If a password is selected the "symmetric key" encryption format can be chosen. This encryption is not specific to the user so other user knowing the password could access the store.
- The "no encryption" option is not recommended since it stores the credential store in a plain text file accessible to all users.

- **"Location":**
  - Take the "Removable media" option to save the store in an external device such as a memory stick. In this case, jobs can only be launched when the device is attached to the computer. This can be the safest option since the user can decide when the credential store is present. Combined with the use of a password and its encryption it can be protected even against loss or robbery.
  - The "Registry" option moves the "Credential Store" to the Windows registry.
  - Finally, it can be stored in the "Hard drive" which turns out to be the most common decision.

Cluster tab

In this tab, the computers in the cluster are displayed and the DeinoMPI version installed in each of them.

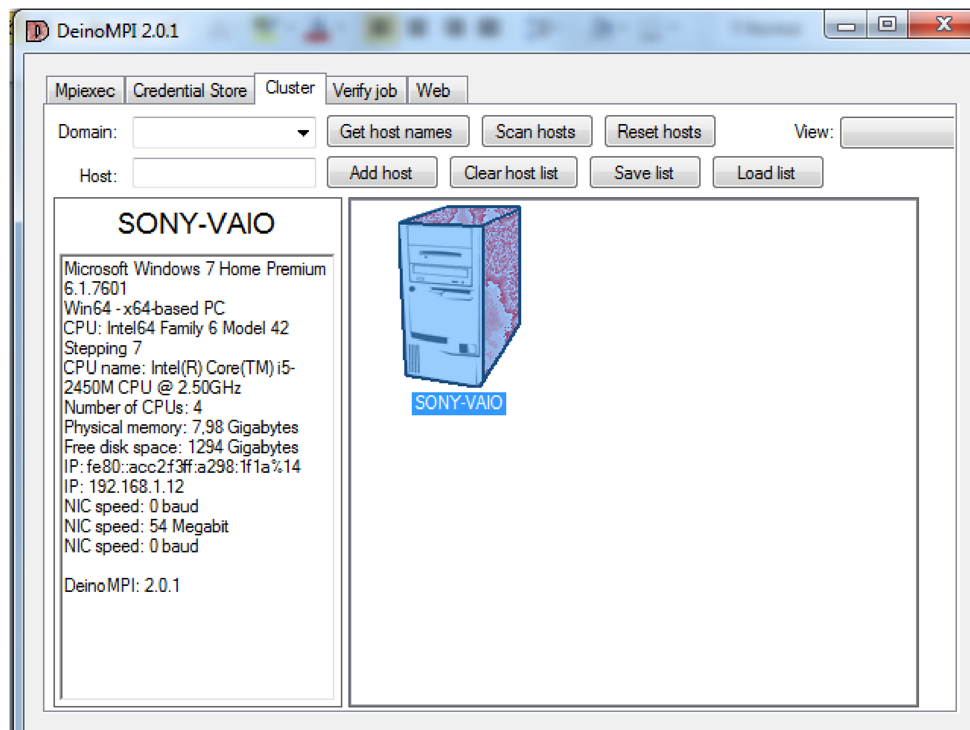


Figure A3. Cluster tab – Big icons view.

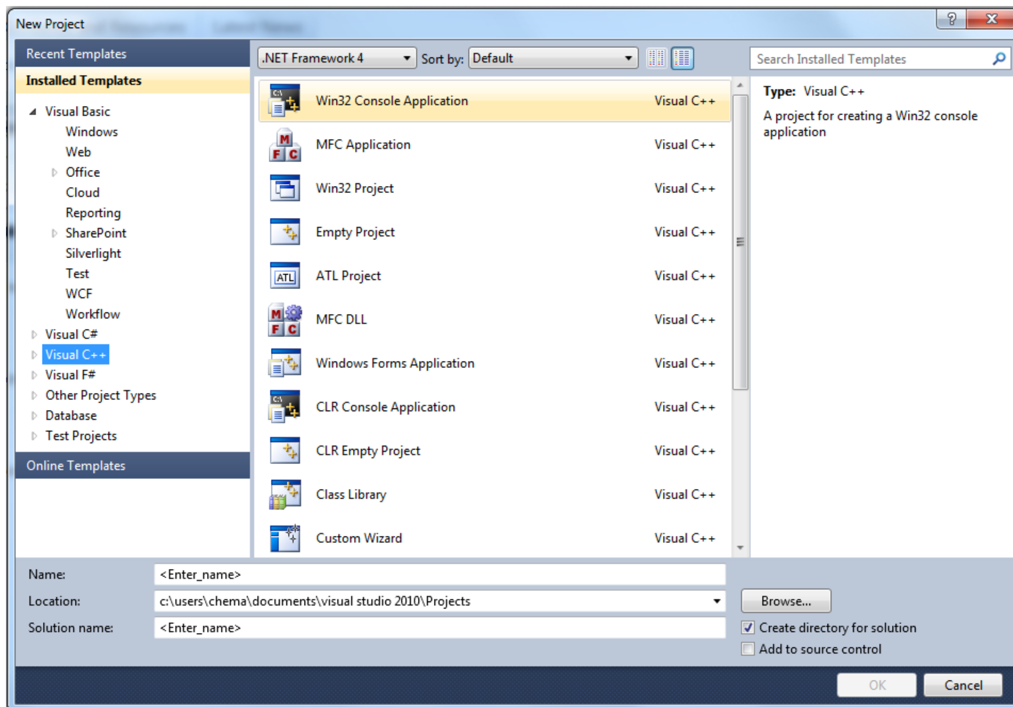
More hosts can be added writing down their name of can be found automatically within the selected domain.

Deino MPI manual. Available at: http://mpi.deino.net/manual.htm

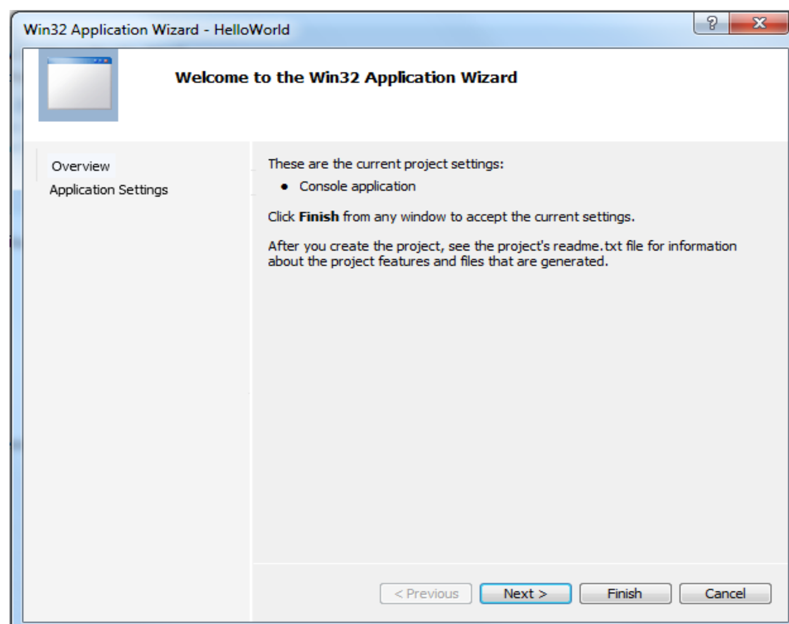
Appendix B: Project Configuration in Visual Studio 2010

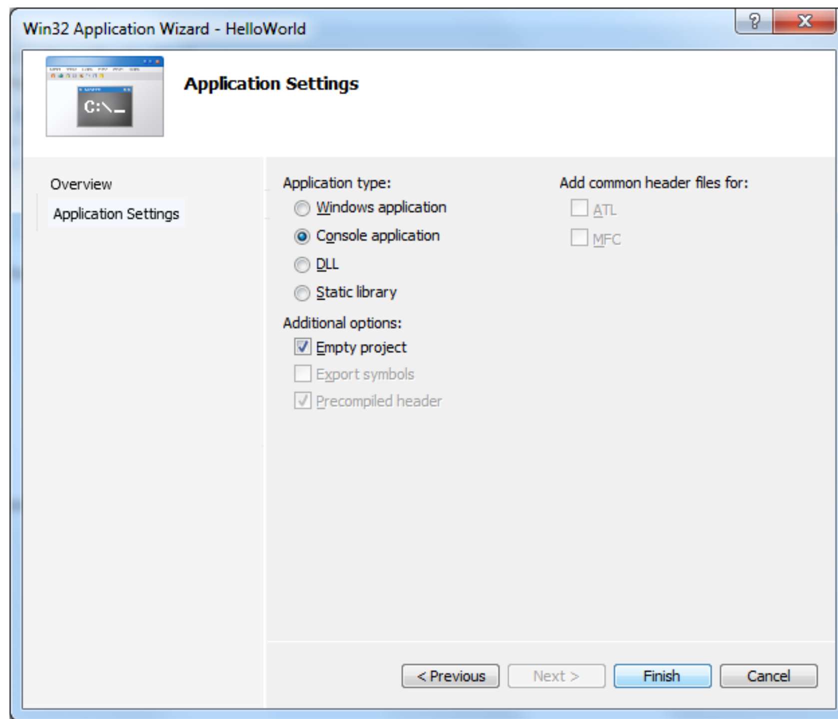
In this section we will describe the same configuration process but for the 2010 version of Microsoft Visual Studio. Configuration in more recent versions of Visual Studio is analogous.

- Generate a **new project and solution**. They may have both the same name:

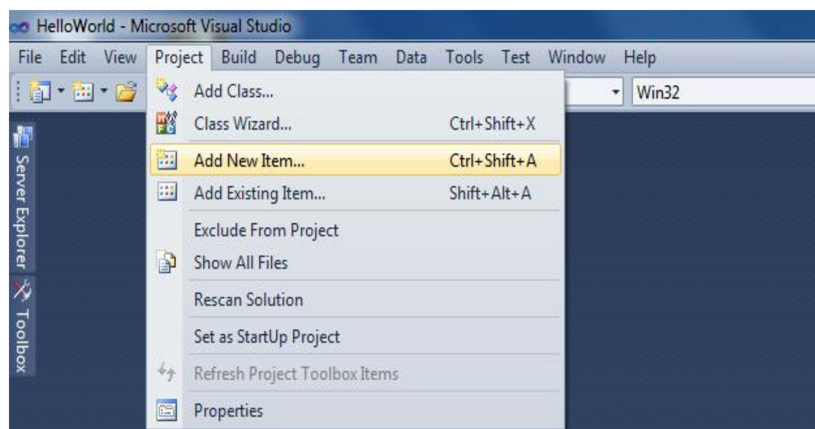


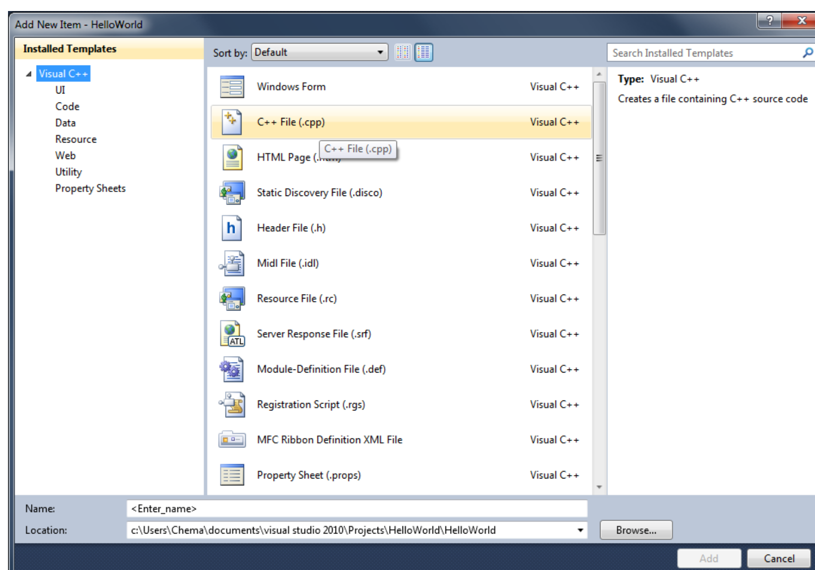
- Set it as **empty project**:



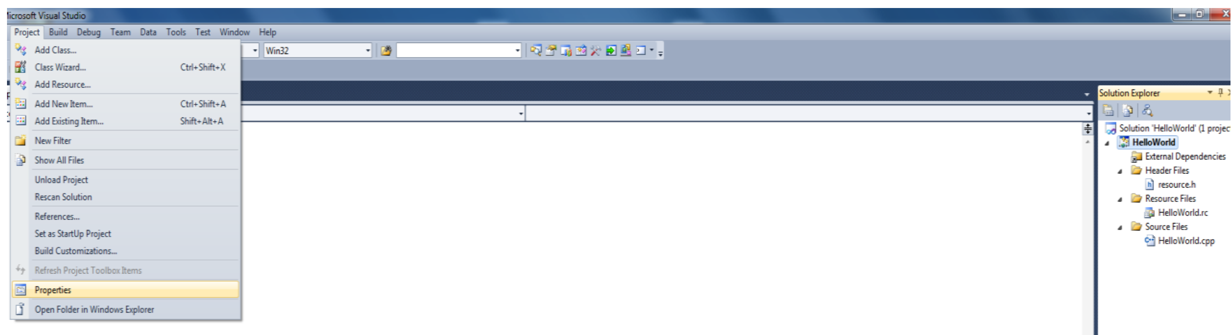


- Once created both the Project and solution, add a code file as **new item**:

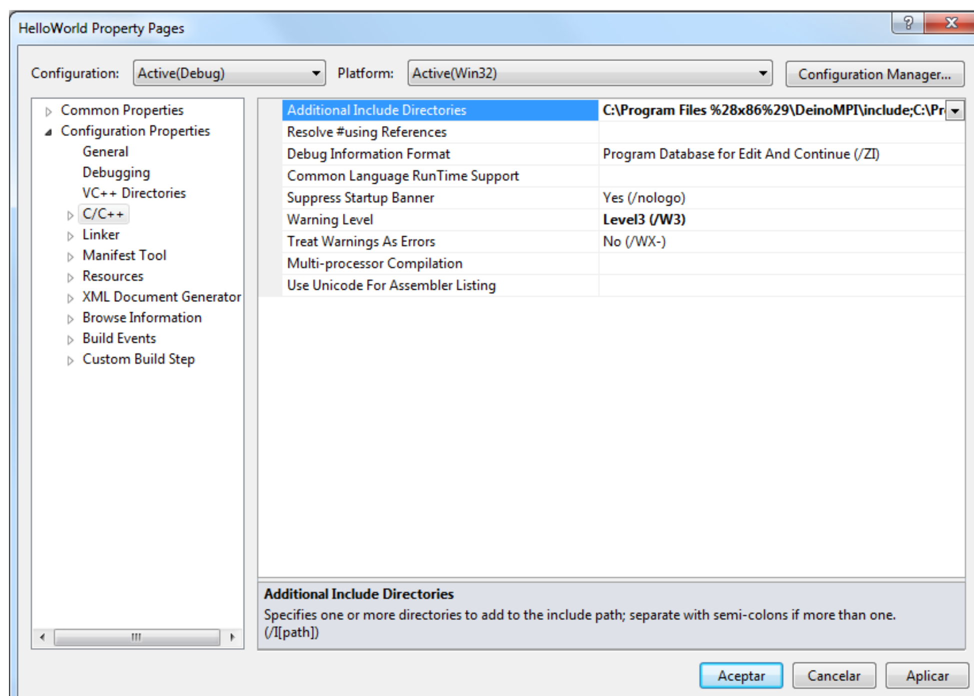




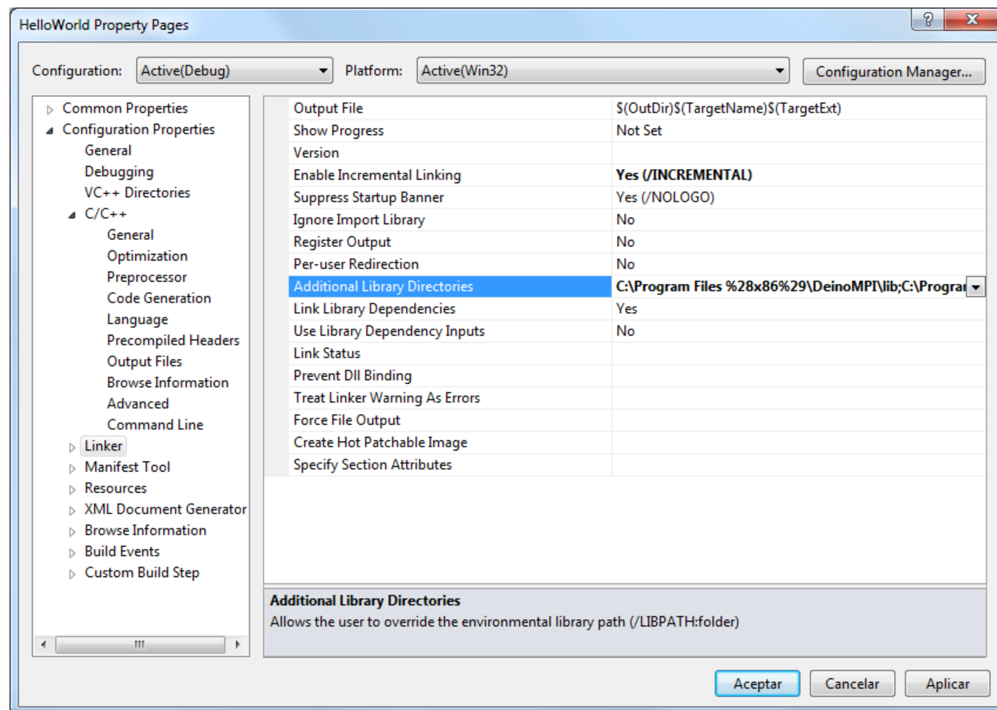
- Now, and never before, the Project settings are entered ("*Properties*"):



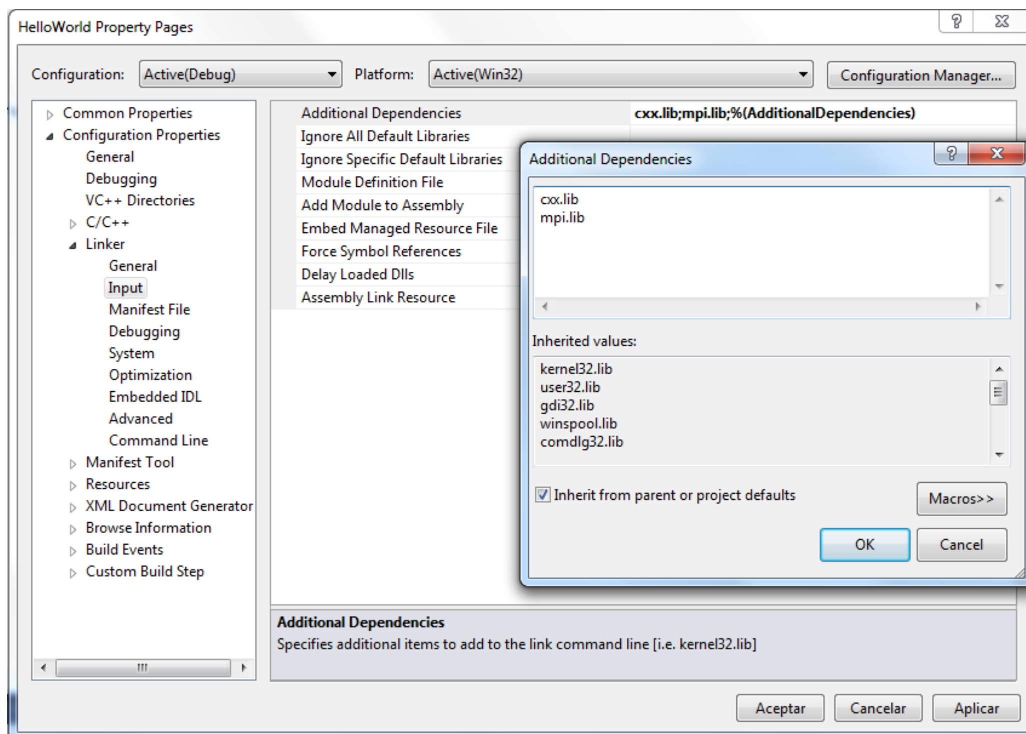
1. In the **C/C++** section we must enter the route to the folder where the header MPI files are located ("*Additional Include Directories*"). By default the *\Archivos de Programa (x86)\DeinoMPI\include* is assumed:



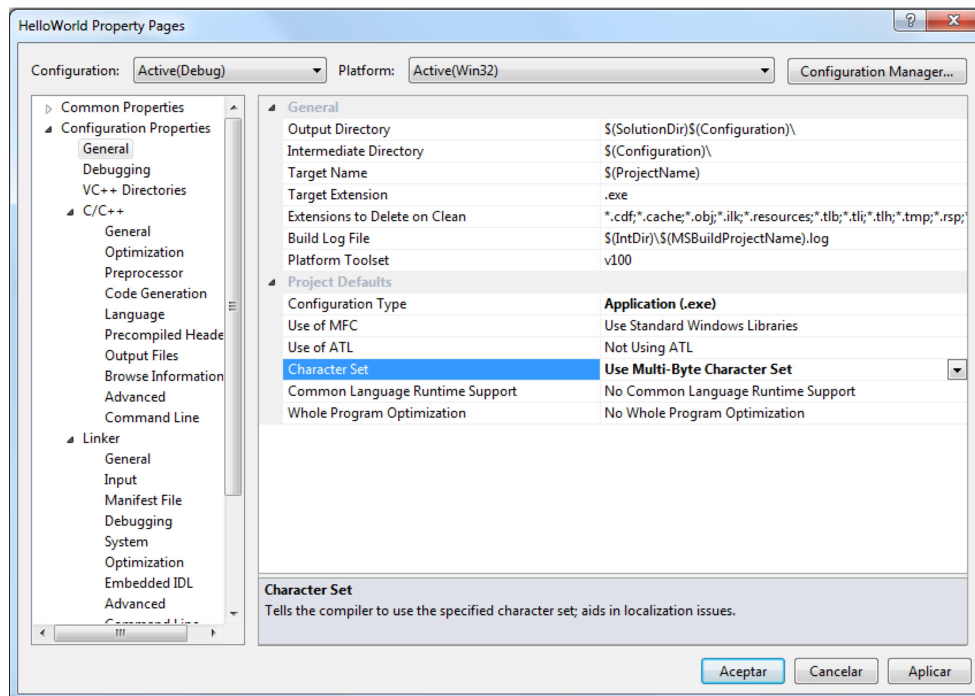
2. In the **Linker** section we must enter the route to the folder where the MPI libraries are located ("*Additional Library Directories*"). By default *\Archivos de Programa (x86)\DeinoMPI\lib* is assumed:



3. In the **Linker** section, in the input entry ("*Input*") the "**cxx.lib**" y "**mpi.lib**" files must be added as additional dependencies:



4. In the **General** section the **Multi-Byte** set of characters ("*Characrer Set*") must be selected:

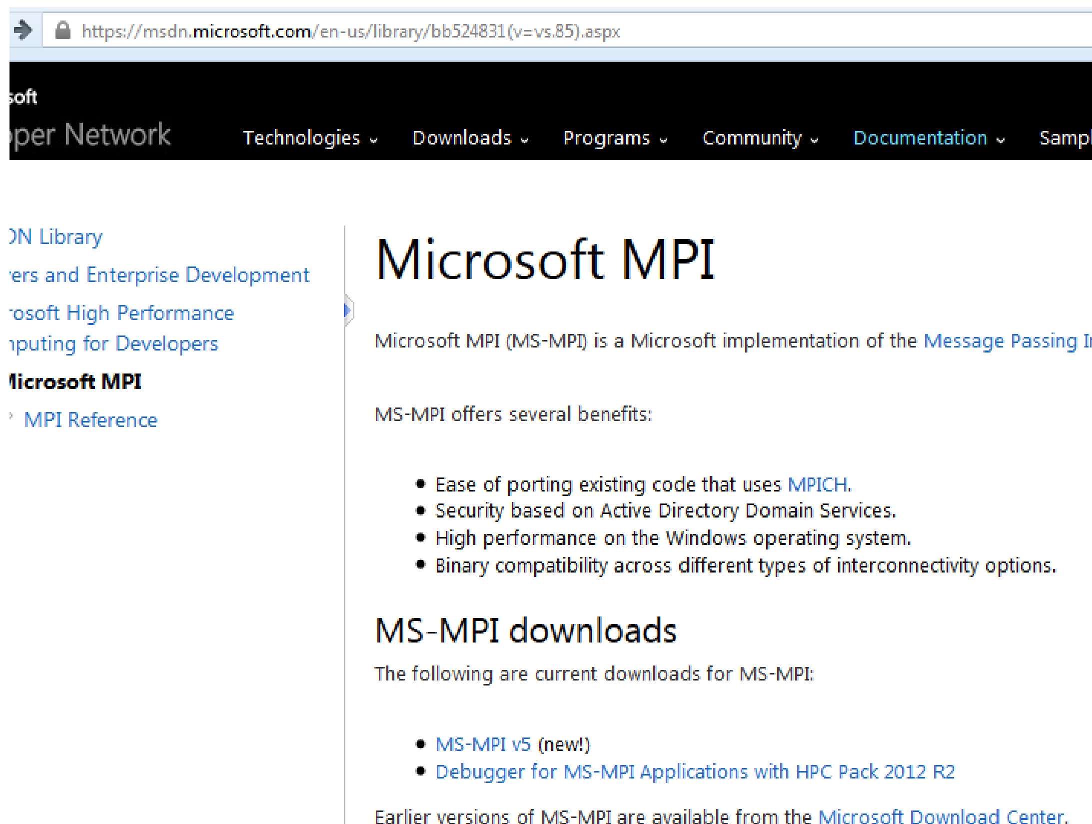


- Finally enter code in the selected source file and build the project.

Appendix C: Configuration of MS-MPI.

DeinoMPI is hard to configure in some systems and it may not eventually work. As an alternative we can install and configure the Microsoft distribution of MPI. It doesn't provide a graphical interface but from the command line everything can be done. Take the following steps to get it to work:

- Download MS-MPI v5 from its web location:

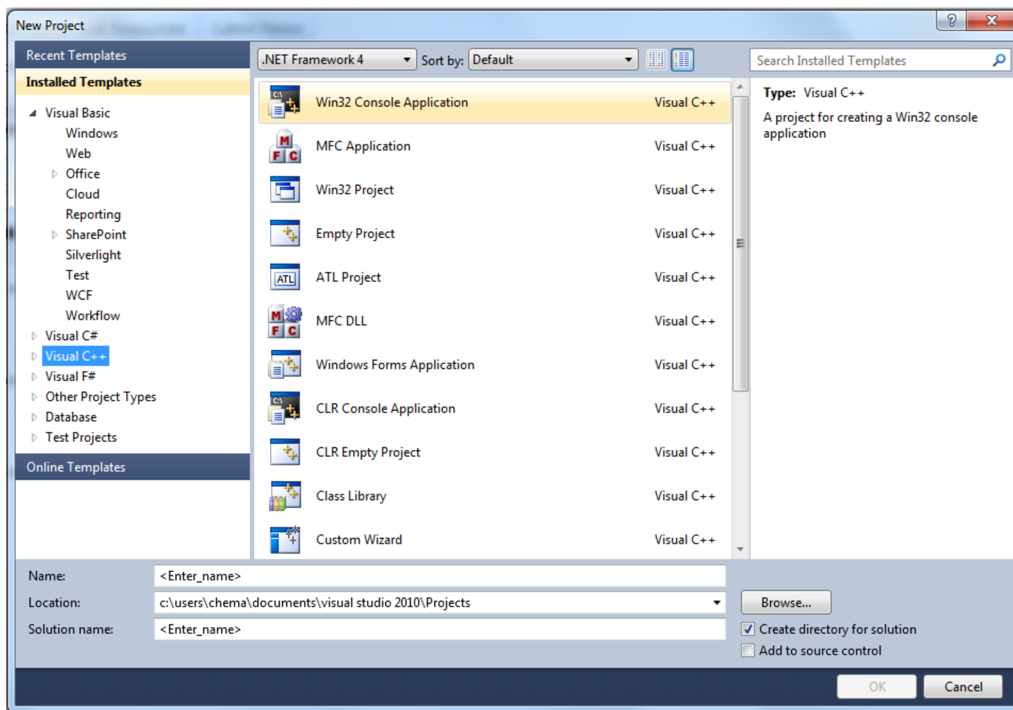


- There are two files and both have to be downloaded and installed:

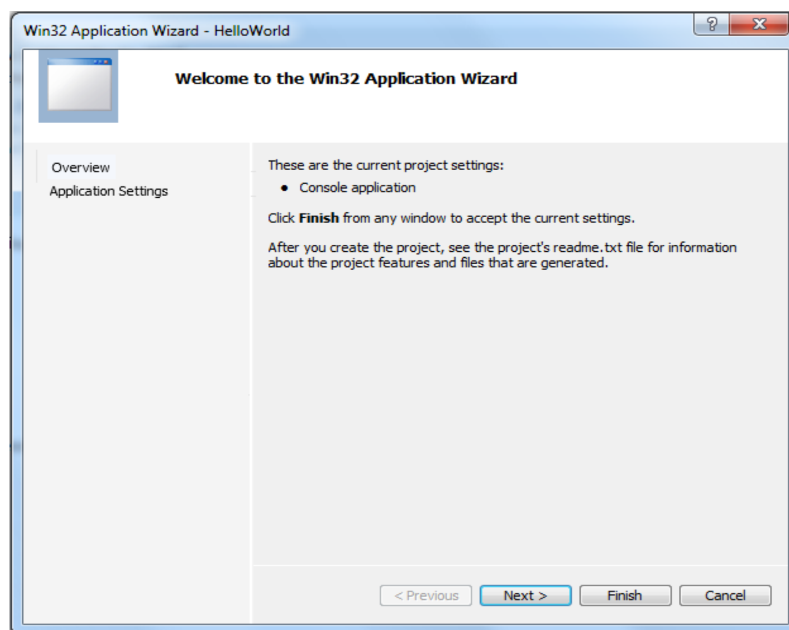
Choose the download you want

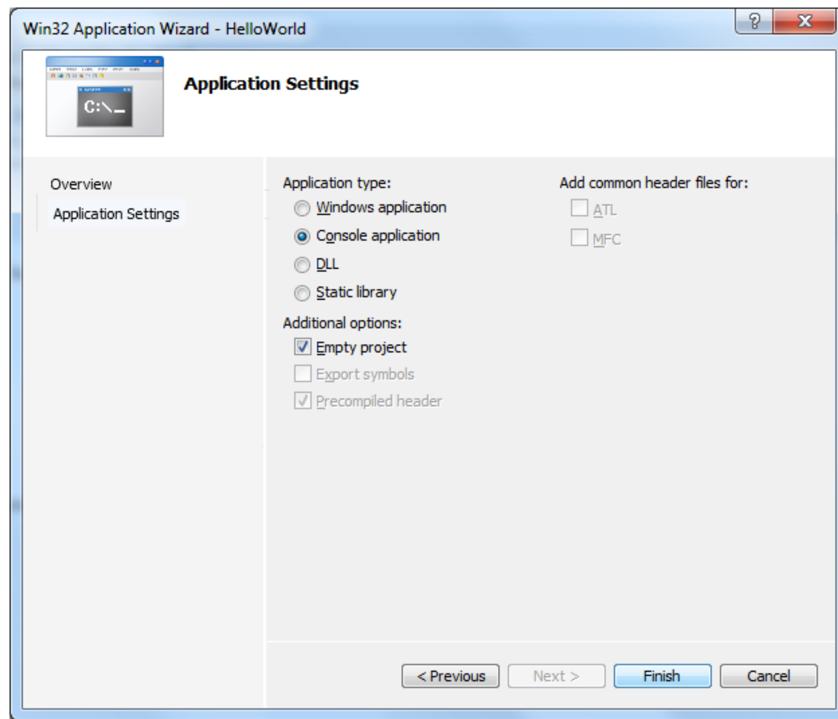
File Name	Size
msmpisdk.msi	1.9 MB
MSMpiSetup.exe	4.9 MB

- Each package creates a new folder: Program Files > Microsoft MPI and Program Files > Microsoft SDKs > MPI.
- Generate a **new MS Visual Studio project and solution**. They may have both the same name:

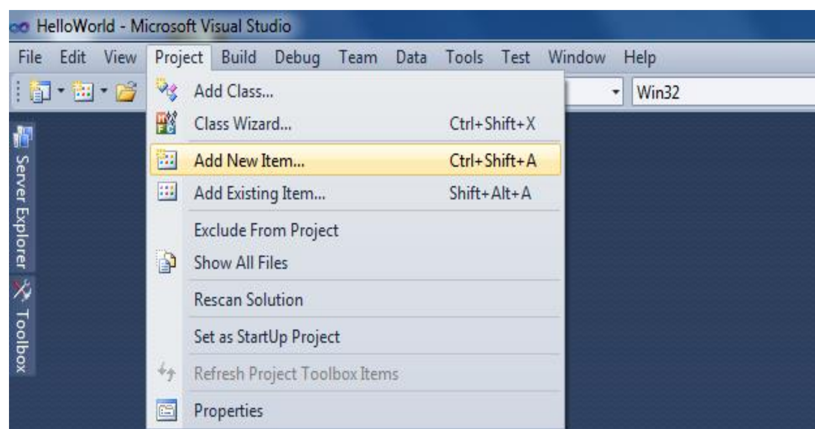


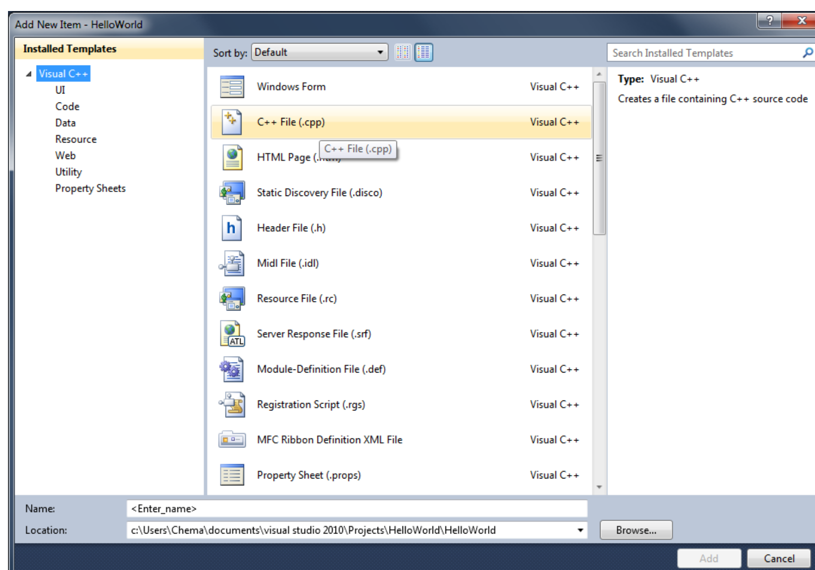
- Set it as **empty project**:



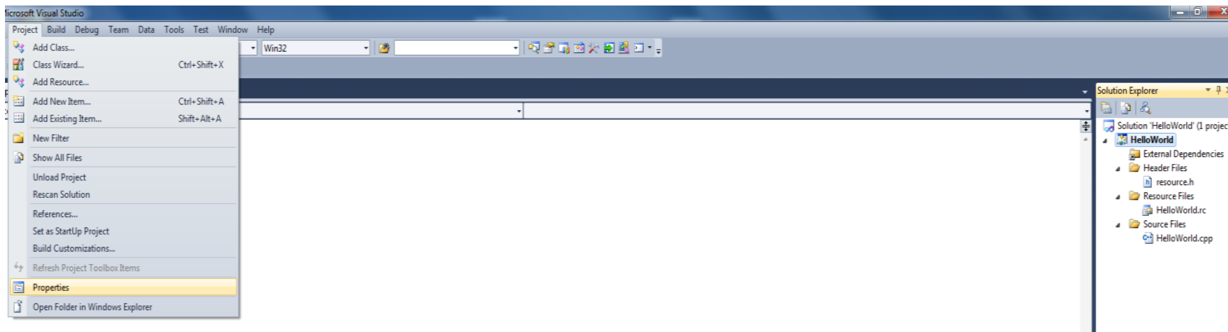


- Once created both the Project and solution, add a code file as **new item**:



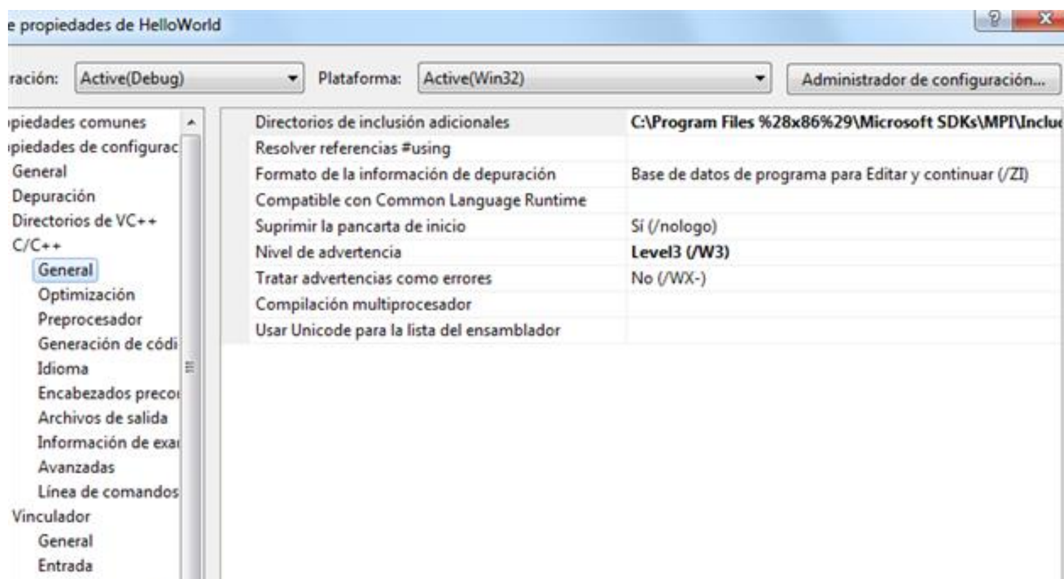


- Now, and never before, the Project settings are entered (“*Properties*”):

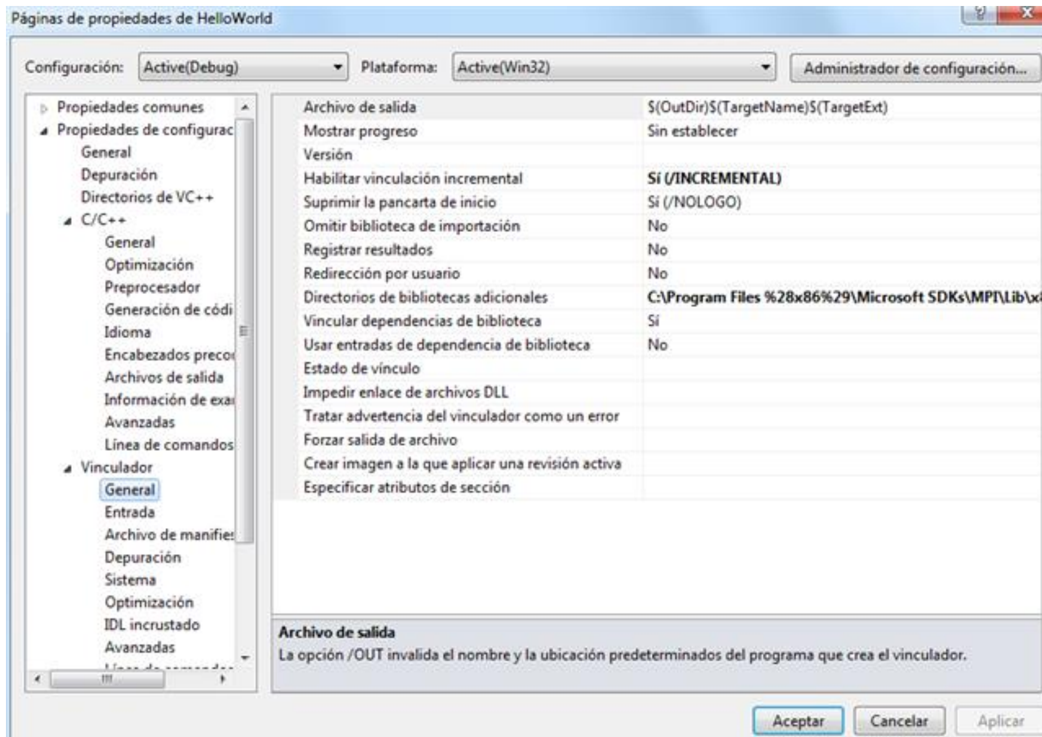


- When reaching the project configuration options proceed as follows:

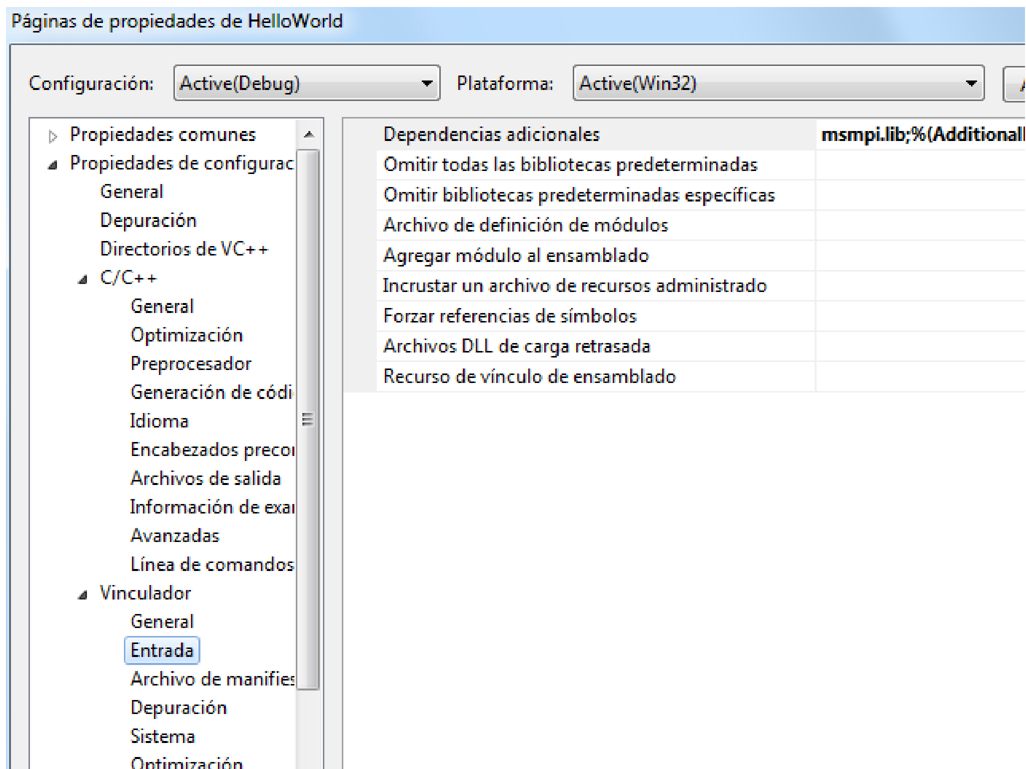
    1. Set the new additional include folder.



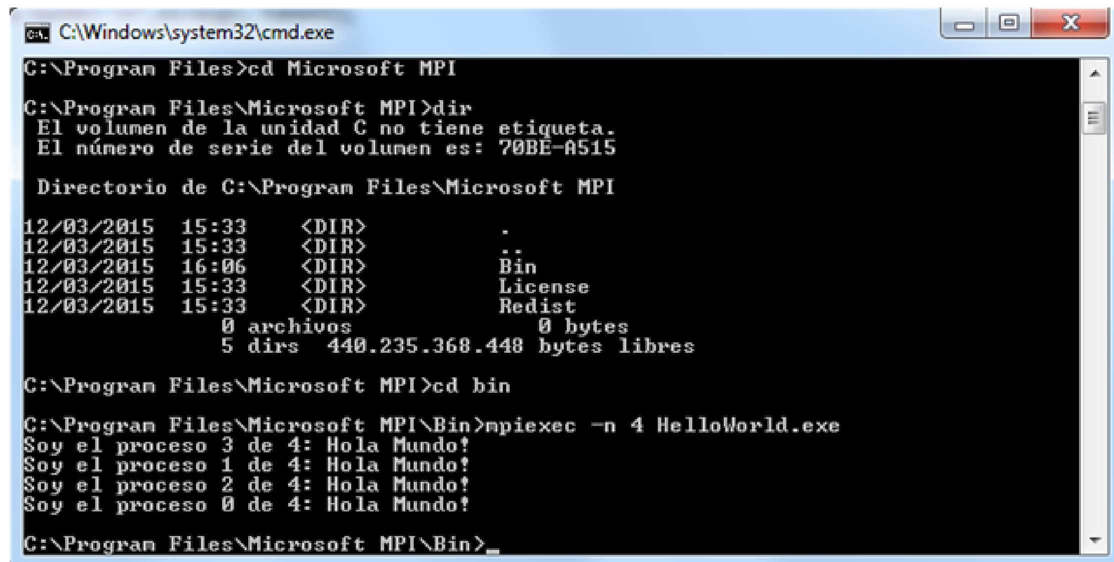
    2. Similarly set the new lib folder. Under lib choose the folder that matches your development (x86 for 32 bit applications or x64 for 64 bit ones).



3. Set also the new library file.



4. When all these parts have been configured the solution can be built as usual. In order to execute the program, the .exe file and MPI's launcher must be in the same folder or either the path configured accordingly. The launcher is mpiexec.exe and is placed in Program Files > Microsoft MPI > bin. Write down mpiexec –n np program.exe, where np is the number of processes to be launched.


C:\Windows\system32\cmd.exe
C:\Program Files>cd Microsoft MPI
C:\Program Files\Microsoft MPI>dir
El volumen de la unidad C no tiene etiqueta.
El número de serie del volumen es: 70BE-A515
Directorio de C:\Program Files\Microsoft MPI
12/03/2015 15:33 <DIR> .
12/03/2015 15:33 <DIR> ..
12/03/2015 16:06 <DIR> Bin
12/03/2015 15:33 <DIR> License
12/03/2015 15:33 <DIR> Redist
0 archivos 0 bytes
5 dirs 440.235.368.448 bytes libres
C:\Program Files\Microsoft MPI>cd bin
C:\Program Files\Microsoft MPI\Bin>mpiexec -n 4 HelloWorld.exe
Soy el proceso 3 de 4: Hola Mundo!
Soy el proceso 1 de 4: Hola Mundo!
Soy el proceso 2 de 4: Hola Mundo!
Soy el proceso 0 de 4: Hola Mundo!
C:\Program Files\Microsoft MPI\Bin>_