An Introduction To MPI

1 Why Parallel Computing?

When a long series of identical computations is to be performed, such as those required for the formation of numerical tables, the machine can be brought into play so as to give several results at the same time, which will greatly abridge the whole amount of the processes.[1]

One reason for using Parallel Computing is economic. By making use of off the shelf components, parallel computers can offer higher performance al lower prices than machines which use specially developed processors. In addition, the inherent scalability of parallel computers allow for them to be upgraded as the need arises. Whereas serial architectures are upgraded by making the previous processors obsolete, parallel architectures can, in theory, be upgraded simply by adding more processors.

However, there exists another reason, fundamental physics law, which will ultimately limit the speed of a single processor, irrespective of the economics. Movement of information forms the basis of a computer, but the speed of this movement is eventually limited by the speed of light.

2 Message-Passing Concepts

2.1 Introduction

The main concepts needed to build and program a serial computer are well understood. A physical device called a *processor*, is connected to a memory as illustrated in 1. The data in the memory can be read or overwritten by that processor.

In the *message-passing programming model*, each process has a local memory and no other process can directly read from or write to that local memory; the paradigm is illustrated in 2.

Material realizat de Cristina Marinescu

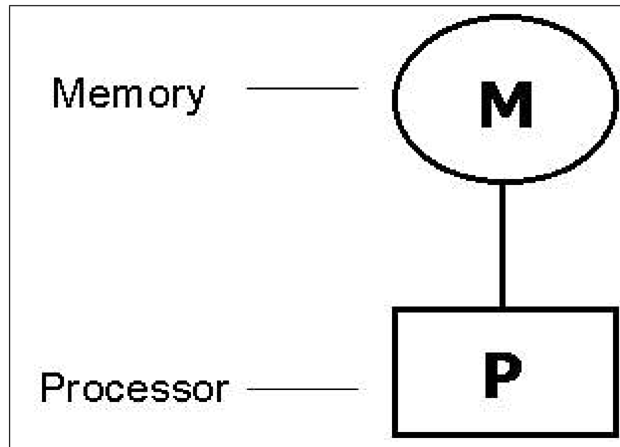

Figure 1: The sequential programming paradigm

2.2 Messages

Parallel programming by definition involves cooperation between processes to solve a common task. There are two sides to the question of programming processes that co-operate with one another. The programmer has first to define the processes that will be executed by the processors, they also have to specify how those processes are to synchronized and exchange data with one other. A central point of the message-passing model is that the processes communicate and synchronize by sending each other messages. As far as the processes are concerned, the message passing operations are just calls to a message passing interface that is responsible foe dealing with the physical communication network linking the actual processors together.

2.3 Message-passing programs

Let us now turn to the question of how a message-passing program might be written by a programmer. The most frequently encountered situation with multicomputers, is for the programmer to write a single source code program that is compiled and linked one a front-end computer. The resulting object code is copied in the local memory of every processor taking part in the computation. The parallel program is executed by having all processors interpret the same object code. This model of parallel computing is known as *Single-Program-Multiple-Data* model, or SPMD for short. If all the variables with the same name in every process had the same value and if all these processes only had access to the same data there would no parallelism at all, each process would act exactly the same as all others. In a message-passing

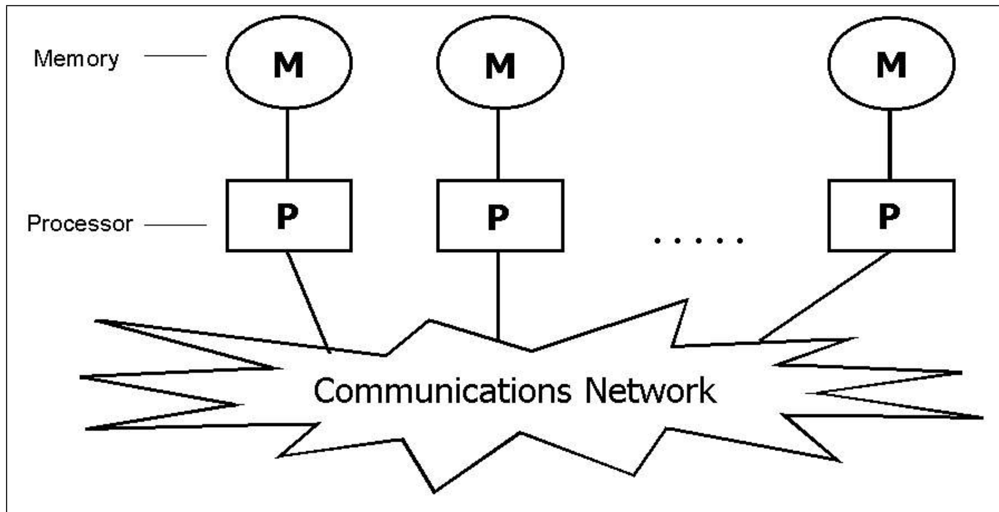

Figure 2: The message-passing programming paradigm

context, the communication interface typically includes a function or a procedure that returns a value identifying the process that called that function or procedure. Further more the same value will always be returned to the same process; this returned value is called the *process identifier*. Typically the sequence of instructions executed by a process will be determined by the input data and the identifier of that process.

As an ilustration, consider r̃ef3 below which shows the local memory of two processes. The local variables called whoami in each processes have been set somehow to the identifier of the corresponding process.

When the two processes execute the following statement, process A will increment the value of its variable x but the value of the variable x in the local memory of process B will remain unchanged:

```
if (whoami == 0) x = x + 111;
```

The local memories of the two processes will then look as shown in r̃effig4.

3 Basic Messages Concepts

In the message-passing model of parallel computation, the process executing in parallel have separate address spaces. Communication occurs when

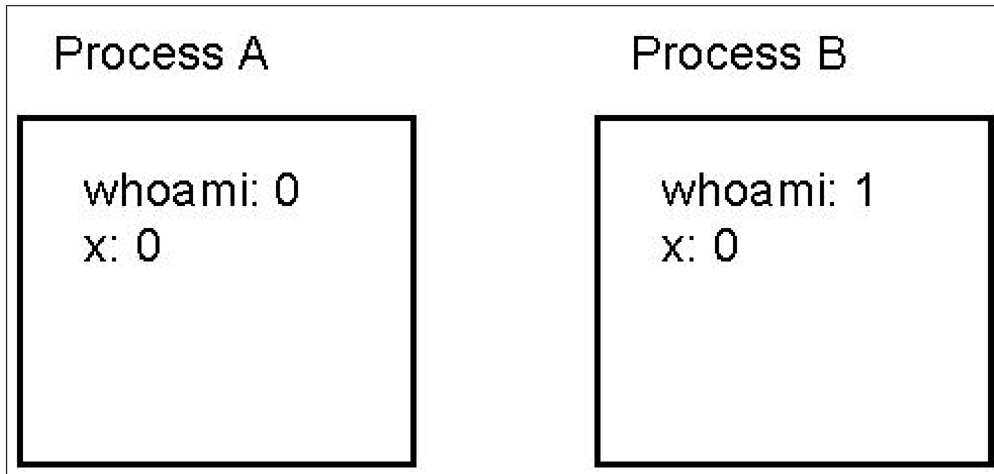

Figure 3: The local memories of two processes A and B

a portion of one process's address space is copied into another process's address space. This operation is cooperative and occurs only when the first process executes a *send* operations and the second executes a *receive* operation. **What are the minimal arguments for the send and the receive functions?**

For the sender, the obvious thing that must be specified are the data to be communicated and the destination process to which the data is to be sent. The minimal way to describe data is to specify a starting address and a length(in bytes). Any sort of data item might be used to identify the destination; typically it has been an integer.

On the receiver side, the minimum arguments are the address and length of an area in local memory where the received variable is to be placed, together with a variable to be filled in with the identity of the sender, so that the receiving process can known which process sent it the message. Although an implementation of this minimum interface might be adequate for some applications, more features usually are needed. One key notion is that of *matching*: a process must be able to control which messages it receives, by screening them by means of another integer, called the *tag* of the message. It is useful for the receiver to specify a maximum message size but allow shorter messages to arrive. An MPI message buffer is defined by a triple (address, count, datatype), describing count occurrences of the data type datatype starting at address. The power of this mechanism comes from the flexibility

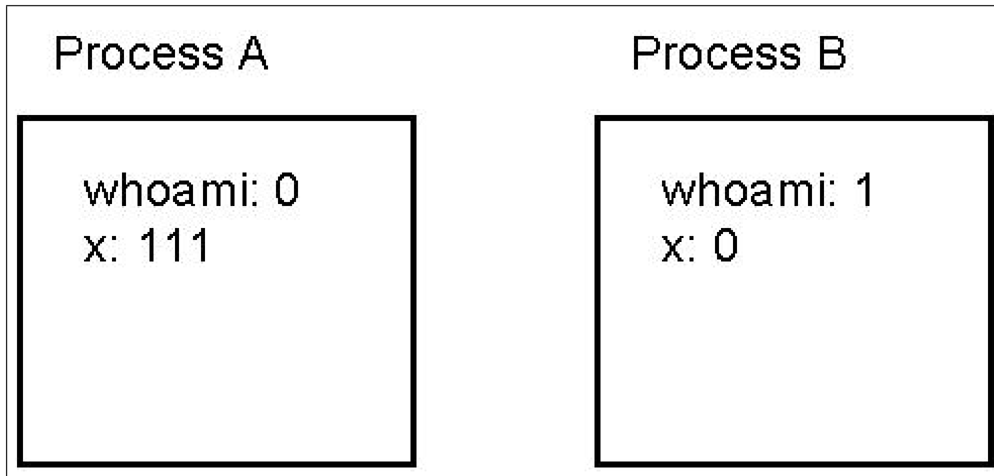

Figure 4: The local memories of the same processes after they both completed the above condition statement

in the values of datatype.

Processes belongs to *groups*. If a group contains n processes, then its processes are identified within the group by *ranks*, which are integers from 0 to n-1. There is an initial group to which all processes in an MPI implementation belong.

4 Is MPI Large or Small?

```
MPI_Init
MPI_Comm_size
MPI_Comm_rank
MPI_Send
MPI_Recv
MPI_Finalize
```

References

[1] L. F. Menabrea. *Sketch of the Analytical Engine.* 1842.

[2] W. Gropp, E. Lusk, A. Skjellum. *Using MPI. Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, 1999.