Chapter 4

Parallel linear solvers

4.1 Introduction

The reliability of the engineering community on CFD is growing due to the ability to solve complex fluid flow and heat transfer phenomena with accuracy and within a reasonable elapsed time. During the past decade, the increase of the speed of processors and memories has contributed to the reduction of this time, and hence, it has enabled to afford large engineering problems. However, while the complexity of such problems grows, the improvements in processor technology become physically limited. For that reason, only parallelism is able to boost performance significantly and to deal with long time and large memory consuming problems.

4.1.1 Hardware for parallel computing

Among the different architecture types derived along the past two decades, only a few of them have become nowadays the commonly used machines for scientific computing.

Following the classification of computers introduced by Flynn (1972), the most recent architectures fit into the SIMD (Single Instruction stream/Multiple Data stream) and MIMD (Multiple Instruction stream/Multiple Data stream) categories. The term stream relates to the sequence of data or instructions as seen by the machine during the execution of a program. However, it was not until the early to mid 1980s that machines conforming to the MIMD classification were available commercially. At this point, it is worth distinguishing between two MIMD categories, shared memory and distributed memory MIMD computers. The shared memory machines are considered to be tightly coupled, whilst the distributed memory machines are regarded as loosely coupled and employ the relatively slow message passing approach. See [56] for a comparative study of various computers.

More recently and following the development of distributed memory computers, there is an increasing interest in the use of clusters of workstations [57] connected together by high speed networks. The trend is mainly driven by the cost effectiveness of such systems as compared to large multiprocessor systems with tightly coupled processors and memories.

Although the numerical algorithms presented throughout this work are developed for any MIMD machine, the validation and measures of performance are carried out only for distributed memory machines. A Cray T3E with 32 tightly coupled processors and a 32 PC

cluster with fast ethernet based network are used for the numerical experiments. Further details of these computers may be found in the appendix.

4.1.2 Parallel programming models

The two basic models used in parallel computing are the Single Program Multiple Data (SPMD) model and the Multiple Program Multiple Data (MPMD) model. For our purpouses, i.e. CFD problems, the model most commonly used is SPMD. In this approach, each processor p runs an identical program but only computes on its own data $data_p$. To do so, the whole data or computational domain is distributed over all the processors, say np. For CFD problems, the distribution of data is done via domain decomposition, as it will be described later on. Furthermore, since the communication of data among processors may be necesary, each processor knows who is it and who are its neighbour processors nbg_p. Hence, the SPMD model is written in Alg. 19 as follows.

Algorithm 19 SPMD model

get *a processor, an identification number from* $1, .., np : p$
set *the neighbour processors:* ngb_p
get *a portion of data from the np-partitioned domain:* $data_p$

compute *on* $data_p$
communicate *with* ngb_p

This model enables the programmer to write the same or different operations (both computations and communications) for each processor. This depends on the parallelization of the operations.

For example, the matrix-vector product, say $y = Ax$, where A, x and y represent the whole computational domain is partitioned into np subdomains containing the data: A_p, x_p and y_p for $p = 1.., np$. In order to perform the global product, each processor performs its own subproduct and exchanges data with its neighbour processors ngb_p where needed. Since all processors does the same this operation is summarized with independence of the processor in Alg. 20.

Algorithm 20 SPMD example: y=Ax

get *a processor, an identification number from* $1, .., np : p$
set *the neighbour processors:* ngb_p
get *a portion of data from the np-partitioned domain:* $data_p$
A_p=partition(A, np)
x_p=partition(x, np)
y_p=partition(y, np)

exchange *data with processors* ngb_p
evaluate *the matrix-vector product with the p-portion of the domain*
$y_p = A_p x_p$

Although Alg. 20 will be explained in detail later on, it is executed at each processor doing more or less all processors the same, and hence, taking similar timings. However, this fact depends on the load of processors in both the computation and communication sense. It is worth noting that a significative delay among processors would produce bottlenecks. Hence, in order to ensure the high efficiency of the parallel algorithm, a few synchronizations, like barriers, must be introduced in strategic points and for all processors. By doing so, the processors drop the continuously unbalanced load that may arise during the computation or communication of several, say k, consecutive steps. The synchronization points are introduced implicitly in the communication steps such as the represented in Alg. 21.

Algorithm 21 Synchronization of computation and communication

computation (1)
communication + synchronization (1)

computation (2)
communication + synchronization (2)

...

computation (k)
communication + synchronization (k)

More precisely, the communication is composed by an exchange of data followed by a synchronization. At the exchange step, there is a pair of send and receive processes from processor p with the neighbour processors ngb_p. Once a processor has finished these tasks, it waits until the rest have done their respective tasks. This procedure is detailed in Alg. 22.

Algorithm 22 Communication + synchronization

communications of processor p
  send *local data to neighbour processors* ngb_p
  receive *local data from neighbour processors* ngb_p

synchronization with all processors
  wait *for all processors*

MPI implements this algorithm in different manners as it will be discussed in detail later on.

Although most of the parallel algorithms implemented under the SPMD model do the same operations for all processors, there are few operations that, for their implementation, require a hierarchy of processors, and hence, a different set of operations. As an example, we refer to the master-slave paradigm.

An example of the master-slave paradigm is the following. There is a master processor (often it is identified with $p = 0$) which collects the data sent from the rest of processors, named slaves (for $p = 1, \ldots, np$). The master carries out a set of operations which are

considered global operations, and finally, if it is required, the final result is distributed back to all the slaves. As represented in Alg. 23, there is an idle step for the group of slave processors while the master is doing its work. Therefore, a synchronization point at the end of the algorithm must be introduced in order to drop the arising bottlenecks.

Algorithm 23 Master-slave paradigm

if (*p=master*)
  receive *local data from p =slaves*
  compute *master operations*
  send *global data to p =slaves*
end if

if (*p=slave*)
  send *local data to p =master*
  keep *waiting idle*
  receive *global data from p =master*
end if

synchronize *master and slaves*

The inner product of two vectors can be coded as an example of the master-slave paradigm. Although all processor compute a part of the inner product, namely local inner product, one processor (the master) collects these partial results and compute a global summation. After that, the master distributes back to the rest of processor (the slaves) the resulting value. However, there are better implementations of this operation by minimizing the number of messages and hence, improving the efficiency. Other operations that can be implemented with the master-slave paradigm are the input(read)/output (write) of global results from/to a file disk respectively.

4.1.3 Message-passing programming

On these distributed memory machines, the parallel programming model is based on explicit message-passing programming. The first MIMD machines implemented proprietary message-passing libraries, basically specifying the sending and receiving messages between processors and grouped in two categories of communication: the point-to-point communication and the group communication. Although the concepts of these proprietary libraries were equal, the no portability between platforms was not considered as ideal. For that reason, research efforts have been conducted to develop portable and standard message-passing libraries. Currently, the PVM (Parallel Virtual Machine) and more recently, MPI (Message Passing Interface) are adopted by all of the parallel computer vendors.

Since MPI is becoming de-facto standard for message passing, it has been set for the implementation and description of the communication subroutines of this work.

4.2 Performance measurements of an implementation

Once an algorithm is implemented for the parallel execution the next step is the evaluation of the parallel performance for np processors. To do so, we compare the wall clock time of the computation of the parallel implementation, denoted by $\tau(np)$ with the wall clock time of computation of the sequential version, namely $\tau(1)$. We call this rapport the efficiency $E(np)$ of the parallel implementation for np processors.

$$E(np) = \frac{\tau(1)}{np\tau(np)}$$

If a parallel implementation with np processes was ideally 100% efficient, it means a np reduction of the time of computation respect to the computation with one processor. This reduction is called the speed-up $S(np)$ and it is measured as

$$S(np) = npE(np) = \frac{\tau(1)}{\tau(np)}$$

This is the ideal case but under certain circumstances, it is also possible to obtain a super linear speed-up. This usually occurs when the sparsity of a matrix is exploited on a parallel architecture or advantageous caching occurs.

It should be noted that there is currently some debate as to which sequential time, the parallel computation should be compared to. In this work the sequential time has been evaluated from the serialized version $np = 1$ of the parallel implementations.

The efficiency and speed-up are closely related and give an indication of how well balanced the computational load is and how the problem scales. In practice, an ideal speed-up cannot be attained and there are several reasons for this, which are:

- Inherent sequential parts
- Unbalanced load
- Overhead of the communication and synchronization

The first item refers to the those parts of the algorithm that may not be perfectly parallel. For example, the inner product between two vectors and the factorization of a matrix. The second one refers to a the different distribution of load among processors. For example, the computational domain may not be equally distributed, or some operations require a master which performs more tasks than the slaves. These factors may be more or less controlled by our implementation. However, the major factor that contributes to the decrease of the efficiency of the parallel implementation is the overhead due to the exchange of data among processors. Each time a message is sent, an overhead in timing cost is incurred. Therefore, if these factors are included in the efficiency formula, it is possible to point out the drawbacks to a given implementation.

For instance, an improved measure of the efficiency is based on an accurate description of the operations and the time spent in each operation. This is in essence the measure of the sequential algorithm. For the parallel algorithm, we have to add the time spent in the communication processes needed in some operations. Then the efficiency is computed as

$$E(np) = \frac{n_o(1)\tau_o}{np(n_o(np)\tau_o + n_c(np)\tau_c(np))}$$

Where n_o stands for the number of operations, τ_o the time spent in one of these operations, $n_c(p)$ the number of communications and $\tau_c(np)$ the time spent in one of these communications. Notice that the time spent in the communication process depends on the number of processors p. More processors means more number of communications $n_c(np)$ but the quantity of data transferred per communication and processor is reduced. Furthermore it is convenient to express more accurately the time of communication τ_c split into two parts: the proper time of communication when sending a packet of data with size d at β^{-1} rate of communication and the time of setting up the communication τ_s.

$$\tau_c(np) = \tau_s + \beta d$$

Since the bandwidth β and the latency τ_s are hardware dependent parameters, the efficiency would be very different when it is evaluated in either tightly coupled processors or loosely coupled processors. Introducing these concepts in the previous expression the efficiency leads to

$$E(np) = \frac{n_o(1)\tau_o}{np(n_o(p)\tau_o + n_c(np)(\tau_s + \beta d(np)))}$$

Having a look at this expression and assuming that $n_o(1) \approx npn_o(np)$ in our algorithms, we see that the efficiency is always less than 100% and it decreasses as np increases. The number of communications $n_c(np)$ increases linearly with np, and although the time of communication of data $\tau_c(np)$ decreases because the overall data are better distributed, the latency remains constant and hence the overall time of communication per process increases.

If we partition our domain with a certain overlapping, the number of operations performed in sequential and in parallel per processor np is different. If we express the number of operations per process $n_o(np)$ in terms of the number of control volumes handled by the processor $n_{cv}(np)$ multiplied by the number of operations per control volume $o_{cv}(np)$, we get

$$n_o(np) = n_{cv}(np)o_{cv}(np)$$

and substituting it into the efficiency, we obtain an expression with three component effects: algorithm $E_a(np)$, load $E_l(np)$ and communication $E_c(np)$.

$$E(np) = E_a(np)E_l(np)E_c(np)$$

Where

$$E_a(np) = \frac{o_{cv}(1)}{o_{cv}(np)}, \quad E_l(np) = \frac{n_{cv}(1)}{npn_{cv}(np)}, \quad E_c(np) = \frac{1}{1 + \frac{n_c(np)}{n_o(np)}\frac{\tau_s + \beta d(np)}{\tau_o}}$$

The first effect means the rapport of operations performed in the sequential algorithm versus the parallel algorithm. For Krylov solvers, we consider an equal number of operations so this effect is neglected. The second effect takes account of the overlapping ov. Each processor contains its own data plus an overlapping. However for large scale problems the overlapping is much smaller than the data contained in the processor so this effect is reduced. The third effect is the rapport between the amount of work done of computation and communication per processor. Therefore, the efficiency not only depends on the parallel implementation but also on the parameters of communication, i.e. the latency τ_s and the bandwidth β^{-1} which are hardware dependent.

Finally, there is another performance measure so called scalability, which is related with the computing time of the problem with np and how it increase when increasing in the same ratio (e.g. doubling) both the problem size and the number of processors. It is clear that, for an ideal scalability of the implementation, the computing time will remain constant. Hence, the problem may be scaled by two, four, and so on, by means of $2np$, $4np$, ... processors.

Thus, the scalability may be computed as

$$Sc(np) = \frac{\tau_{np} size_{np}}{\tau_{np} size_1}$$

4.3 Modellization of the communication time

As mentioned above, the loss of performance of the parallel algorithm depends directly on the time spent in the communication of data between processors. A low communication cost is always desirable for high performance computing. The time of communication while sending a block of data is affected by two parameters the latency or setup time τ_s and the bandwidth or byte transferred rate per second β^{-1} . The correlation between the time of communication τ_c (given in seconds) and the transferred data d (given in bytes) was expressed in the first approximation by a linear equation:

$$\tau_c = \tau_s + \beta d$$

Indeed, both parameters depend clearly on the hardware (netcards, switch and crossbars) and the software (operating system, TCP/IP and MP library implementations, compilation and executing flags). Details about the influence of such hardware and software issues over the communication performance are out of the scope of this work.

An approximate evaluation of the latency and bandwidth parameters can be done by sending blocks of data with different sizes, and measuring the time spent in the communication [56]. This procedure or test is commonly named microbenchmark of the network because it transfers small data packets that takes short time of communication. Since the measure of time can be either intrusive or do nor have enough accuracy, the time of communication is enlarged by returning the same packets of data (i.e. a round-trip of the packets) and halving the measured time. The test does a thousand of experiments by sampling the size of the packed data randomly within a range of 0-1MB and measuring the time of the round-trip. For instance, the type of data transferred is the *char*, which size in bytes is the unit.

In order to ensure non overlapped send and receive process which can reduce the time of the round-trip, the block communication mode is used. Alg. 24 summarizes this test.

Algorithm 24 Latency and bandwidth parameters

```
for (sample = 1 to sample ≤ 1000)
   set size[sample]=random(0,1000000)
   synchronize processors
      MPI_Barrier(MPI_COMM_WORLD)
      time the round-trip
         t0=MPI_Wtime()
         if (fmod(rank,2)=0)
            MPI_Send(char_send,size[sample],rank+1,..,..)
            MPI_Recv(char_recv,size[sample],rank+1,..,..)
         else
            MPI_Recv(char_recv,size[sample],rank-1,..,..)
            MPI_Send(char_send,size[sample],rank-1,..,..)
         end if
         t1=MPI_Wtime()
         time[sample]=(t1-t0)/2
end for
```

The MPI_Barrier step ensures the synchronization of the measured times and therefore reduces the dispersion of results. Furthermore, this test is executed by a wide range of even numbers of processors 2,4,6,8 obtaining results (see Fig. 4.2) without significative differences.

This figure shows the linear behaviour of the communication time of the transferred pack of bytes. The difference of slopes for each facility indicates different byte rates per second. For the estimation of the slope, the range of large messages (1KB,..,1MB) has been used. And the inverse of the slope is the estimated bandwidth.

For the estimation of the latency parameter, there are two possibilities. The first one is taking the time value for a zero pack of bytes. But as mentioned before, time measure may be inaccurate. In a second approach, a zoom (see Fig. 4.1) near the short transferred pack of bytes (0,..,1KB) is used with a linear fit regression. The latency parameter is then estimated for a zero pack of bytes. The different values of latency and bandwidth parameters for both facilities are summarized in the table 4.1.

Facility	Latency (1) μsec	Latency (2) μsec	Bandwidth MB/sec
PC cluster	226.98	218.95	10.46
Cray T3E	23.17	21.18	119.98

Table 4.1: Latencies measured (1) and estimated (2) and bandwidth estimated.

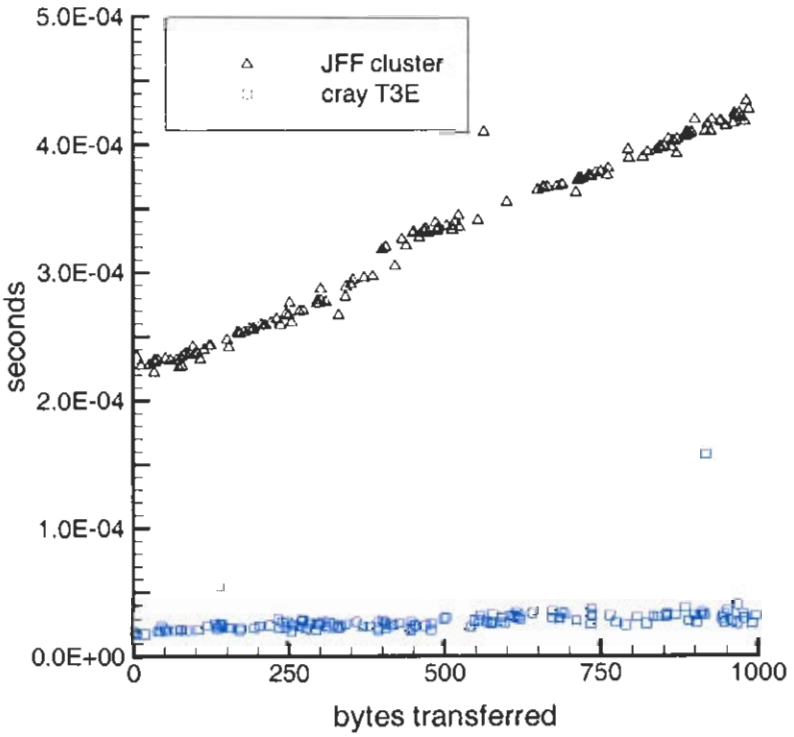

Figure 4.1: Zoom of previous figure near the short transferred pack of bytes.

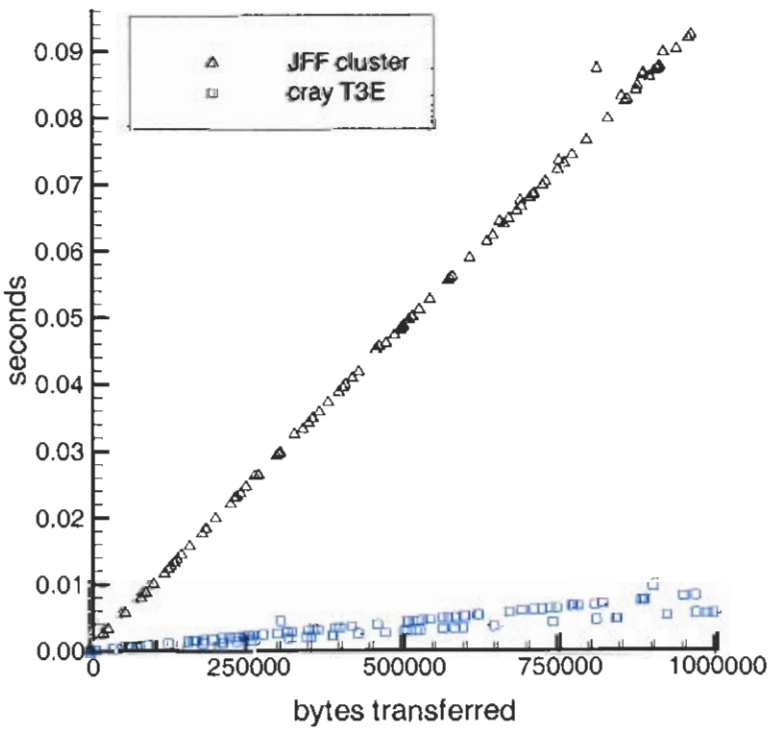

Figure 4.2: Timing results on two facilities the PC cluster and the Cray T3E.

These values clearly show the great difference of communication performance between facilities. The low latency and large bandwidth of the Cray T3E explain in part the high price against the PC cluster.

4.4 Communication modes

The exchange of data among processors can be performed over two different modes of communication: blocking and non blocking. The blocking communication mode disables the processors to perform other operations while the communication is being done. Since the process of sending or receiving the data involves the access to the buffer of memory, MPI library protects the data by blocking the processor and hence, avoiding the access to the data for reading or writing. The blocking communication mode among processor p and its neighbours ngb_p for a given buffer of data, say $data_p$, is described in Alg. 25.

Algorithm 25 Blocking communication

send $data_p$ *to neighbours* ngb_p
  block_send $(data_p, ngb_p)$

receive $data_p$ *from neighbours* ngb_p
  block_receive $(data_p, ngb_p)$

Since the communications in this mode follow the order of the algorithm, processors either the sender or the receiver keep idle waiting for completion of the communication processes, and hence, resulting in poor efficiency. However this mode has an implicit synchronization procedure so it is not necessary to explicitly synchronize the processors after the ends of the pairs of send and receive processes.

The order of the communication processes among processors has to be considered carefully. A very important fact in a bad scheduled communication is the dead lock. It appears when two or more processors send and receive data in a wrong order. When this happens, the processors do not understand themselves and keep waiting infinitely for the communication request. The dead lock may be avoided by organizing the communication in two simple steps. Firstly, all procesors send their respective data to their respective receivers, i.e., the neighbour processors. Any order for the senders could be defined. Secondly, the senders are ready to receive the data from their respective receivers. The dead locks are avoided by ensuring the existence of pairs of send and receive messages. If not, it will appear to have an orphan message process (a sender without a receiver or a receiver without a sender) and thus resulting into a dead lock. The use of tags is convenient for the right schedule of sending and receiving data.

The other mode of communication to be used in data exchange is the non blocking communication. Each processor copies the data to communicate x_p to another buffer y_p. Then MPI library acts on that data without blocking the processors to perform other operations on the original buffer. The non blocking communication for the same vector is described in Alg. 26.

Algorithm 26 Non blocking communication (x_p)

copy *data from* x_p *into another buffer* y_p

send *data of* y_p *from* p *to neighbours* ngb_p
non_block_send (y_p, ngb_p)

operate *over* x_p

...

receive *data in* y_p *from neighbours* ngb_p *to* p
non_block_receive (y_p, ngb_p)

operate *over* x_p

...

synchronize *all processes*
wait *for all*

copy *data from* y_p *into the buffer* x_p

However, this procedure could reduce the performance of parallel algorithms due to the elapsed times of the pre step of copying data from the buffer x_p to the buffer y_p and, after the communication step, the post step of copying back the data to the original buffer. In addition, for large amounts of data, the performance of buffering decreases due to the cache missings.

Furthermore, the data in x_p to be exchanged, e.g. halos and inside blocks, are usually located in non contiguous locations of the buffer. Therefore the data must be arranged continuously in the buffer y_p before any communication. After the communication is done, the data contained in the buffer y_p must be redistributed to x_p in the non contiguous locations.

In order to see the differences of performance of both modes of communication, a second test is done. This test (see Alg. 27) shows the performance of the implementation of an exchanged pack of bytes.

Algorithm 27 Blocking and non-blocking communication modes

```
for (mode=blocking to mode=non-blocking)
  for (sample=1 to 1000)
    size[mode][sample]=random(0,1000000)
    MPI_Barrier(MPI_COMM_WORLD)
    t0=MPI_Wtime()
    if (mode=blocking)
      if (fmod(rank,2)=0)
        MPI_Send(char_send,size[mode][sample],rank+1,..,..)
        MPI_Recv(char_recv,size[mode][sample],rank+1,..,..)
      else
        MPI_Recv(char_recv,size[mode][sample],rank-1,..,..)
        MPI_Send(char_send,size[mode][sample],rank-1,..,..)
      end if
    end if
    if (mode=non-blocking)
      if (fmod(rank,2)=0)
        MPI_Isend(char_send,size[mode][sample],rank+1,..,..)
        MPI_Irecv(char_recv,size[mode][sample],rank+1,..,..)
      else
        MPI_Isend(char_send,size[mode][sample],rank-1,..,..)
        MPI_Irecv(char_recv,size[mode][sample],rank-1,..,..)
      end if
      MPI_Wait(send)
      MPI_Wait(recv)
    end if
    t1=MPI_Wtime()
    time[mode][sample]=t1-t0
  end for
end for
```

Regarding the blocking communication mode, it performs the send and the receive processes of a given pack of bytes consecutively. It is like the round-trip of the pack described in the previous test, but the measure of time is not halved.

If the network enables communications among processors in both senses at same time (i.e. duplex network), a pair of send and receive processes can be done in half of time. Conversely, the non-blocking communication mode enables one to do this pair of processes at same time and to wait until it completes the two processes independently.

Like in the previous test, a thousand of experiments with randomized packet sizes of type *char* and covering the range of (0,..,1MB) is done.

Due to the duplex network feature, the order of the non-blocking communication pro-

cesses within the algorithm does not affect the communication procedure and therefore no dead-locks could arise. The scope of the MPI_Wait step is to wait for termination of communication processes.

The test is carried out, analogously to the previous test, on both facilities PC cluster and Cray T3E, whose networks are duplex. The results are shown in Fig. 4.3.

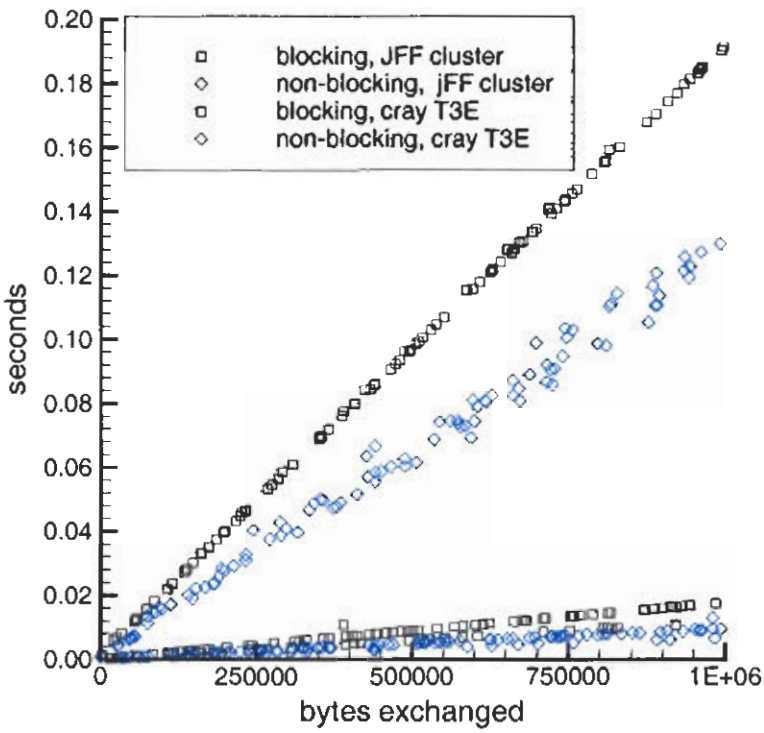

Figure 4.3: Blocking and non-blocking communication modes for the PC cluster and the Cray T3E.

Fig. 4.3 shows that the behaviour of both communication modes are similar in both machines but at different scales of time. The non-blocking communication of any pack of bytes is performed more efficiently and nearly at half time of the the time taken by the blocking communication. The byte exchanged rates (i.e. the inverse of the slopes), for each implementation and their rapports are given in table 4.2.

Therefore, it is convenient to use a non-blocking communication implementation for those operations where an exchange of information is required such the matrix-vector product. For that reason, MPI library provides a set of data types which enables to skip these pre and post copying steps, and hence, to improve the communication performance.

modes	PC cluster	Cray T3E
blocking (1) MB/sec	5.255	60.321
non-blocking (2) MB/sec	7.889	110.387
rapport: (2)/(1)	0.66	0.54

Table 4.2: Megabyte exchanged rates (MB/sec) and rapports for both modes of communication on the PC cluster and the Cray T3E.

4.5 Domain decomposition

The most popular approach to solving CFD problems on MIMD architectures whether it is memory shared or memory distributed is that of the domain decomposition [58]. The objective is to distribute the computational domain onto a number of processors such that the work load on each processor is equivalent. The practical application involves the discretization and the solution of the systems of equations derived in previous chapters on an equally distributed load per processor. The details of the implementation of the domain decomposition are given hereinafter from an algebraic point of view.

Most algebraic operations involved in the discretization and solution procedures are based on the addition, the subtraction and the product of three types of objects: scalars, vectors and matrices. The implementation of these operations on these objects can be thought to be performed either in sequential or in parallel. In a one-processor machine the operation and the storage of objects are done as it is expressed mathematically. However, in a MIMD machine, the algebraic operations are performed on a part of the objects. The vector object, for example, is equally distributed as much as possible and stored among all processors. This equally distribution of objects is stressed to balance the load, and hence, to obtain a good efficiency of the implementation.

From here on, we shall use subindex for distinct parts of the objects. If we have a np-processor machine where np stands for the number of processors and object, named o, the object is partitioned into np objects and stored at each processor. Labeling the resulting objects from $p = 1$ to $p = np$ it follows that

$$o = \bigcup_{p=1,\ldots,np} o_i$$

An algebraic operation, named for generality $\oplus$, between two objects x and y is performed in each process p using only the respective parts x_p and y_p. The result can be stored in another distributed object z.

$$z = x \oplus y \Longleftrightarrow z_p = x_p \oplus y_p,\ p = 1, \ldots, np$$

Thinking this way, it seems easy to implement parallel $np > 1$ or sequential $np = 1$ algebraic operations with distributed objects. Nevertheless, some operations like the inner product between two vectors or the 2-norm of a vector, the maximum and minimum value of the components of a vector, and the product of a matrix with a vector involve information stored in the closest processors or even in all processors. Therefore, this information must

be copied from these processors, named neighbour processors ngp_p to the affected processor p. This yields to an extra information per processor of the object.

$$ov \neq \bigcap_{p=ngb_p} o_p$$

Hereinafter, we shall call this additional information as the overlapping values ov. Most operations requires an overlapping of a single point, one line of points or a surface of points for a one, two or three dimensional object respectively. The implementation of these ideas to vectors and matrices are developed in next subsections.

4.5.1 Block vector

Let us suppose x to be a vector object which maps a d-dimensional domain Ω. The partition of this domain in np parts is carried out in an equally distributed manner among the np processors. Assuming that the domain Ω is discretized onto a structnred grid of points, the partition is performed easily following the orthogonal directions. For instance, a vector that maps a three dimensional domain (see Fig. 4.4) is partitioned in two orthogonal directions $p_1 = 2$ and $p_2 = 3$ leading into 6 block vectors x_p with $p = 1, \ldots, np = p_1 \times p_2$.

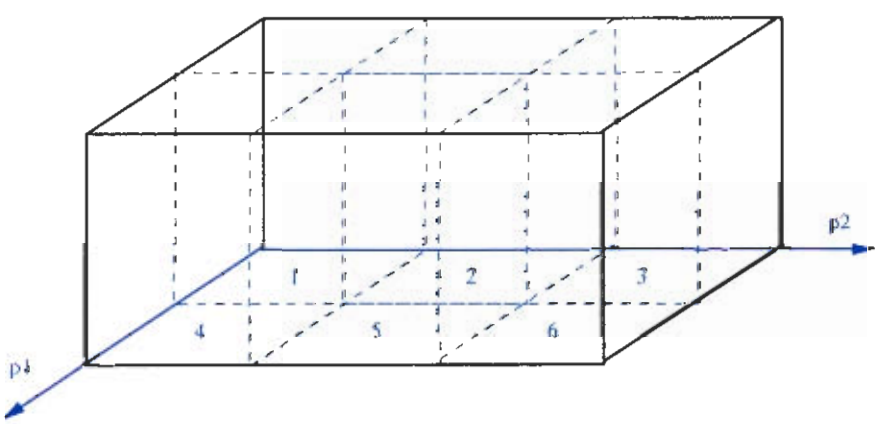

Figure 4.4: Two dimensional partition of a vector that maps a three dimensional domain.

Let $I \times J \times K$ be the overall grid points over the domain Ω, the block vector x_p maps the p-part of the domain at least in $(I/p_1) \times (J/p_2) \times K$ grid points. Notice that, a perfect load balancing in all directions is considered. An unbalanced partition leads into a generalized grid size $id \times jd \times kd$. Fig. 4.5 shows the mapping of x_p with a generalized index i, j, k.

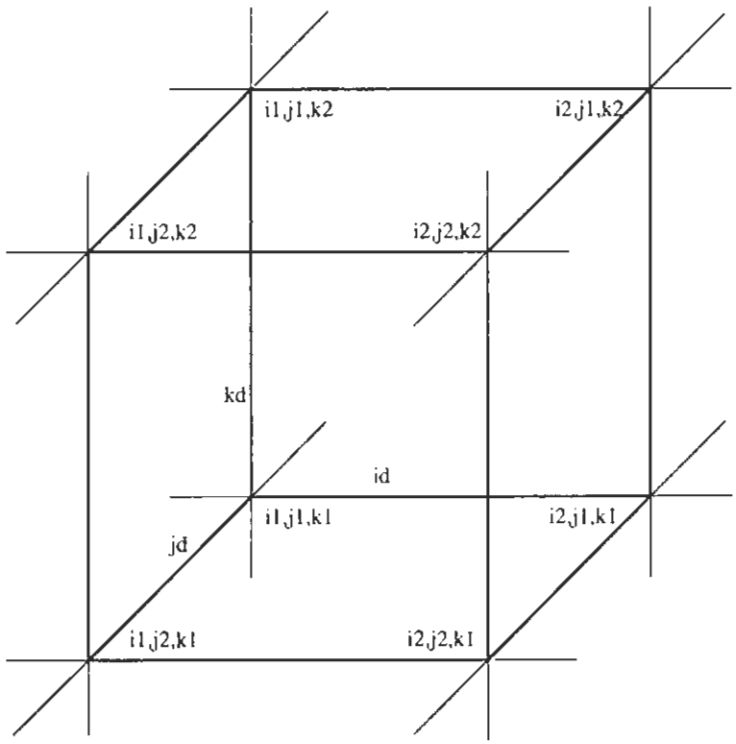

Figure 4.5: Generalized $id \times jd \times kd$ partition x_p of a three dimensional vector x. The index i, j, k has a range from $i1, j1, k1$ to $i2, j2, k2$.

The overlapping ov is added in those directions where the information of the neighbour processors is needed. The blue dashed lines in Fig. 4.6 show the overlapping areas among processors.

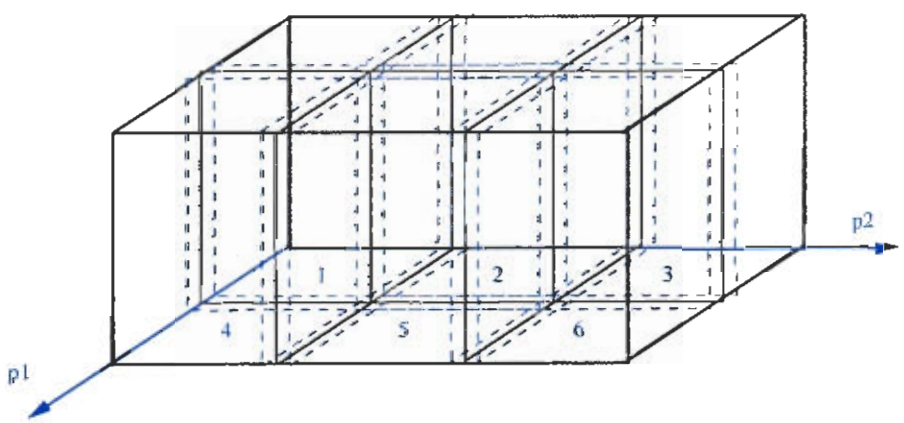

Figure 4.6: Overlapping areas ov for a 2×3 partitioned vector that maps a three dimensional domain.

The resulting dimension of a generalized x_p vector with grid size $id \times jd \times kd$ and with the same overlapping ov in all directions is represented in Fig. 4.7.

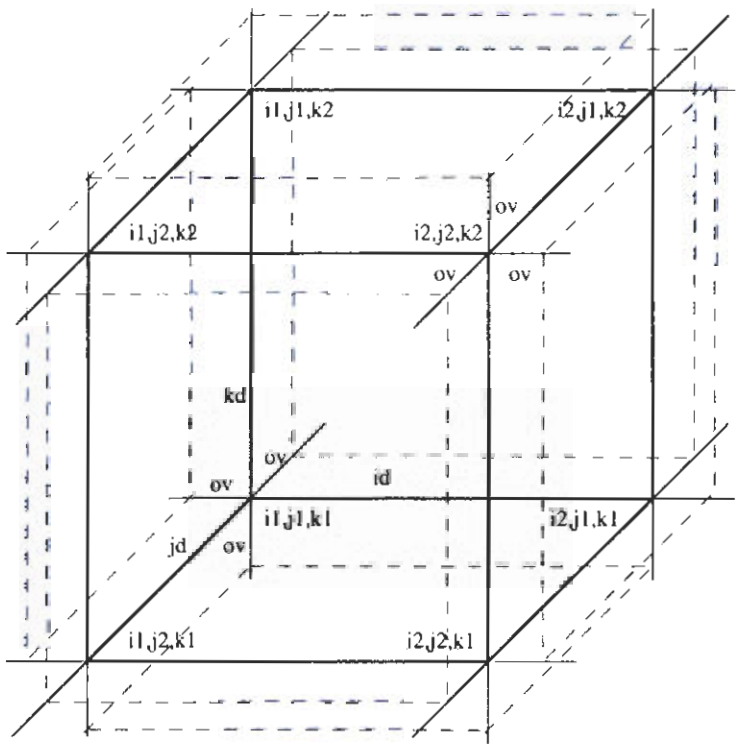

Figure 4.7: Overlapping areas ov for a generalized $id \times jd \times kd$ partition x_p of a three dimensional vector x.

Therefore, the three dimensional vector x that represents the domain Ω is expressed in terms of a set of np partitioned vectors x_p with a defined size $id \times jd \times kd$ plus an overlapping of ov. For practical implementation reasons, this overlapping is added to all block vectors and in all directions whether there are neighbour processors or not.

In order to gain clarity of the algebraic representation of the full vector x, the overlapping information is omitted yielding to a representation of the vector as follows

$$x = (x_1, x_2, \ldots, x_{np-1}, x_{np})^T$$

By doing so, it is easy to represent the operations with vectors. For instance, the copy of a vector x into another vector y is represented by omitting the overlapping as

$$y = x \Longleftrightarrow \left\{ \begin{array}{c} x_1 \\ x_2 \\ \vdots \\ x_{np-1} \\ x_{np} \end{array} \right\} = \left\{ \begin{array}{c} y_1 \\ y_2 \\ \vdots \\ y_{np-1} \\ y_{np} \end{array} \right\}$$

However, the implementation of this operation considers the overlapping. Alg. 28 gives such example considering an overlapping of ov.

Algorithm 28 Copy_vect(x_p, y_p)

```
for (i = i1 − ov to i = i2 + ov)
   for (j = j1 − ov to j = j2 + ov)
      for (k = k1 − ov to k = k2 + ov)
         y_p(i, j, k) = x_p(i, j, k)
      end for
   end for
end for
```

4.5.2 Block matrix

Let A be a N-square matrix such that $N = (I \times J \times K)$ arises from the discretization of the governing equations of a CFD problem onto a grid of $I \times J \times K$ points. We represent the matrix A partitioned by rows as balanced as possible in np blocks of matrices A_p:

$$A = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_{np-1} \\ A_{np} \end{bmatrix}$$

Each of these matrices A_p contains the matrix coefficients of the linear system $Ax = b$ associated with the partition p.

$$\begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_{np-1} \\ A_{np} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{np-1} \\ x_{np} \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{np-1} \\ b_{np} \end{Bmatrix}$$

In previous chapter, we described the different patterns of these sparse matrices for a natural ordering (i.e. the *7-point* and the *19-point* formulation in a three dimensional CFD problem. Since a narrowed band of coefficients are non zero, we shall store only the blocks of non zero coefficients of each partition.

Here, we used the following notation for a one dimensional partition:

$$\begin{bmatrix} A_1 & A_{1,2} & 0 & \cdots & & \\ A_{2,1} & A_2 & A_{2,3} & 0 & \cdots & \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ & \cdots & 0 & A_{np-1,np-2} & A_{np-1} & A_{np-1,np} \\ & & \cdots & 0 & A_{np,np-1} & A_{np} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{np-1} \\ x_{np} \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{np-1} \\ b_{np} \end{Bmatrix}$$

Having a look at the structure of this matrix each process p stores two different matrices A_p and A_{p,ngb_p}. Where ngb_p stands for the neighbour processes of process p. The first matrix indicates the operations performed with the information stored within the processor p and the second one indicates the operations performed with the information stored at neighbour processors.

For a generalized partition in two or three directions, the structure of blocks of matrix A the linear system is written as follows:

$$\begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_{np-1} \\ A_{np} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{np-1} \\ x_{np} \end{Bmatrix} + \begin{bmatrix} A_{ngb_1} \\ A_{ngb_2} \\ \vdots \\ A_{ngb_{np-1}} \\ A_{ngb_{np}} \end{bmatrix} \begin{Bmatrix} x_{ngb_1} \\ x_{ngb_2} \\ \vdots \\ x_{ngb_{np-1}} \\ x_{ngb_{np}} \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{np-1} \\ b_{np} \end{Bmatrix}$$

Thus, a set of communications between the processes involved in such operations must share the information of the ngb_p process to the p process. In the chapter ahead, we shall explain some issues of this communication between the different processes.

4.6 Exchange of data blocks

MPI data type has been used to send and receive at once the non contiguous data y_p of a vector x_p. MPI data type provides a new type of variables in order to send and receive blocks of information located in different points of the buffer at once. By doing so, we can reduce, on one hand, the number of communications and latency, and on the other, the number of cache missings in the buffering processes. A sender processor can explicitly pack noncontiguous data into a contiguous buffer and then send it. A receiver processor can explicitly unpack data received in a contiguous buffer and store in noncontiguous locations.

For simplicity, let us suppose that a x_p vector has dimensions 4×4 and the noncontiguous data y_p has dimensions 2×3 (see Fig. 4.8).

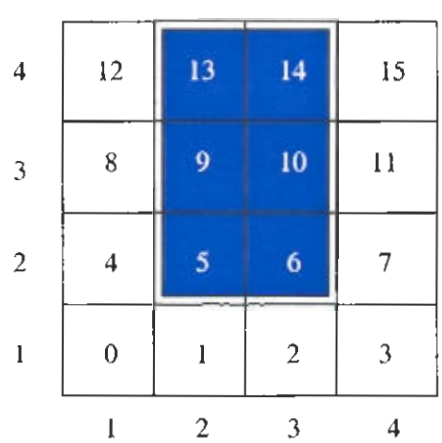

Figure 4.8: The 4×4 x_p vector and the 2×2 y_p data filled in blue. The numbers written within the x_p vector represent the order of data in the buffer.

A schematic representation of the buffer (see Fig. 4.9) shows the noncontiguous data y_p embedded in the map x_p.

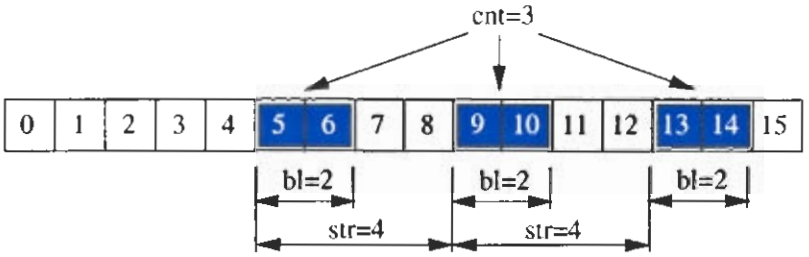

Figure 4.9: Buffer representation of x_p and the noncontiguous data y_p. The noncontiguous data follows a pattern of $cnt = 3$ noncontiguous blocks of length $bl = 2$ separated with an stride of $str = 4$.

Notice that the noncontiguous data follows a pattern of 3 noncontiguous blocks of dimension 2 separated with an stride of 4. This information is enough to construct a new data type with continuous data by packing the y_p data.

This idea has been easily extended to more complex noncontiguous data such as a three dimensional vectors, with two different number of blocks with different lengths and strides. Fig. 4.10 shows a three dimensional x_p vector with dimensions $I \times J \times K$ and a $i \times j \times k$ noncontiguous data y_p.

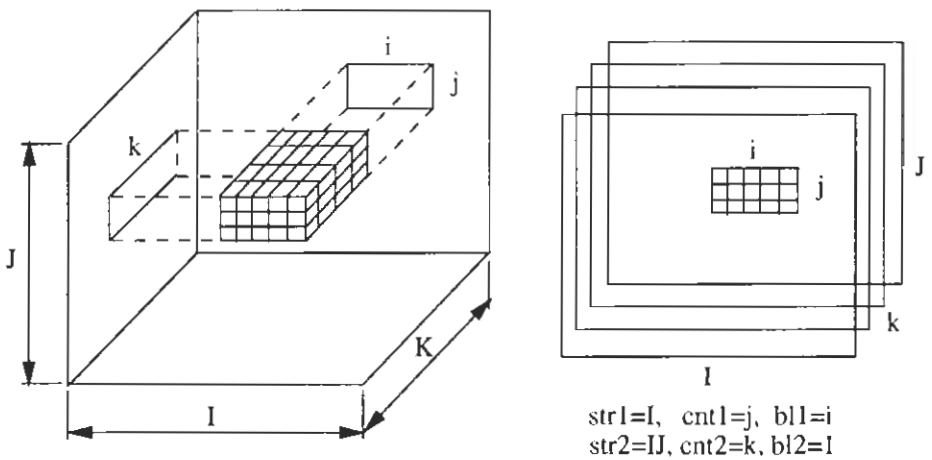

Figure 4.10: Representation of x_p vector and the noncontiguous data y_p. The noncontiguous data follows a pattern of two noncontiguous blocks cnt_1, cnt_2 with different lengths bl_1, bl_2 separated with strides str_1 and str_2 respectively.

In this case, we apply recursively two data types. The first one has j blocks of length i with an stride of I. The second one put over the first one, has k blocks of length 1 with an stride of IJ.

Finally, the data type implementation over a non blocking communication leads to Alg. 29, used in the product of a matrix by a vector.

Algorithm 29 Update (ov, x_p)

set the data type y_p *for noncontiguous data of* x_p
  data_type (ov, x_p, y_p)
send y_p *data from* p *to neighbours* ngb_p
  non_blocking_send (y_p, ngb_p)
compute *over* x_p

  ...

receive y_p *data from neighbours* ngb_p *to* p
  non_blocking_receive (y_p, ngb_p)
compute *over* x_p

  ...

synchronization of all process
  wait *for all*

Further details on data types can be found in MPI documentation available in internet [7].

The following test measures and compares the communication performance of two possible implementations of the *update*(ov, x_p) subroutine. Both implementations use the non-blocking mode of communication which has been tested to be the better. Furthermore, both implementations perform a copy of values to be exchanged to a buffer and then they are exchanged. By doing so, it is possible to perform operations with the original values and exchange a copy at same time. Therefore, this procedure increases the performance of operations because it overlaps the computation and the communication. A draft (see Fig. 4.11) of the processes shows this procedure.

However the implementation of the buffering process affects this overlapping. The original buffer contains the amounts of data in non-contiguous blocks and the copied buffer must contains the data in contiguous blocks before they are sent to another process. This process is called packing. Conversely, after the communication is done, the received pack of data contained in a contiguous buffer has to be unpacked at non-contiguous locations in the original buffer of the receiver process.

In order to reduce the time of the whole procedure of exchange of data, an efficient implementation of the packing and unpacking procedures is desired. A first implementation does an explicit copy of the non-contiguous blocks of data in a contiguous buffer. This implementation is tedious to implement: dynamic allocations, initializations and copying data from one buffer to the other. In addition, it may suffer possible overheads due to the cache missings. This implementation is called simply non-blocking.

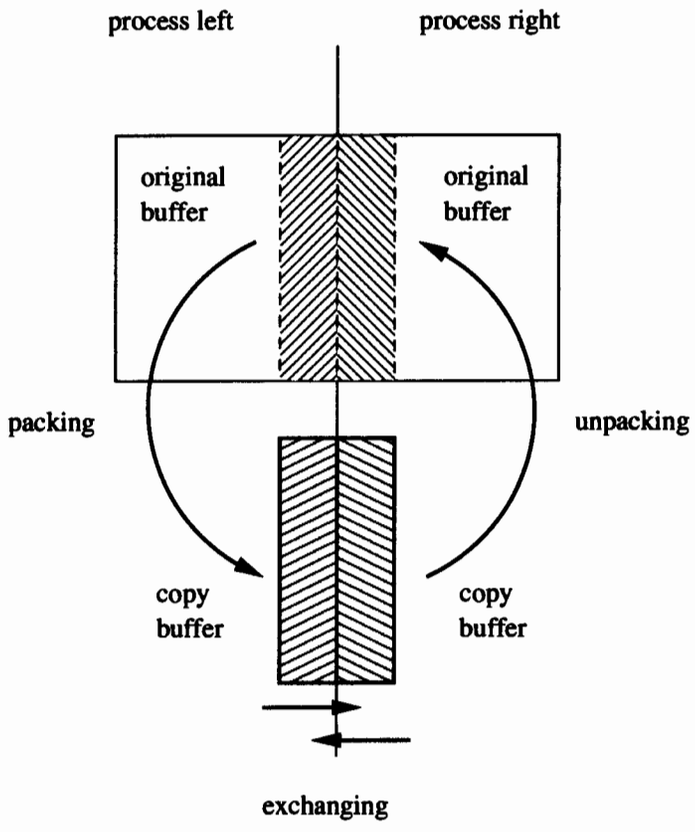

Figure 4.11: Pack, exchange and unpack procedure between two processes

A second and easier implementation use the MPI_datatype to do implicitly the copy from the original and non-contiguous buffer to a contiguous buffer, thus reporting coding and time savings. This implicit copy means that only a structure of pointers to the different locations of the non-contiguous blocks are stored in a MPI_datatype. After that, the MPI library performs the packing and unpacking processes implicitly and in an efficient way when the non-blocking send and receive subroutines are called.

Differences of both implementations are tested as follow. A three dimensional halo with random size in k direction is updated between two processes, left and right. It has been chosen to vary the k direction instead of i or j since the distribution of non-contiguous blocks is sparser and may produce more cache missings. See Fig. 4.12 for details of block data sizes and graphical explanation of the communication process.

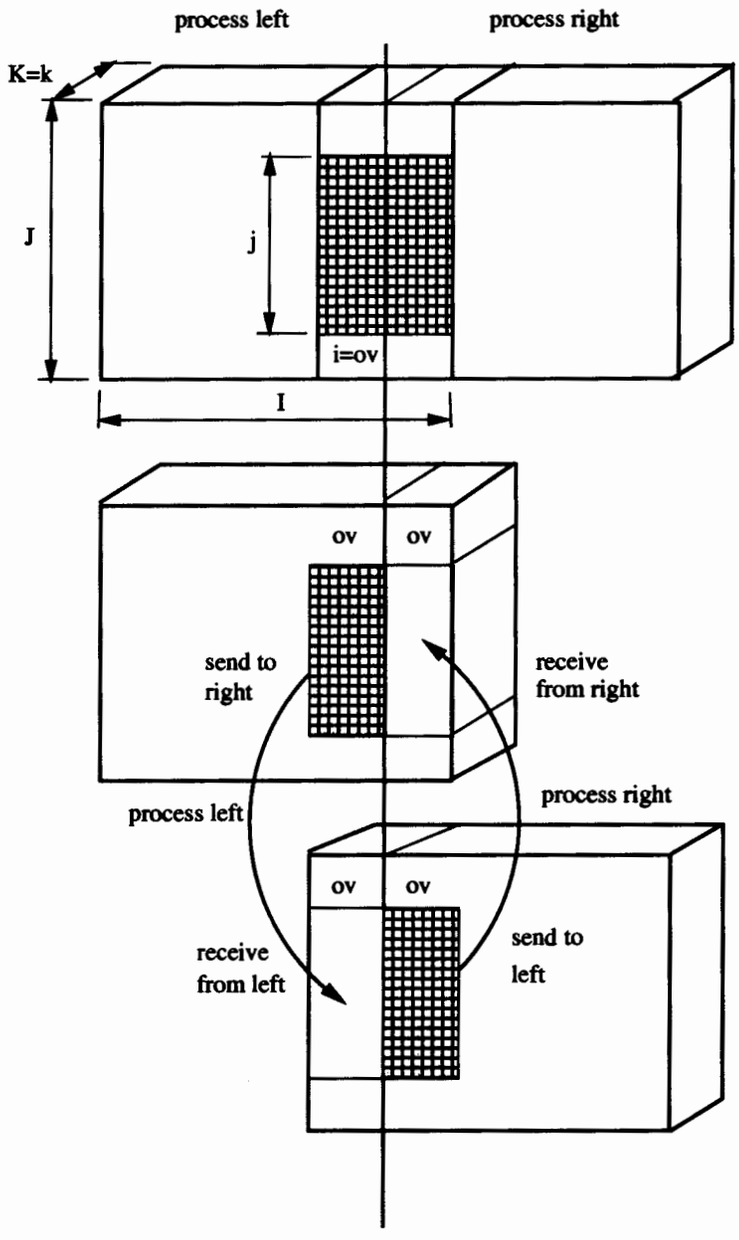

Figure 4.12: Update of *ov* halo between processes left and right.

The test is run at PC cluster for three different halos: ov=1, ov=2 and ov=4, which are often used in the algebraic operations with communications. For each size of halo, two measures of times are taken. The time of packing and unpacking, called pack, and the time of communication, called comm. These results and the overall time (i.e. the addition of these quantities) are presented for both implementations.

A comparison of all cases is given in Fig. 4.13:

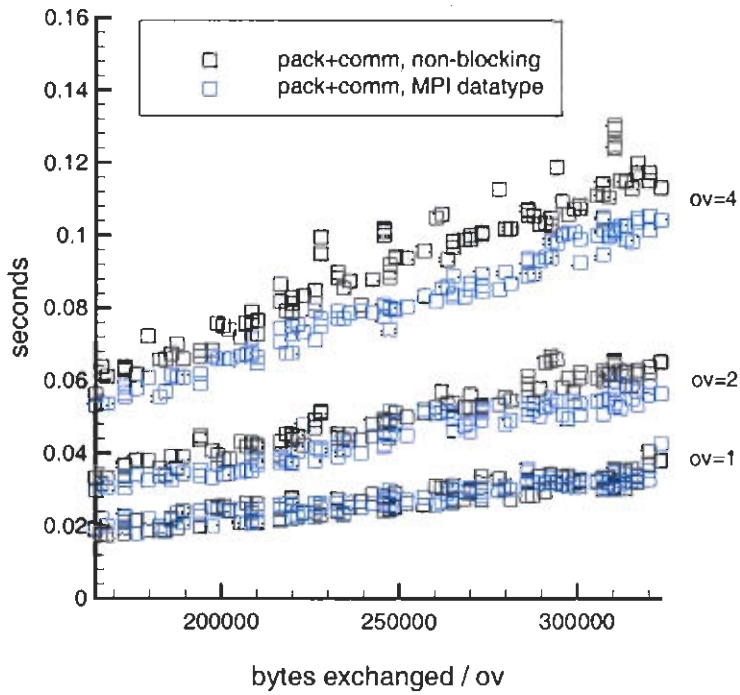

Figure 4.13: Comparison of both implementations for all halo sizes.

For each size of halos, the times of packing and unpacking of both implementations are very different. The details of each case are reported in Figs. 4.14, 4.15, 4.16, 4.17, 4.18 and 4.19.

The efficiency of the MPI_datatype implementation at PC cluster is based on the cache optimization for non-contiguous blocks when they are packed and unpacked. In addition, the packing and unpacking processes of this implementation have to be done only once because it only points to the non-contiguous data. Conversely, in the first implementation, the packing and unpacking processes are done at each exchange thus it is an explicit copy. So the performance of the overall communication is poor.

Although the results of Cray T3E has not been presented here, the difference of both implementations are not meaningful. We guess that this fact is due to the special cache built-in the cpu (see hardware issues in Appendix).

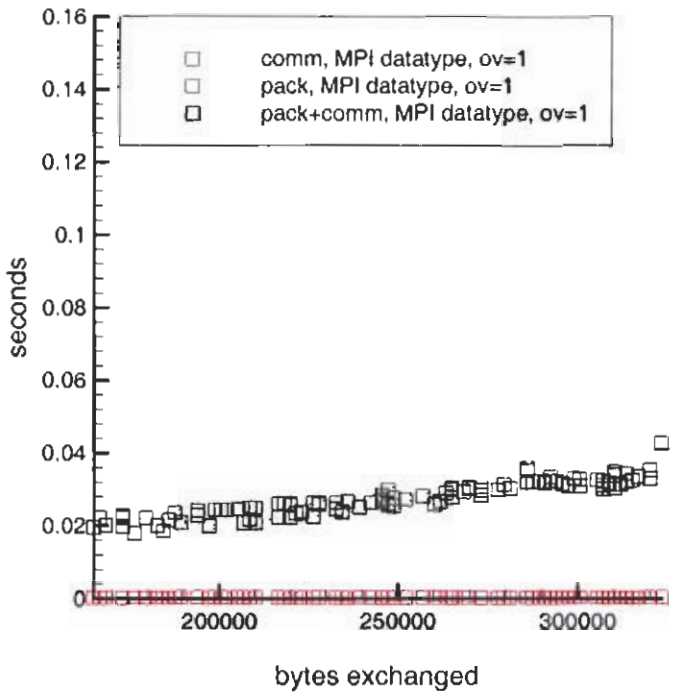

Figure 4.14: Update of $ov = 1$ with MPI_datatype.

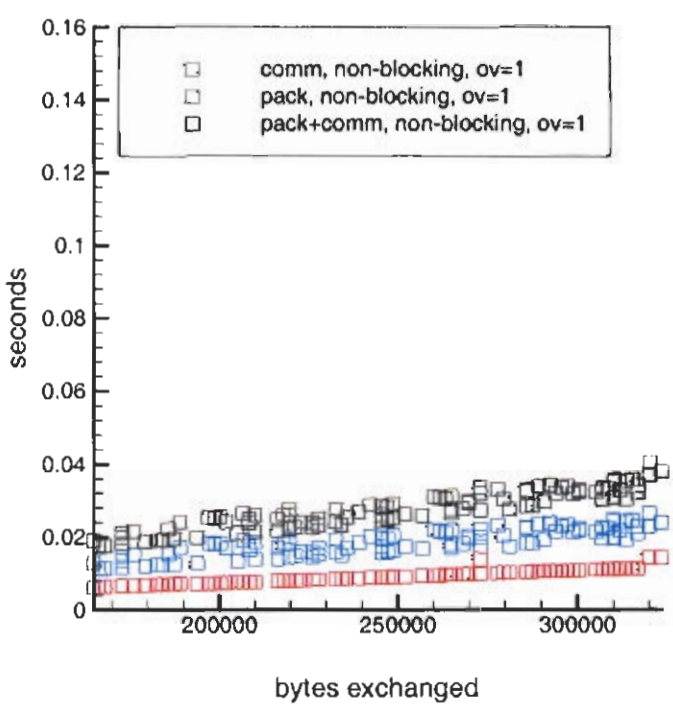

Figure 4.15: Update of $ov = 1$ with explicit copy.

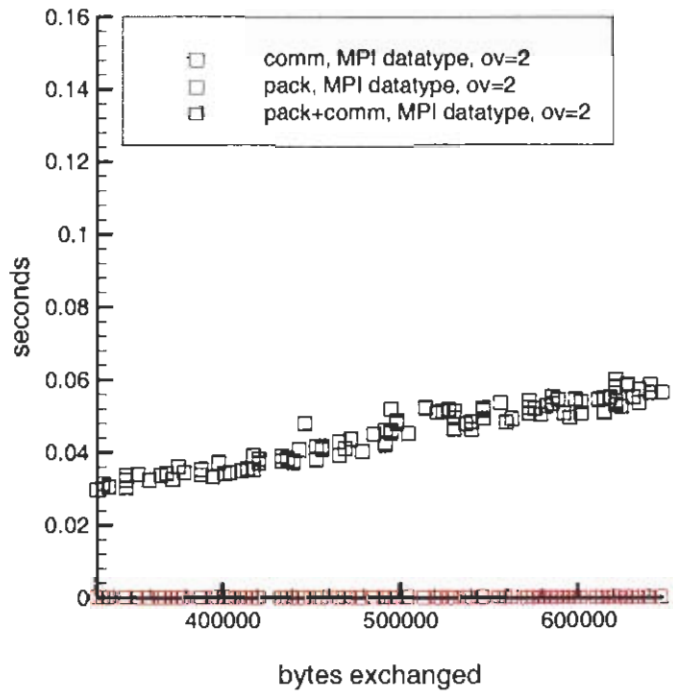

Figure 4.16: Update of $ov = 2$ with MPI_datatype.

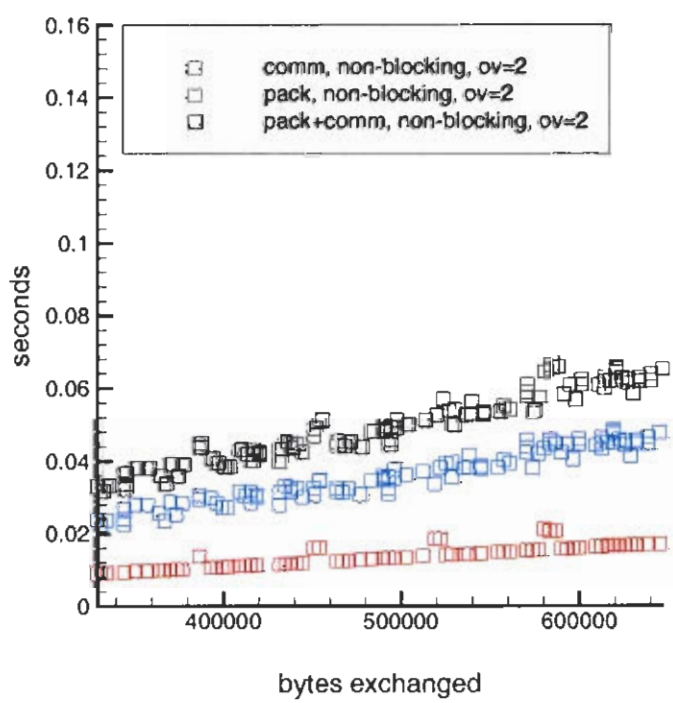

Figure 4.17: Update of $ov = 2$ with explicit copy.

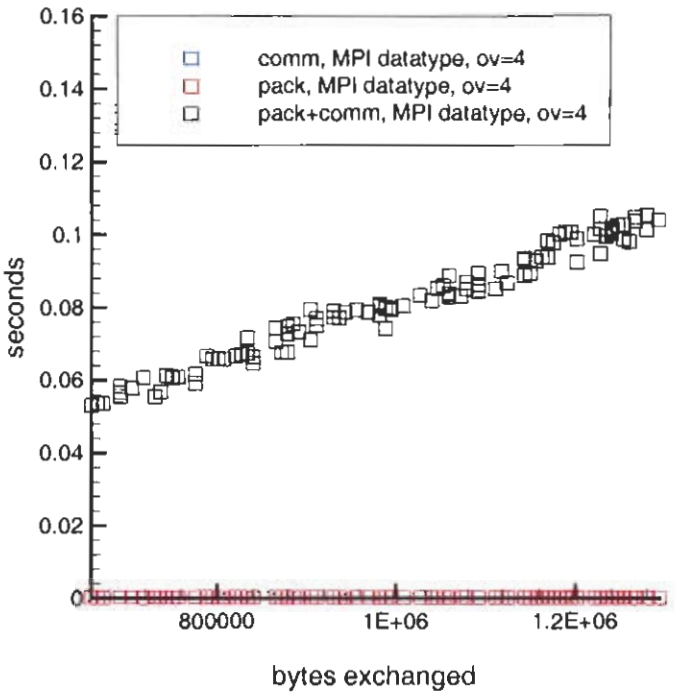

Figure 4.18: Update of $ov = 4$ with MPI_datatype.

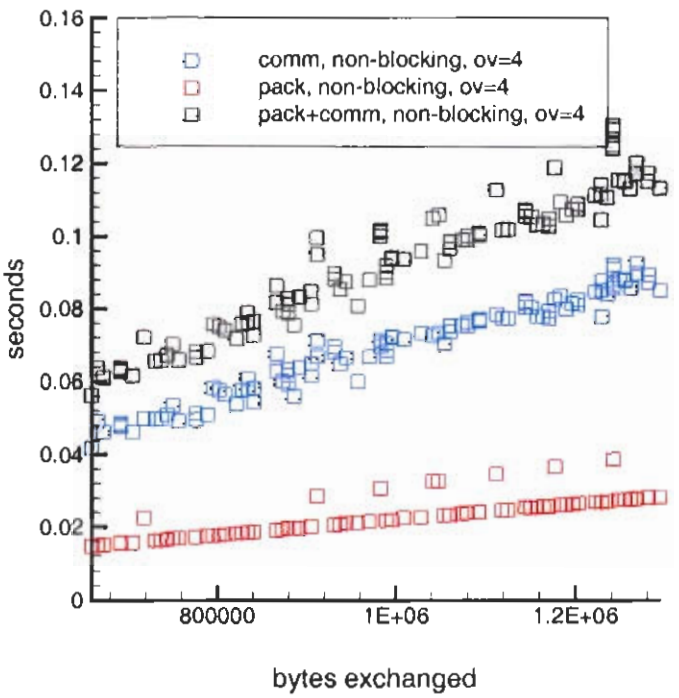

Figure 4.19: Update of $ov = 4$ with explicit copy.

4.7 Algebraic operations with vectors and matrices

Three types of algebraic operations are detailed: those without communication between processors, so the parallel efficiency is 100%, those operations that combine computation and communication, and those operations where most part of the job is the communication.

4.7.1 Addition, difference and scaling of vectors

The addition or the difference of two vectors x and y is stored in a third vector z. The algorithm (see Alg. 30) that represents any of these operations may be written as

Algorithm 30 operation_vect(x_p, y_p, z_p)

```
for (i = i1 to i = i2)
   for (j = j1 to j = j2)
      for (k = k1 to k = k2)
         z_p(i,j,k) = x_p(i,j,k) ± y_p(i,j,k)
      end for
   end for
end for
```

Another 100% parallel algebraic operation is the vector scaling (see Alg. 31). A vector x can be scaled by a real value α leading to a vector z.

Algorithm 31 Scal_vect(x_p, α, z_p)

```
for (i = i1 to i = i2)
   for (j = j1 to j = j2)
      for (k = k1 to k = k2)
         z_p(i,j,k) = α x_p(i,j,k)
      end for
   end for
end for
```

Notice that we have not considered the overlapping area in the computation of the vector z. This fact reduces the number of floating point operations, and it produces a reduction of time of the computation. Furthermore, it is possible to perform these operations reusing any vector, e.g., $x = x \pm x$, $x = x \pm y$ and $x = \alpha x$, and hence, reducing storage requirements.

4.7.2 Saxpy operation

The name of the subroutine saxpy [59] comes from the scientific literature and it represents a composition of two previous types of operations.

$$z = x + \alpha y$$

Although it can be implemented in two steps by means of the above algebraic operations, it has been packed in one step. Alg. 32 also enables the reuse of any vector.

Algorithm 32 Saxpy(x_p, α, y_p, z_p)

```
for (i = i1 to i = i2)
   for (j = j1 to j = j2)
      for (k = k1 to k = k2)
         z_p(i,j,k) = x_p(i,j,k) + α y_p(i,j,k)
      end for
   end for
end for
```

4.7.3 Inner product of vectors

The computation of the inner product of two vectors is one of the most important keys in the parallel efficiency of most solvers, because it involves a global communication between all processors. Let

$$\rho = < x, y >$$

be the inner product of two vectors, it is performed in two steps (see Alg. 33). It starts with the inner product of each pair of sub vectors x_p and y_p where $p = \{1, 2, \ldots, np\}$. The resulting set of np inner products is stored in an auxiliar variable ρ_p. Then a global sum of these values is performed and shared to all the processors by means of a global communication, so there is a fraction of time spent on the communication, and hence a lose of efficiency.

Algorithm 33 Inner_product(x_p, y_p, ρ)

evaluate *inner product of vectors* x_p , y_p
$\rho_p = 0$
for $(i = i1$ *to* $i = i2)$
  for $(j = j1$ *to* $j = j2)$
    for $(k = k1$ *to* $k = k2)$
      $\rho_p = \rho_p + x_p(i,j,k)y_p(i,j,k)$
    end for
  end for
end for
evaluate *global summation of* ρ_p
  global_sum (ρ_p, ρ)

Notice that Alg. 33 enables to compute the 2-norm of a vector x (see Alg. 34).

The inner product operation contains a global communication that broadcast each sub inner product to the rest of processors. After that, a sum of all values is performed in each processor. The implementation of this broadcast plus the summation relays on the MPI library and it is performed in the MPI_Allreduce subroutine.

Algorithm 34 Norm_vect(x_p, ρ)

compute *the inner product of* x *with itself.*
  inner_product(x_p, x_p, ρ_0)
evaluate *the* 2*-norm of* x
  $\rho = \sqrt{\rho_0}$

Since the number of messages and data does not depend on the partitioning configuration, differences of the speed-ups are due to the differences in computation. Therefore, only the cache missing effects may arise for some configurations for a given case with large size. For instance, the size of vectors are $20 \times 20 \times 20$, $40 \times 40 \times 40$, $60 \times 60 \times 60$, $80 \times 80 \times 80$ and $100 \times 100 \times 100$.

The test is executed on both facilities, PC cluster and Cray T3E within the range of 1 to 12 processors and for different partitioning directions (see Fig. 4.20).

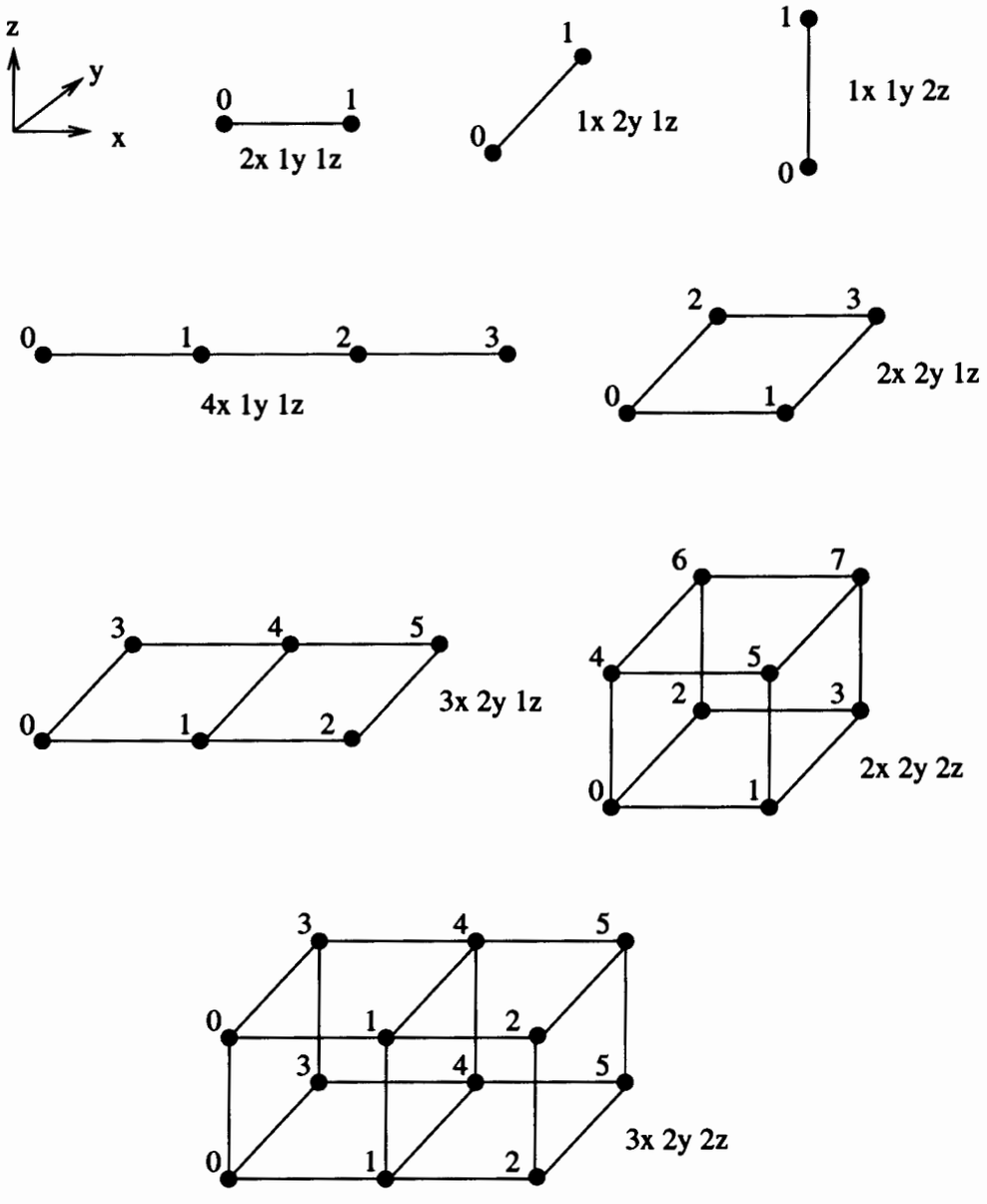

Figure 4.20: partitioning directions that yield different topologies of processors: (line, plane and hexahedron) with 2, 4, 6, 8 and 12 processors.

The results are represented in terms of the speed-up in Figs. 4.21,4.22 and tables 4.3,4.4 for PC cluster and the Cray T3E respectively.

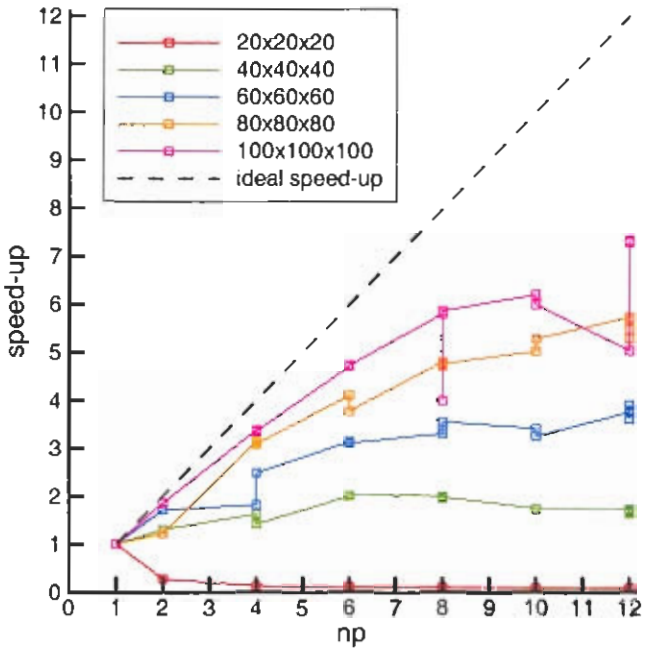

Figure 4.21: Speed-up for the inner product of 3D vectors in PC cluster.

np	partition	$20 \times 20 \times 20$	$40 \times 40 \times 40$	$60 \times 60 \times 60$	$80 \times 80 \times 80$	$100 \times 100 \times 100$
1	1x1y1z	1.00	1.00	1.00	1.00	1.00
2	1x1y2z	0.27	1.30	1.71	1.22	1.85
4	1x1y4z	0.13	1.63	1.82	3.15	3.38
4	1x2y2z	0.13	1.42	2.49	3.09	3.35
6	1x1y6z	0.12	2.00	3.14	4.10	4.71
6	1x2y3z	0.12	2.02	3.11	3.77	4.73
8	1x1y8z	0.11	2.00	3.30	4.80	5.80
8	1x2y4z	0.11	1.95	3.38	4.72	3.99
8	2x2y2z	0.11	1.98	3.55	4.76	5.86
10	1x1y10z	0.09	1.74	3.41	5.02	6.20
10	1x2y5z	0.09	1.73	3.25	5.28	5.99
12	1x1y12z	0.09	1.73	3.75	5.73	5.03
12	1x2y6z	0.09	1.71	3.90	5.48	7.29
12	1x3y4z	0.09	1.68	3.79	5.68	7.30
12	2x2y3z	0.09	1.64	3.60	5.29	7.33

Table 4.3: Speed-np of the inner product of 3D vectors in the PC cluster.

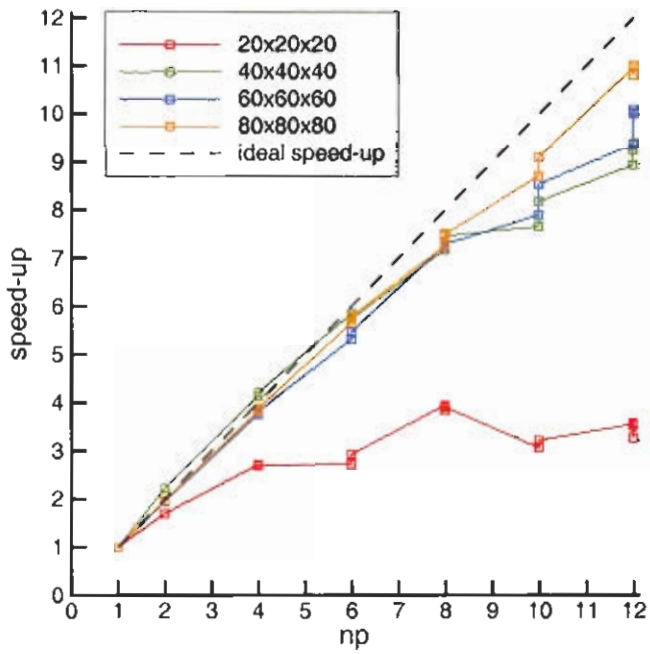

Figure 4.22: Speed-up for the inner product of 3D vectors in the Cray T3E.

np	partition	$20 \times 20 \times 20$	$40 \times 40 \times 40$	$60 \times 60 \times 60$	$80 \times 80 \times 80$
1	1x1y1z	1.00	1.00	1.00	1.00
2	1x1y2z	1.70	2.23	1.96	1.98
4	1x1y4z	2.72	4.14	3.77	3.84
4	1x2y2z	2.70	4.21	3.82	3.88
6	1x1y6z	2.74	5.85	5.32	5.66
6	1x2y3z	2.94	5.74	5.47	5.79
8	1x1y8z	3.96	7.18	7.28	7.26
8	1x2y4z	3.85	7.51	7.24	7.46
8	2x2y2z	3.92	7.47	7.29	7.49
10	1x1y10z	3.08	7.66	7.90	8.71
10	1x2y5z	3.23	8.18	8.54	9.10
12	1x2y12z	3.57	8.94	9.36	10.96
12	1x2y6z	3.53	9.38	10.01	10.80
12	1x3y4z	3.60	9.36	10.03	11.01
12	2x2y3z	3.28	9.25	10.09	10.96

Table 4.4: Speed-up of the inner product of 3D vectors in the Cray T3E.

As stated above, the effect of the partitioning configuration has a slight influence on the inner product speed-up.

4.7.4 Matrix-vector product

This operation appears in almost all the solver algorithms. Due to the intensive computational work, it is even used as a work counter in solvers instead of the number of iterations which involve additional operations but with cheaper work. Moreover in a *np*-parallel machine, the matrix-vector product becomes less effective due to the extra work of communication among the the processor p and the neighbour processors ngb_p. The information of neighbour processors ngb_p is previously "passed" and stored in the mentioned overlapping areas of processor p and then the operation is fully performed in processor p as

$$y_p = A_p x_p + A_{ngb_p} x_{ngb_p}$$

Indeed the size of the overlapping area plays an important role in the time spent in the communication processes. Furthermore, the type of formulation defines the sparsity of the matrix or in other words, the dependencies between the nodes stored in the processor p and those nodes stored in the neighbour processors ngb_p. For a matrix-vector product in a 5,7,9 or 19-point formulation, it is only necessary an overlapping of one ($ov = 1$) in the orthogonal directions of the domain.

Higher order schemes lead to formulations where the overlapping must be higher ($ov \geq 2$), and thus, the amount of data passed among processes increases the time of communication. A generalization of the product of a 7-point formulation matrix with a 3D vector named $mat - vect$ is written for a given overlapping ov in Alg. 35.

Algorithm 35 Mat_vect (A_p, x_p, y_p)

update *overlapped information of* x_p
  update(ov, x_p)

evaluate *the matrix-vector product*
for $(i = i1 - ov + 1$ *to* $i = i2 + ov - 1)$
  for $(j = j1 - ov + 1$ *to* $j = j2 + ov - 1)$
    for $(k = k1 - ov + 1$ *to* $k = k2 + ov - 1)$

$$\begin{aligned} y_p(i,j,k) = {} & A_p^p(i,j,k)x_p(i,j,k) + \\ & A_p^w(i,j,k)x_p(i-1,j,k) + A_p^e(i,j,k)x_p(i+1,j,k) + \\ & A_p^s(i,j,k)x_p(i,j-1,k) + A_p^n(i,j,k)x_p(i,j+1,k) + \\ & A_p^b(i,j,k)x_p(i,j,k-1) + A_p^t(i,j,k)x_p(i,j,k+1) \end{aligned}$$

    end for
  end for
end for

At this point, the above algebraic operations enable us to build more complicate operations. For example, let us show how the residual needed in the stopping criteria for a 7-point formulation matrix is built in Alg. 36.

Algorithm 36 Residual(A_p, x_p, b_p, r_p)

evaluate *the operation* $r_p = A_p x_p$
  mat_vect (A_p, x_p, r_p)
evaluate *the operation* $r_p = b_p - r_p$
  diff_vect (b_p, r_p, r_p)

4.7.5 Minimum matrix-vector product size per processor

The following test models the communication and computation timings of the matrix-vector product. It is designed to show the need of an overlapping of the communication and the serial local computation within the operation. The matrix-vector operation for both two dimensional $I \times I$ and three dimensional $I \times I \times I$ CFD problems has been taken (i.e. 5-point and 7-point formulations respectively). The measures of both times, the serial local computation of the matrix-vector product and the exchange of halos of size ov=1, are compared for a wide range of size problems.

The measures of the serial local computation timings are carried out in one processor for different size problems. For instance, the two dimensional problem version is described in Alg. 37.

Algorithm 37 Serial local computation of y=Ax

```
for (sample = 1 to sample = 1000)
  I[sample]=J[sample]=random(1,1000)
  MPI_Barrier()
  t0=MPI_Wtime()
  for (j = 1 to j = J[sample])
    for (i = 1 to i = I[sample])
      y(i,j) = A^p(i,j)x(i,j)
             + A^w(i,j)x(i-1,j)
             + A^e(i,j)x(i+1,j)
             + A^s(i,j)x(i,j-1)
             + A^n(i,j)x(i,j+1)
    end for
  end for
  t1=MPI_Wtime()
  tcomp[sample]=t1-t0
end for
```

Meanwhile, the exchange of halos is simulated in only two processors for the same range of size problems and on the assumption that the exchange of halos is in all directions. The measure of the times while the exchange of the halos of x for the two dimensional problem is described in Alg. 38.

Algorithm 38 Exchange of halos ov=1 of x

```
for (sample = 1 to sample = 1000)
   I[sample]=random(1,1000)
   MPI_Barrier()
   exchange sides
      t0=MPI_Wtime()
      exchange(I[sample]*ov)
      t1=MPI_Wtime()
      tcomm[sample]=4*(t1-t0)
end for
```

Here, in order to reduce as much as possible the time of communication, the exchange subroutine transfers the packed data from one processor to another in the non-blocking mode.

The test is run in both machines PC cluster and the Cray T3E. The time results versus the size problem are presented in Figs. 4.23, 4.24, 4.25 and 4.26.

If the communication process and the serial local computation process are performed consecutively (i.e. in non-overlapped fashion), the intersection points for each case define the minimum estimate sizes of the local problem which a processor could do with a parallel efficiency of 50%.

Let $tcomp(1)$ and $t_{comp(p)} + t_{comm(p)}$ be the sequential and parallel (for np processors) timings of the overall operation, the efficiency is approximately expressed as

$$E(np) = \frac{t_{comp(1)}}{np(t_{comp(np)} + t_{comm(np)})} \approx \frac{npt_{comp(np)}}{np(t_{comp(np)} + t_{comm(np)})}$$

In the intersection point $t_{comm(np)} = t_{comp(np)}$. Therefore

$$E(np) = \frac{np\ t_{comp(np)}}{np(t_{comp(np)} + t_{comp(np)})} = \frac{1}{2}$$

These points are given in table 4.5:

Problem	PC cluster	Cray T3E
Two dimensional	$40.000 = (200 \times 200)$	$2.500 = (50 \times 50)$
Three dimensional	$64.000 = (40 \times 40 \times 40)$	$3.375 = (15 \times 15 \times 15)$

Table 4.5: Minimum estimate size per processor for a parallel efficiency of 50% for the non-overlapped computation and communication.

Analogously to the previous section, the measure of the speed-up of the matrix-vector product for a given size problem is obtained by timing the algebraic operation at different number of processors. The test is executed within the range of 1 to 12 processors in both facilities PC cluster and Cray T3E. Indeed, partitioning in two or three directions affects

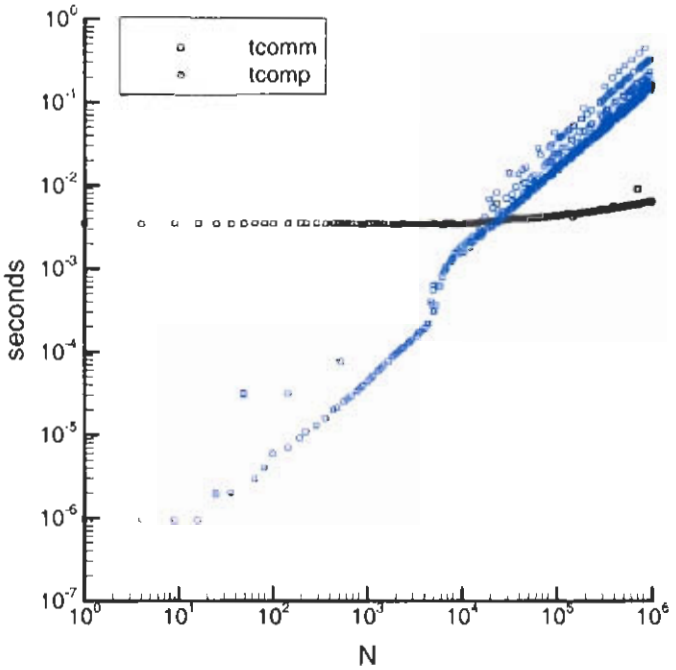

Figure 4.23: Timings of computation and communication of a two-dimensional matrix-vector product in PC cluster.

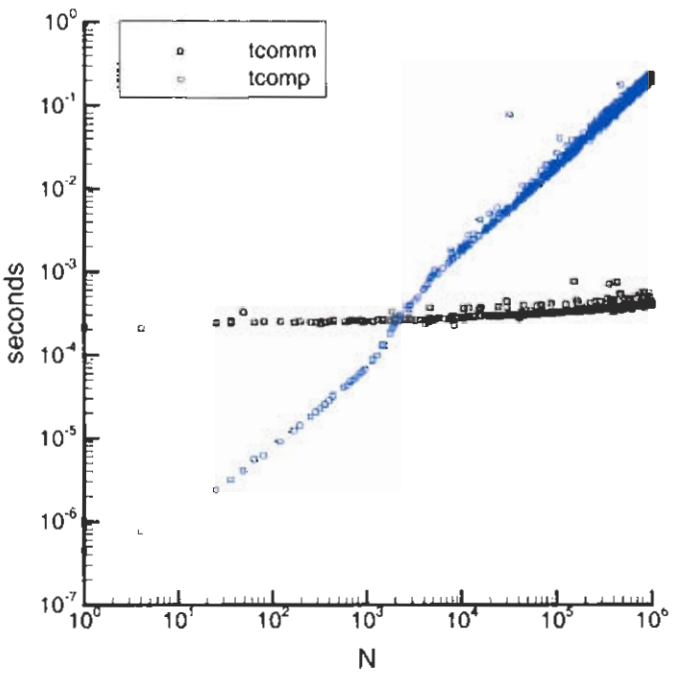

Figure 4.24: Timings of computation and communication of a two-dimensional matrix-vector product in the Cray T3E.

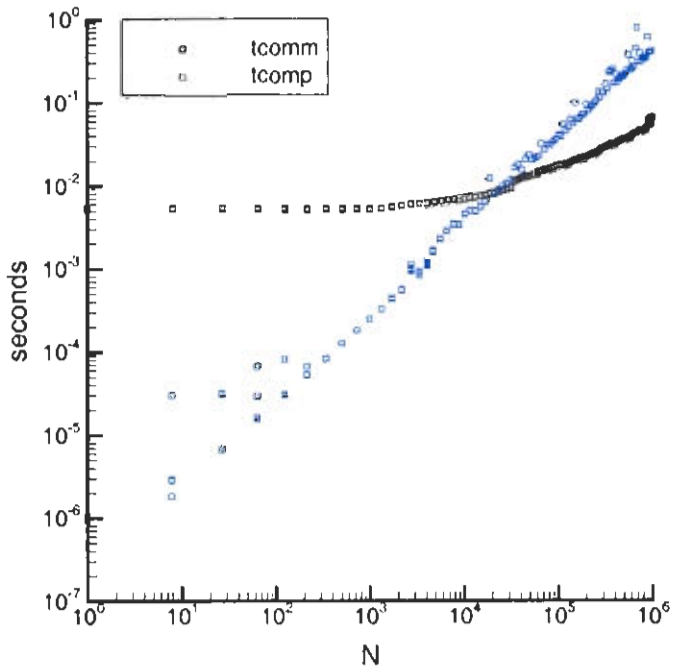

Figure 4.25: Timings of computation and communication of a three-dimensional matrix-vector product in PC cluster.

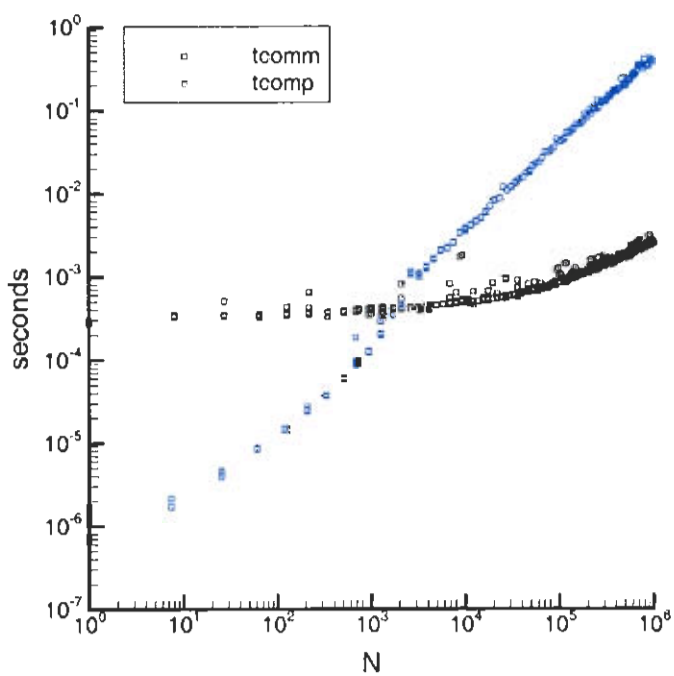

Figure 4.26: Timings of computation and communication of a three-dimensional matrix-vector product in the Cray T3E.

the efficiency due to (1) the different ratios of computation versus the exchange of data and (2) the cache effects for large amounts of data distributed in few processors.

These effects are analyzed for a set of cases ($20 \times 20 \times 20$), ($40 \times 40 \times 40$), ($60 \times 60 \times 60$), ($80 \times 80 \times 80$) and ($100 \times 100 \times 100$) (the last one only for PC cluster).

As mentioned above, the size of all of these cases is over the minimum estimated for the Cray T3E ($15 \times 15 \times 15$) so one may expect efficiencies higher than 50%. For PC cluster, these efficiencies are expected under the ($40 \times 40 \times 40$) size.

The experiment is repeated several times for each number of processors and for each partitioning configuration. The speed-up for each case is evaluated with the averaged timings. Full results (all partitioning configurations) are represented in Figs. 4.27, 4.28 and tables 4.6, 4.7 for the PC cluster and the Cray T3E respectively.

The successive experiments have been computed with a generalized algorithm 39 where the number of processors, partitioning configurations, and problem sizes are expressed in a set of nested loops.

Algorithm 39 Performance of operations

for ($np = 1$ *to* $np = 12$)
  for (*partitions* $= p_x,\ p_y,\ p_z = 1$ *to 12, such that* $p_x p_y p_z = np$)
    for (*size* $= 20$ *to size* $= 100$, *size=size+20*)
      for (*sample* $= 1$ *to sample* $= 20$)

        MPI_Barrier()
        t0=MPI_Wtime()
        evaluate *algebraic operation or* **solve** *a problem*
        t1=MPI_Wtime()
        time=time + t1-t0

      end for
      $t_{comp}[operation][size][partition][np] = \dfrac{time}{20}$
    end for
  end for
end for

Where the algebraic operations are the matrix vector product and inner product, and the solvers are Jacobi, Gauss-Seidel, MSIP, Conjugate Gradient, BiCGSTAB and GMRESR.

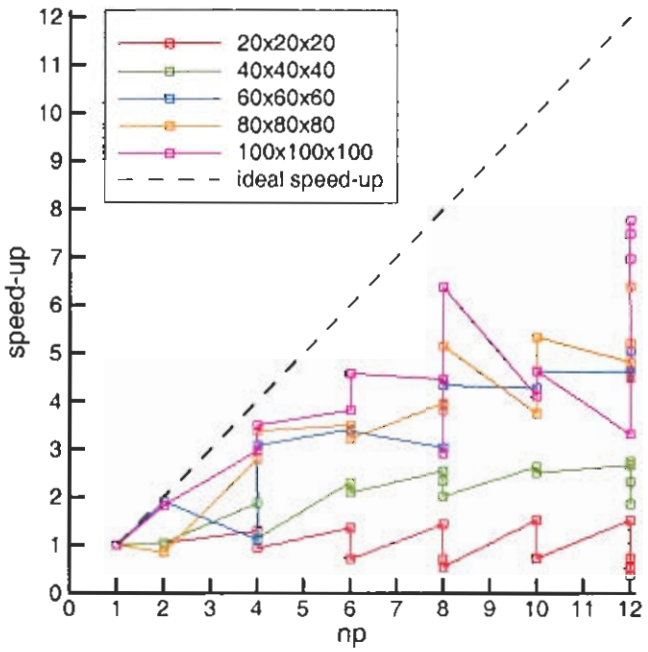

Figure 4.27: Speed-up for the matrix-vector product in PC cluster.

np	partition	$20 \times 20 \times 20$	$40 \times 40 \times 40$	$60 \times 60 \times 60$	$80 \times 80 \times 80$	$100 \times 100 \times 100$
1	1x1y1z	1.00	1.00	1.00	1.00	1.00
2	1x1y2z	1.04	1.04	1.90	0.84	1.82
4	1x1y4z	1.29	1.87	1.10	2.79	2.97
4	1x2y2z	0.93	1.13	3.08	3.38	3.50
6	1x1y6z	1.36	2.30	3.42	3.50	3.81
6	1x2y3z	0.71	2.09	3.37	3.22	4.57
8	1x1y8z	1.43	2.55	3.03	3.95	4.45
8	1x2y4z	0.71	2.33	3.91	3.81	2.91
8	2x2y2z	0.53	2.01	4.33	5.14	6.36
10	1x1y10z	1.52	2.63	4.26	3.74	4.10
10	1x2y5z	0.72	2.50	4.62	5.32	4.62
12	1x2y12z	1.51	2.67	4.59	4.80	3.31
12	1x2y6z	0.72	2.74	5.04	5.19	6.95
12	1x3y4z	0.52	2.31	4.53	6.37	7.48
12	2x2y3z	0.37	1.85	4.50	4.47	7.75

Table 4.6: Speed-up of matrix-vector product in PC cluster.

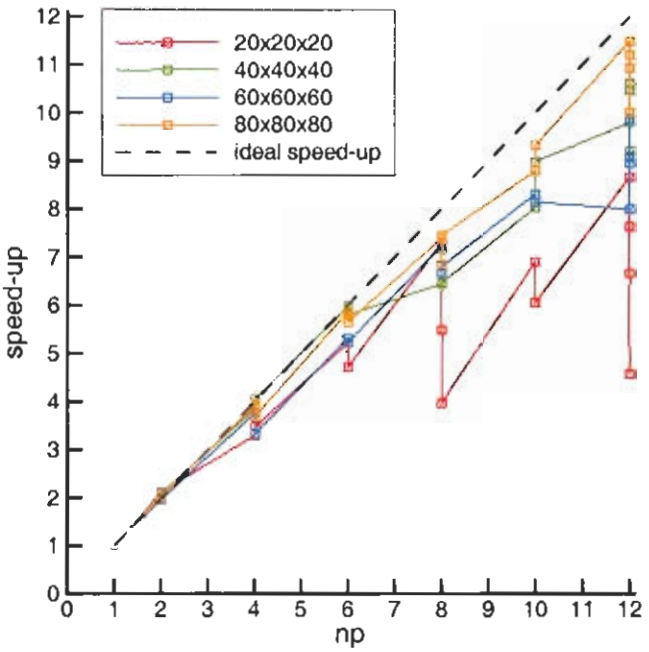

Figure 4.28: Speed-up for the matrix-vector product in the Cray T3E.

np	partition	$20 \times 20 \times 20$	$40 \times 40 \times 40$	$60 \times 60 \times 60$	$80 \times 80 \times 80$
1	1x1y1z	1.00	1.00	1.00	1.00
2	1x1y2z	2.12	2.09	1.97	2.02
4	1x1y4z	3.29	3.84	3.74	3.92
4	1x2y2z	3.46	4.03	3.31	3.67
6	1x1y6z	5.20	5.96	5.30	5.85
6	1x2y3z	4.70	5.78	5.21	5.63
8	1x1y8z	7.29	6.42	7.21	7.38
8	1x2y4z	5.47	7.15	6.66	6.83
8	2x2y2z	3.96	6.44	6.79	7.44
10	1x1y10z	6.89	8.01	8.29	8.78
10	1x2y5z	6.04	8.96	8.12	9.31
12	1x2y12z	8.65	9.77	7.98	11.47
12	1x2y6z	7.62	10.59	8.93	11.18
12	1x3y4z	6.65	10.44	9.84	9.99
12	2x2y3z	4.56	9.18	9.06	10.91

Table 4.7: Speed-up of matrix by vector in the Cray T3E.

Seeing these figures it is stated that for a given size of the problem and number of processors, the partitioning configuration is definitively a critical factor on the speed-up. The reason is that while the time of computation remains constant independently of the partitioning configuration, the time of communication varies strongly. Then the ratio between the times of computation and communication varies with the partitioning configuration. This ratio in PC cluster is greater than the obtained in the Cray T3E because the time of communication of any packet of data is longer in PC cluster while the time of computation is quite similar in both facilities.

4.8 Parallel performance of solvers

The parallel implementation of these solvers has been tested and the performance, in terms of the speed-up, is measured for a three dimensional Laplace problem (e.g. the 3D heat conduction problem) with full Dirichlet boundary conditions. In this test, the stopping criterion of $\epsilon = 10^{-6}$ has been chosen and analogously to the parallel performance of the algebraic operations, the different partitioning configurations have been tested in different sizes of problems: $20 \times 20 \times 20$, $40 \times 40 \times 40$, $60 \times 60 \times 60$, $80 \times 80 \times 80$, $100 \times 100 \times 100$. For each solver, size of problem number of processors and partitioning configurations, the test is repeated several times and the timings are averaged.

The test is executed at both computers PC cluster and the Cray T3E. The results are summarized by means of the speed-ups. Figures and tables for each solver are reported below.

- For Jacobi solver see Figs. 4.29, 4.30 and tables 4.8,4.9.
- For Gauss-Seidel solver see Figs. 4.31, 4.32 and tables 4.10,4.11.
- For MSIP solver with $\alpha = 0.5$ see Figs. 4.33, 4.34 and tables 4.12,4.13.
- For preconditioned BiCGSTAB solver see Figs. 4.35, 4.36 and tables 4.14,4.15.
- For preconditioned GMRESR(10) solver see Figs. 4.37,4.38 and tables 4.16,4.17.

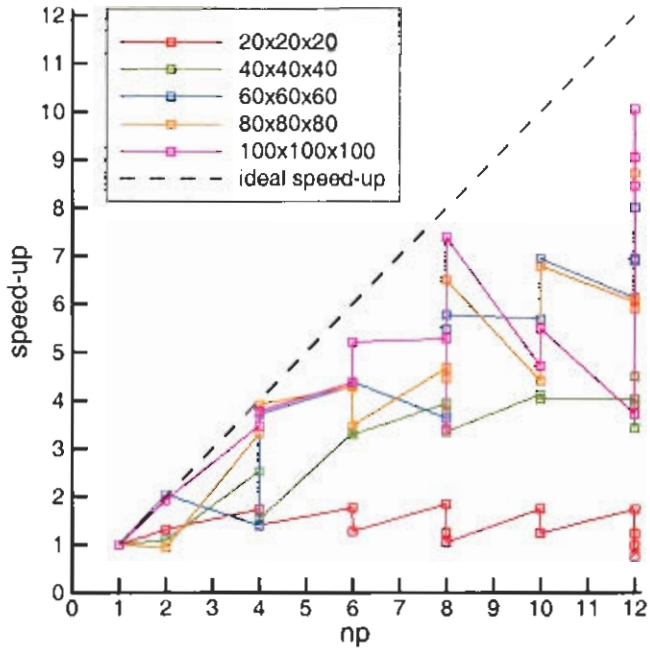

Figure 4.29: Speed-up for the Jacobi in PC cluster.

np	partition	$20 \times 20 \times 20$	$40 \times 40 \times 40$	$60 \times 60 \times 60$	$80 \times 80 \times 80$	$100 \times 100 \times 100$
1	1x1y1z	1.00	1.00	1.00	1.00	1.00
2	1x1y2z	1.32	1.09	2.04	0.94	1.93
4	1x1y4z	1.74	2.52	1.39	3.31	3.47
4	1x2y2z	1.41	1.52	3.71	3.92	3.78
6	1x1y6z	1.77	3.33	4.30	4.28	4.37
6	1x2y3z	1.28	3.28	4.38	3.47	5.21
8	1x1y8z	1.85	3.94	3.63	4.68	5.28
8	1x2y4z	1.26	3.73	5.47	4.46	3.40
8	2x2y2z	1.05	3.34	5.77	6.50	7.39
10	1x1y10z	1.75	4.12	5.68	4.41	4.72
10	1x2y5z	1.24	4.02	6.94	6.79	5.49
12	1x2y12z	1.74	4.03	6.12	6.04	3.72
12	1x2y6z	1.23	4.49	8.01	6.11	8.46
12	1x3y4z	0.99	3.91	6.94	8.72	9.05
12	2x2y3z	0.77	3.42	6.89	5.90	10.04

Table 4.8: Speed-up of Jacobi solver in PC cluster.

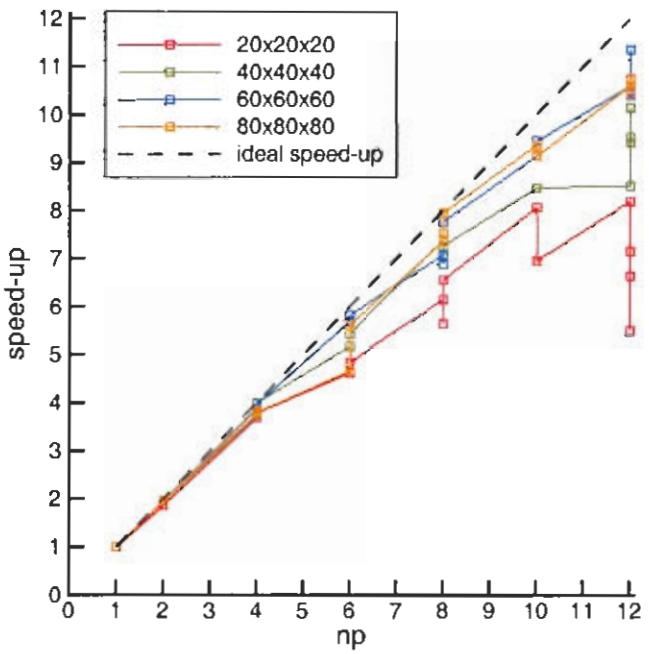

Figure 4.30: Speed-up for the Jacobi in the Cray T3E.

np	partition	$20 \times 20 \times 20$	$40 \times 40 \times 40$	$60 \times 60 \times 60$	$80 \times 80 \times 80$
1	1x1y1z	1.00	1.00	1.00	1.00
2	1x1y2z	1.86	1.95	1.96	1.95
4	1x1y4z	3.69	3.75	3.88	3.85
4	1x2y2z	3.78	4.00	3.97	3.80
6	1x1y6z	4.62	5.17	5.69	4.67
6	1x2y3z	4.83	5.43	5.82	5.61
8	1x1y8z	6.12	7.37	7.06	7.33
8	1x2y4z	5.62	7.01	6.87	7.51
8	2x2y2z	6.53	7.25	7.76	7.94
10	1x1y10z	8.05	8.47	9.13	9.35
10	1x2y5z	6.93	8.47	9.44	9.12
12	1x2y12z	8.18	8.51	10.59	10.59
12	1x2y6z	7.13	9.41	10.48	10.43
12	1x3y4z	6.62	10.14	11.35	10.71
12	2x2y3z	5.48	9.52	10.40	10.74

Table 4.9: Speed-up of Jacobi solver at Cray T3E.

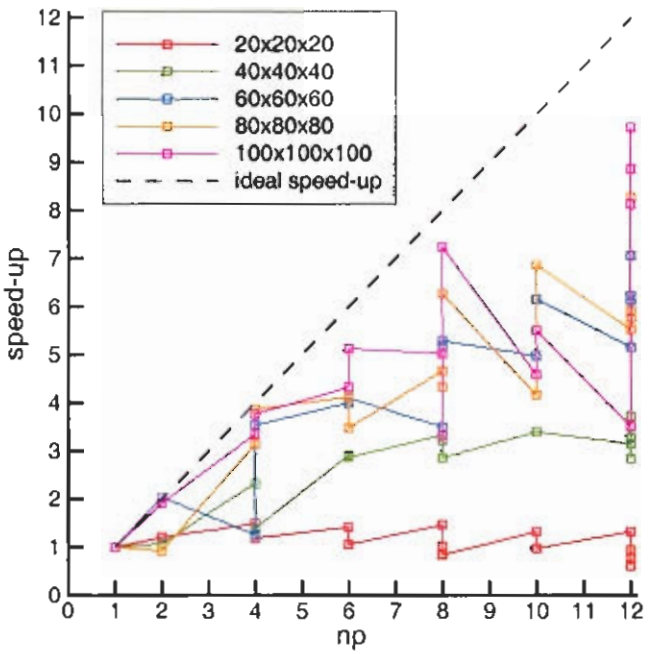

Figure 4.31: Speed-up for the Gauss Seidel in PC cluster.

np	partition	20 × 20 × 20	40 × 40 × 40	60 × 60 × 60	80 × 80 × 80	100 × 100 × 100
1	1x1y1z	1.00	1.00	1.00	1.00	1.00
2	1x1y2z	1.21	1.07	2.03	0.92	1.91
4	1x1y4z	1.49	2.32	1.25	3.14	3.35
4	1x2y2z	1.19	1.38	3.54	3.88	3.77
6	1x1y6z	1.42	2.91	4.00	4.12	4.33
6	1x2y3z	1.06	2.87	4.10	3.48	5.13
8	1x1y8z	1.47	3.33	3.50	4.67	5.03
8	1x2y4z	1.02	3.22	5.18	4.34	3.32
8	2x2y2z	0.84	2.86	5.29	6.27	7.24
10	1x1y10z	1.33	3.40	4.98	4.17	4.60
10	1x2y5z	0.98	3.40	6.15	6.87	5.51
12	1x2y12z	1.33	3.15	5.16	5.53	3.52
12	1x2y6z	0.95	3.73	7.06	5.91	8.13
12	1x3y4z	0.78	3.28	6.22	8.28	8.86
12	2x2y3z	0.62	2.85	6.13	5.77	9.72

Table 4.10: Speed-up of Gauss-Seidel solver in PC cluster.

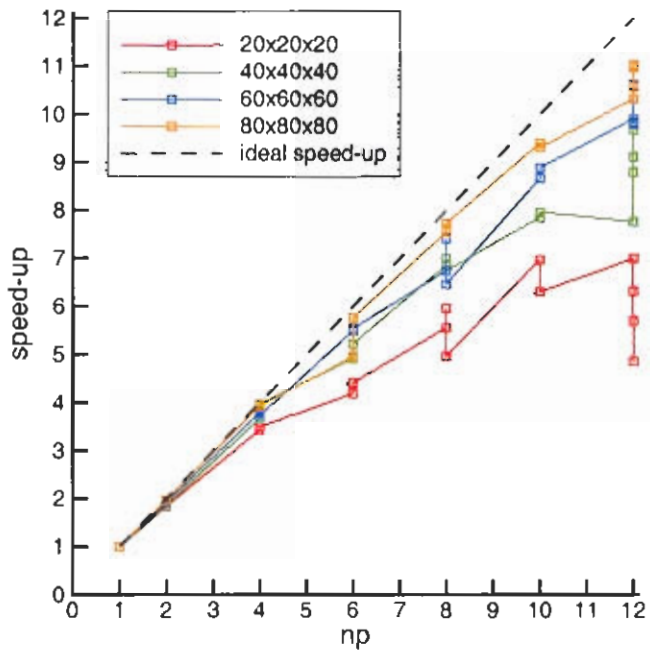

Figure 4.32: Speed-up for the Gauss Seidel in the Cray T3E.

np	partition	$20 \times 20 \times 20$	$40 \times 40 \times 40$	$60 \times 60 \times 60$	$80 \times 80 \times 80$
1	1x1y1z	1.00	1.00	1.00	1.00
2	1x1y2z	1.84	1.86	1.93	1.98
4	1x1y4z	3.43	3.67	3.80	3.94
4	1x2y2z	3.50	3.97	3.73	3.93
6	1x1y6z	4.20	4.93	5.50	4.97
6	1x2y3z	4.42	5.23	5.55	5.76
8	1x1y8z	5.56	6.87	6.75	7.55
8	1x2y4z	5.96	7.01	7.42	7.61
8	2x2y2z	4.97	6.75	6.47	7.73
10	1x1y10z	6.97	7.86	8.67	9.40
10	1x2y5z	6.30	7.97	8.89	9.31
12	1x2y12z	7.00	7.77	9.91	10.32
12	1x2y6z	6.32	9.12	9.84	10.57
12	1x3y4z	5.70	9.68	10.62	10.98
12	2x2y3z	4.86	8.80	9.80	11.03

Table 4.11: Speed-up of Gauss-Seidel solver at Cray T3E.

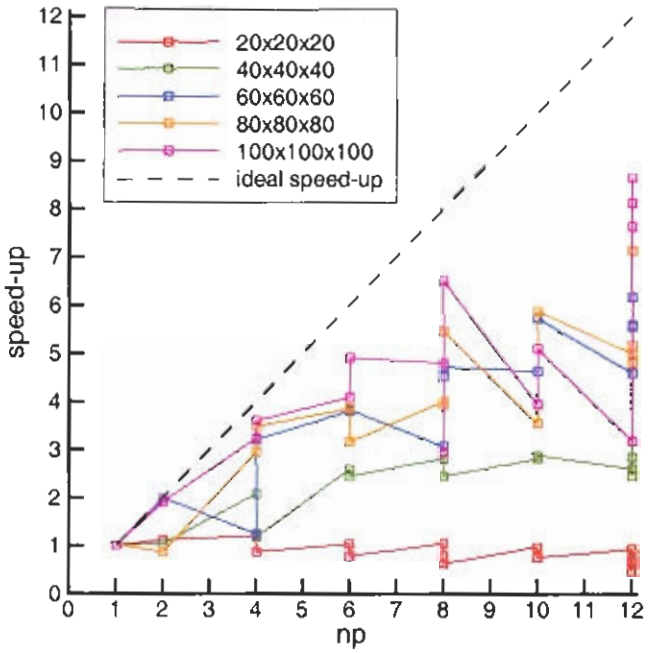

Figure 4.33: Speed-up for the MSIP in PC cluster.

np	partition	$20 \times 20 \times 20$	$40 \times 40 \times 40$	$60 \times 60 \times 60$	$80 \times 80 \times 80$	$100 \times 100 \times 100$
1	1x1y1z	1.00	1.00	1.00	1.00	1.00
2	1x1y2z	1.10	1.01	1.96	0.85	1.89
4	1x1y4z	1.19	2.05	1.22	2.93	3.22
4	1x2y2z	0.85	1.15	3.18	3.46	3.59
6	1x1y6z	1.02	2.56	3.78	3.84	4.07
6	1x2y3z	0.77	2.42	3.80	3.14	4.89
8	1x1y8z	1.03	2.79	3.04	3.99	4.78
8	1x2y4z	0.77	2.77	4.49	3.91	2.92
8	2x2y2z	0.60	2.41	4.69	5.43	6.47
10	1x1y10z	0.95	2.78	4.59	3.52	3.92
10	1x2y5z	0.73	2.85	5.69	5.85	5.07
12	1x2y12z	0.91	2.57	4.56	4.97	3.16
12	1x2y6z	0.69	3.15	6.13	5.14	7.60
12	1x3y4z	0.61	2.81	5.52	7.10	8.09
12	2x2y3z	0.45	2.42	5.56	4.78	8.62

Table 4.12: Speed-up of MSIP solver with $\alpha = 0.5$ in PC cluster.

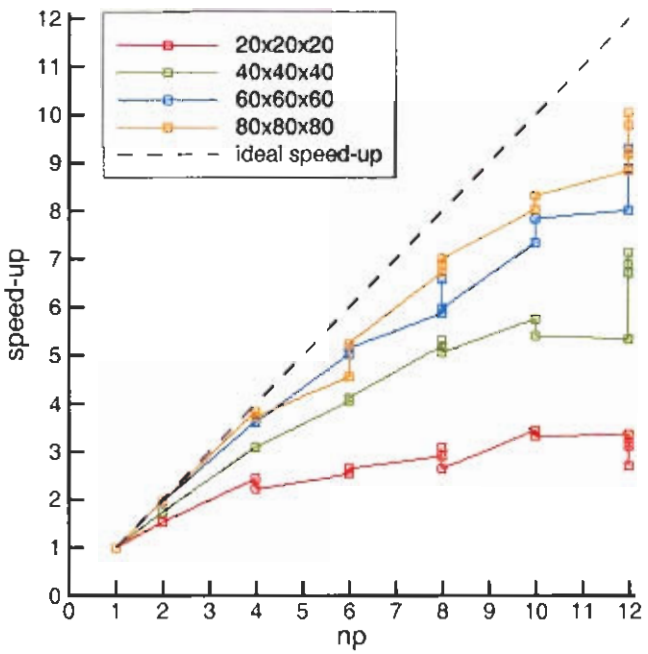

Figure 4.34: Speed-up for the MSIP in the Cray T3E.

np	partition	$20 \times 20 \times 20$	$40 \times 40 \times 40$	$60 \times 60 \times 60$	$80 \times 80 \times 80$
1	1x1y1z	1.00	1.00	1.00	1.00
2	1x1y2z	1.53	1.74	1.97	1.98
4	1x1y4z	2.44	3.10	3.63	3.84
4	**1x2y2z**	2.22	3.09	3.61	3.73
6	1x1y6z	2.53	4.04	5.02	4.55
6	1x2y3z	2.64	4.11	5.14	5.24
8	1x1y8z	2.90	5.19	5.86	6.72
8	1x2y4z	3.08	5.31	6.58	6.88
8	2x2y2z	2.63	**5.05**	5.97	7.01
10	1x1y10z	3.44	**5.75**	7.33	8.03
10	1x2y5z	3.31	**5.40**	7.84	8.32
12	1x2y12z	3.36	**5.33**	8.01	8.84
12	1x2y6z	3.18	**6.88**	8.89	9.19
12	1x3y4z	3.11	7.13	9.30	10.05
12	2x2y3z	**2.69**	**6.70**	**8.87**	9.79

Table 4.13: Speed-up of MSIP solver with $\alpha = 0.5$ in the Cray T3E.

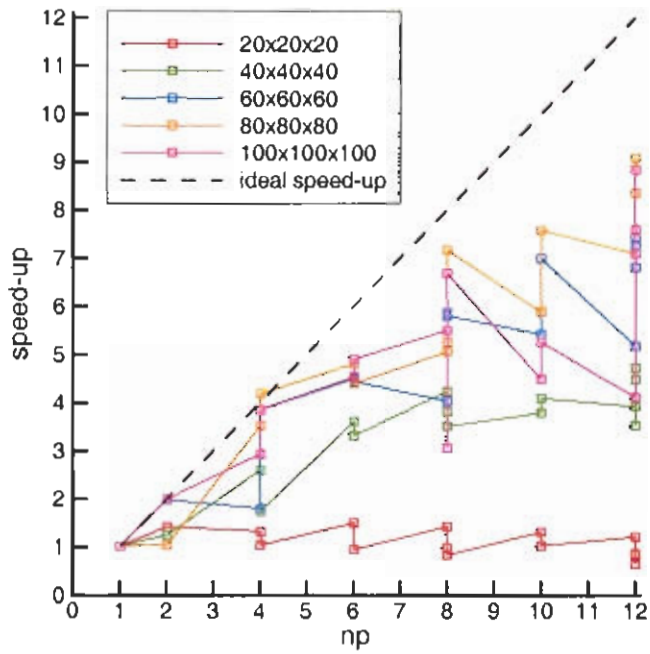

Figure 4.35: Speed-up for the BiCGSTAB preconditioned with MSIP in PC cluster.

np	partition	$20 \times 20 \times 20$	$40 \times 40 \times 40$	$60 \times 60 \times 60$	$80 \times 80 \times 80$	$100 \times 100 \times 100$
1	1x1y1z	1.00	1.00	1.00	1.00	1.00
2	1x1y2z	1.40	1.23	1.96	1.02	1.97
4	1x1y4z	1.30	2.58	1.78	3.50	2.91
4	1x2y2z	1.02	1.72	3.83	4.17	3.83
6	**1x1y6z**	1.48	3.59	4.48	4.79	4.52
6	**1x2y3z**	0.93	3.30	4.42	4.38	4.88
8	1x1y8z	1.41	4.22	4.02	5.04	5.48
8	1x2y4z	**0.97**	**3.80**	**5.85**	5.23	3.04
8	2x2y2z	**0.82**	3.50	5.78	7.15	6.66
10	1x1y10z	**1.30**	3.78	5.40	5.86	4.48
10	1x2y5z	1.01	4.08	**6.97**	**7.56**	5.24
12	1x2y12z	1.20	3.91	5.16	7.07	4.10
12	1x2y6z	**0.84**	4.71	7.40	8.33	7.07
12	1x3y4z	**0.85**	**4.47**	7.24	9.07	7.56
12	2x2y3z	**0.63**	**3.52**	6.79	7.57	8.82

Table 4.14: Speed-up of BiCGSTAB solver preconditioned with MSIP in PC cluster.

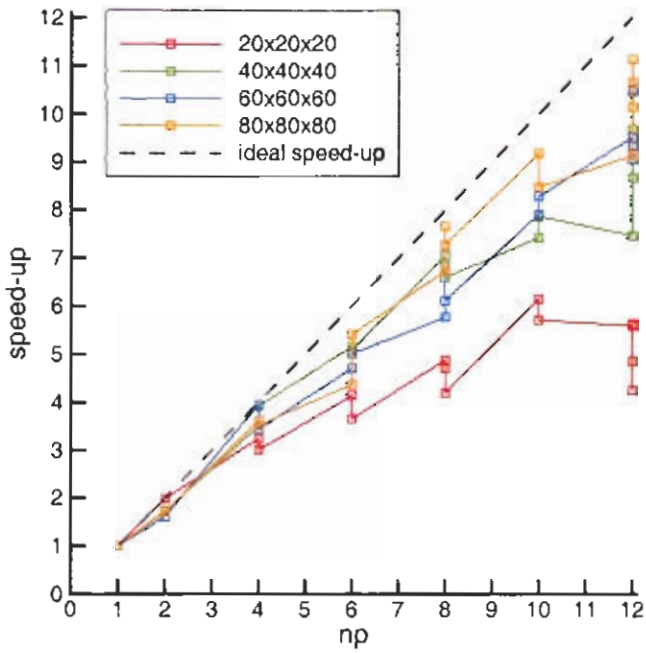

Figure 4.36: Speed-up for the BiCGSTAB preconditioned with MSIP in the Cray T3E.

np	partition	$20 \times 20 \times 20$	$40 \times 40 \times 40$	$60 \times 60 \times 60$	$80 \times 80 \times 80$
1	1x1y1z	1.00	1.00	1.00	1.00
2	1x1y2z	1.97	1.71	1.60	1.69
4	1x1y4z	3.23	3.50	3.95	3.61
4	1x2y2z	2.99	3.91	3.43	3.51
6	1x1y6z	4.13	5.15	4.70	4.36
6	1x2y3z	3.65	5.07	5.00	5.40
8	1x1y8z	4.87	7.05	5.77	6.72
8	1x2y4z	4.71	6.83	7.25	7.65
8	2x2y2z	4.19	6.59	6.11	7.26
10	1x1y10z	6.14	7.41	7.90	9.17
10	1x2y5z	5.70	7.86	8.28	8.47
12	1x2y12z	5.58	7.46	9.50	9.12
12	1x2y6z	4.85	9.66	9.33	11.12
12	1x3y4z	5.62	10.45	10.50	10.13
12	2x2y3z	4.24	8.67	9.04	10.64

Table 4.15: Speed-up of BiCGSTAB solver preconditioned with MSIP in the Cray T3E.

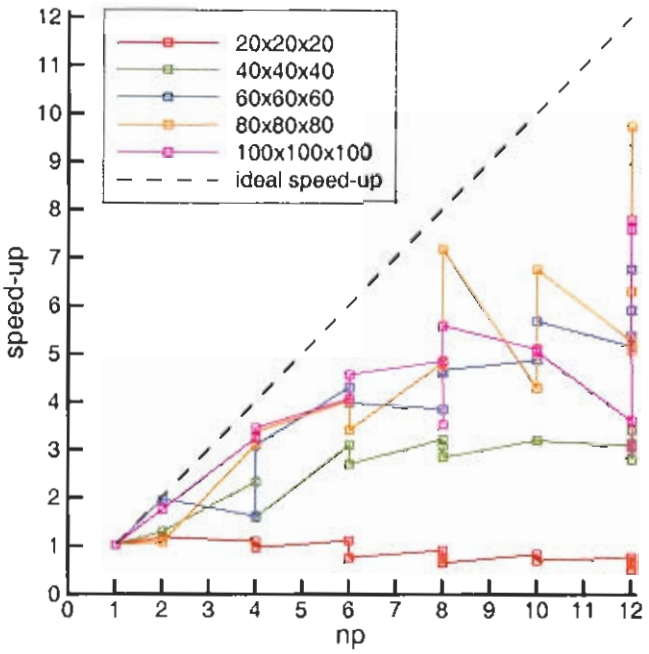

Figure 4.37: Speed-up for the GMRESR preconditioned with MSIP in PC cluster.

np	partition	$20 \times 20 \times 20$	$40 \times 40 \times 40$	$60 \times 60 \times 60$	$80 \times 80 \times 80$	$100 \times 100 \times 100$
1	1x1y1z	1.00	1.00	1.00	1.00	1.00
2	1x1y2z	1.15	1.27	1.94	1.05	1.73
4	1x1y4z	1.07	2.30	1.59	3.08	3.22
4	1x2y2z	0.94	1.56	3.06	3.35	3.43
6	1x1y6z	1.09	3.07	4.27	3.99	4.04
6	1x2y3z	0.73	2.67	3.95	3.38	4.54
8	1x1y8z	0.88	3.18	3.81	4.78	4.82
8	1x2y4z	0.70	3.04	4.58	4.74	3.49
8	2x2y2z	0.62	2.81	4.63	7.14	5.55
10	1x1y10z	0.80	3.17	4.84	4.26	5.07
10	1x2y5z	0.67	3.16	5.64	6.72	5.01
12	1x2y12z	0.74	3.05	5.13	5.24	3.58
12	1x2y6z	0.66	3.37	6.72	6.26	7.55
12	1x3y4z	0.58	3.10	5.87	9.70	7.76
12	2x2y3z	0.49	2.77	5.34	5.04	3.02

Table 4.16: Speed-up of GMRESR solver preconditioned with MSIP in PC cluster.

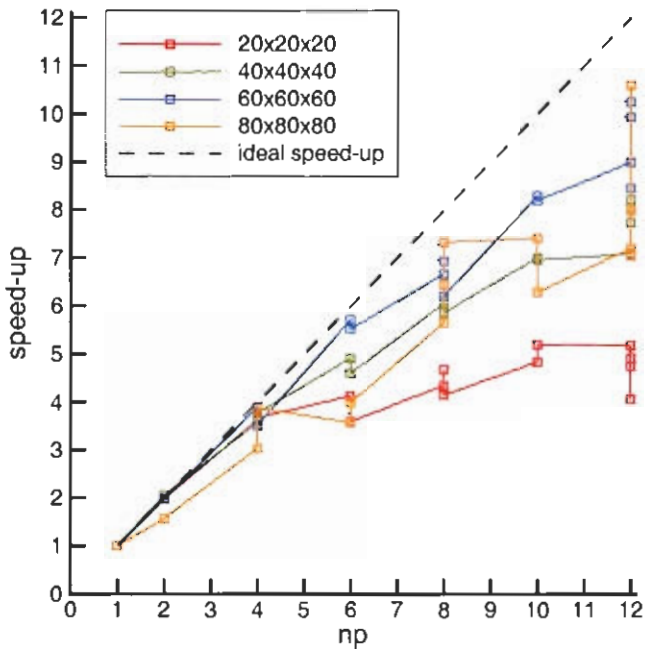

Figure 4.38: Speed-up for the GMRESR preconditioned with MSIP in the Cray T3E.

np	partition	$20 \times 20 \times 20$	$40 \times 40 \times 40$	$60 \times 60 \times 60$	$80 \times 80 \times 80$
1	1x1y1z	1.00	1.00	1.00	1.00
2	1x1y2z	1.99	2.06	1.98	1.56
4	1x1y4z	3.63	3.59	3.90	3.03
4	1x2y2z	3.67	3.78	3.52	3.87
6	1x1y6z	4.13	4.91	5.71	3.57
6	1x2y3z	3.59	4.60	5.52	3.98
8	1x1y8z	4.34	6.05	6.66	5.65
8	1x2y4z	4.67	6.11	6.93	6.44
8	2x2y2z	4.14	5.86	6.20	7.32
10	1x1y10z	4.83	7.00	8.30	7.41
10	1x2y5z	5.19	6.96	8.20	6.28
12	1x2y12z	5.17	7.09	9.00	7.20
12	1x2y6z	4.91	8.06	10.25	7.99
12	1x3y4z	4.74	8.22	9.94	10.60
12	2x2y3z	4.06	7.74	8.46	7.04

Table 4.17: Speed-up of GMRESR solver preconditioned with MSIP in the Cray T3E.

It is shown in these figures that both facilities PC cluster and the Cray T3E have reported similar behaviours in spite of the differences on their respective communication networks, being the speed ups in the PC cluster as good as those of the Cray T3E.

An unexpected result observed in these figures is that matrix-vector product's speed-ups are worst than Jacobi or Gauss-Seidel's speed-ups for a given problem size. The reason may be due to the better computation with communication ratio in solvers. I.e. the computational load in solvers embeds other operations apart of the matrix-vector product (e.g. the difference of vectors for the computation of the residual and its norm) while the additional communication load, (e.g. communication in the inner product) doesn't increase as the computational load.

Apart from the Jacobi and Gauss-Seidel solvers, there is a similar degradation of MSIP solver and MSIP preconditioner based solvers, i.e. BiCGSTAB and GMRESR. The degradation increases for the one partitioning direction while the multiple partitioning direction reduce this degradation. This seems in contradiction with a bigger number of communication messages for a 3D partitioned domain respect to the 1D partitioned domain. The reason is that the global ILU factorization, e.g. MSIP, is better represented by local ILU factorizations in multiple directions.

This fact is also observed in the increment of the number of iterations when the the number of processors in one direction increases. These numbers have been reported in the table 4.18 for the case of $100 \times 100 \times 100$, i.e. a linear system of 10^6 unknowns.

np	partition	Jacobi	Gauss	MSIP	biCGSTAB+prec	GMRESR(10)+prec
1	1x1y1z	6040	3021	634	35	32
2	1x1y2z	6040	3041	666	34	36
4	1x1y4z	6040	3063	700	41	35
4	1x2y2z	6040	3060	712	35	36
6	1x1y6z	6040	3084	731	33	37
6	1x2y3z	6040	3072	730	38	38
8	1x1y8z	6040	3105	763	34	37
8	1x2y4z	6040	3083	745	41	35
8	2x2y2z	6040	3080	755	38	40
10	1x1y10z	6040	3127	794	41	34
10	1x2y5z	6040	3093	761	41	40
12	1x2y12z	6040	3149	826	35	40
12	1x2y6z	6040	3103	776	41	38
12	1x3y4z	6040	3094	770	41	38
12	2x2y3z	6040	3092	772	39	40

Table 4.18: Iterations for the solution of a linear system with 1000000 unknowns.

It is worth noting that the Jacobi number of iterations remains constant and in the case of the BiCGSTAB it also decreases for some partitioning configurations. In order to represent these numbers, the increment of the number of iterations respect to the sequential computation have been calculated and given in terms of the percentage in figure 4.39.

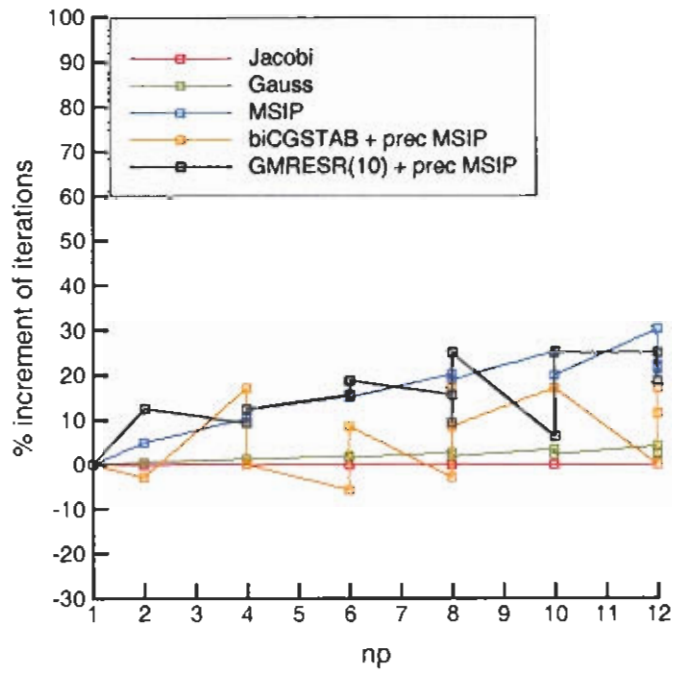

Figure 4.39: Percentage of increment of iterations for the solution of a linear system with 10^6 unknowns when increasing number of processors.

4.9 Nomenclature

A	discretization matrix
a	coeff. in A
b	right hand side
bl	block lenght
d	block of data
E	efficiency
I	unknowns in x-direction
i	index for x-direction
J	unknowns in y-direction
j	similar to i
K	unknowns in z-direction
k	similar to i
N	number of unknowns
n	number of
np	number of processors
ngb	neighbour processors
o	operations of
ov	overlapping area
p	processor identification number
Re	Reynolds number
r	residual
S	speed up
str	stride
t	measure of time
x	unknown
y	auxiliar vector, buffer vector
z	auxiliar vector
$\{x, y, z\}$	cartesian coordinates

Greek symbols

α	network latency
β	network bandwidth
ϵ	precission
ρ	scalar value

Other symbols

$7 - PF$	seven point formulation
$<,>$	inner product
$\|\|.\|\|_2$	2-norm of a vector
$\oplus$	general algebraic operation

Superscripts

(k)	k-th iteration