# Chapter 2: Matrix Multiplication

Almost from the beginning of parallel processing application to numerical problems, different methods for parallelizing matrix multiplication have been studied, designed, implemented and experimented.

From the point of view of the problem in itself, it is useful to have an optimization of this matrix operation since it is always possible to find it in the different applications to be solved in the numerical environment. Thus, the fact that this operation is optimized implies optimizing, in turn, a part of several applications in which it is necessary to multiply matrices.

From a point of view closer to the research, this problem has many characteristics that make it suitable for extensive and intensive study. The two most important characteristics are its simplicity and the possibility of extending its results to other similar operations.

In this chapter, some important aspects of this operation will be included for its parallelization, together with the outstanding characteristics of the already developed parallelization algorithms.

# 2.1 Matrix Multiplication Definition

The definition of the matrix multiplication operation is very simple, all of which simplifies its understanding. Given a matrix $A^{(m \times r)}$ of $m$ rows and $r$ columns, where each element is denoted as $a_{ij}$ with $1 \leq i \leq m$, and $1 \leq j \leq r$; and a matrix $B^{(r \times n)}$ of $r$ rows and $n$ columns, where each element is denoted as $b_{ij}$ with $1 \leq i \leq r$, and $1 \leq j \leq n$; matrix C resulting from the multiplication operation of A and B matrices, C = A × B, is such that each of its elements is denoted as $c_{ij}$ with $1 \leq i \leq m$, and $1 \leq j \leq n$, and it is computed as follows:

$$c_{ij} = \sum_{k=1}^{r} a_{ik} \times b_{kj} \tag{2.1}$$

As with most of linear algebra basic operations, the necessary number of operations between scalars (or "flops" in [59], meaning "floating point operations") for the result matrix computation can be known (computable) in an exact manner. Given the previous definition for matrix multiplication exactly

$$cant\_op = m \times n \times (2r\text{-}1) \tag{2.2}$$

operations (multiplications and sums among scalars) are required. For simplicity's sake, all the analysis is generally carried out in function of square matrices of order $n$, and in this way, the number of basic operations between scalars is exactly

$$cant\_op = 2n^3\text{-}n^2 \tag{2.3}$$

This number of operations is usually called matrix multiplication complexity, and determines the running time necessary to be solved by a computer. In this context, it is also common to find that matrix multiplication is $O(n^3)$ ("of order $n^3$"), emphasizing the fact that the dominant term in Eq. (2.3) is of cubic degree, leaving aside multiplication constants and all the terms of lower degree.

It is worth to mention that this number of operations is independent of the algorithm and/or the computer used for solving the problem. In this sense, it is also important (though not necessarily "essential" in this case) to differentiate this number of operations from, for instance, those performed during the execution of a sequential program based normally in the assignment

$$c_{ij} = c_{ij} + a_{ik} \times b_{kj}$$

which implies the execution of $2n^3$ arithmetical operations between scalars. This is an immediate instance that the programs do not necessarily solve the problems with the minimum number of operations. As stated before, this example do not present many problems as regards total running time, and it is indeed useful for showing that the number of operations solved by a computer is not necessarily the minimal. This fact should always be considered at the time of assessing the performance of computers for the solution of a

particular problem.

The case of parallel computers has to be handled even more carefully since, many times, it is advisable to replicate the computation (and thus, to increase the number of operations effectively solved by processors) to avoid communication or synchronization that would take more running time than the replicated computation. In the analysis of the performance to be carried out in the experimentation, the number of operations in Eq. (2.3) will be used as a reference value to avoid improper conclusions derivable from the number of operations executed in the processor(s).

## 2.2 Linear Algebra Operations

Almost from the very beginning of computers utilization, the developed software was meant to achieve the highest quality in terms of key indexes: performance, reusability and portability. In this sense, the area in charge of solving numerical problem in general and linear algebra problems in particular has not been an exception.

From a long time ago, in the context of linear algebra operations, several libraries have been defined, proposed and developed in order to establish the most reduced and general set of routines or basic operations making use - most of the times (if not always) - of linear algebra applications. One of the first examples is EISPACK [46], based on a set of routines detailed in [123].

The library that has become the de facto standard in the area of linear algebra is LAPACK (Linear Algebra PACKage), developed at the end of the '80s [36] [7] [8]. Apart from having taken advantage of the previous experiences such as EISPACK and LINPACK, together with LAPACK (or at least with a logical development related to this library), two fundamental concepts are added in terms of the library specification's transparency and, also, as regards the maximum local optimization likelihood (according to the computing architecture). These concepts are:

- Basic operations in levels.
- Block algorithms.

In fact, both concepts are closely related, but the division of basic operations in levels is made from the point of view of LAPACK considered as a library for solving linear algebra problems. On the other hand, block algorithms are a consequence of admitting that the architecture of most of the computers (independently whether they are parallel or not) has a hierarchical memory structure where the levels closer to the same processor (cache levels 1 and 2) should be exploited to the utmost in order to achieve the maximum processing capability.

### 2.2.1 BLAS: Basic Linear Algebra Subprograms and Performance

From the subroutines included and defined in LAPACK, a set of subprograms has been

recognized as basic. These subroutines have been called Basic Linear Algebra Subprograms (BLAS) [80] [81] [43]. BLAS is generally divided in three classes (known as levels) in function of the quantity of data over which they operate and in function of the operations quantity required for each of them. There are three BLAS levels [46]:

- Level 1 (or L1 BLAS): for subroutines operating between vectors, such as $y = \alpha x + y$.
- Level 2 (or L2 BLAS): for subroutines operating with matrices and vectors, such as in the equation $y = \alpha Ax + \beta y$.
- Level 3 (or L3 BLAS): for subroutines operating with matrices, such as $C = \alpha AB + \beta C$.

where A, B y C represent matrices, $x$ and $y$ represent vectors, and $\alpha$ and $\beta$ represent scalars.

Beyond the utility of this classification for the characterization and identification of the operations, it was established considering that:

- The amount of data over which level 1 subroutines operate is of $O(n)$, where $n$ represents a vector length, and the number of basic operations between scalars is also of $O(n)$.
- The amount of data over which level 2 subroutines operate is of $O(n^2)$, where $n$ represents square matrices order (row and column quantity), and the number of basic operations between scalars is also of $O(n^2)$.
- The amount of data over which level 3 subroutines operate is of $O(n^3)$, where $n$ represents square matrices order (row and column quantity), and the number of basic operations between scalars is also of $O(n^3)$.

This implies that, on the one hand, subroutines included in Level 3 BLAS are the ones that have more requirements in terms of processing capability. In fact, the difference with Level 2 BLAS is so great -$O(n^3)$ vs. $O(n^2)$- that most of the times (if not all) all the necessary optimization can be achieved by optimizing Level 3 BLAS. On the other hand, it is evident that, as regards subroutines optimization, those of Level 3 BLAS are the most appropriate since they have greater computing requirements than the other two levels. It is generally considered that [46]

- L1 BLAS subroutines cannot achieve a high performance in most of supercomputers. Still, they are useful in terms of portability.
- L2 BLAS subroutines are specially appropriate for some vectorial computers (in terms of performance) though not in all of them due to data movement imposed among the different levels of the memory hierarchy.
- L3 BLAS subroutines are the most appropriate for achieving the maximum performance in current supercomputers, where the memory hierarchy plays a really important role in the performance of all the access to the data processed in the CPU/s.

In fact, even though LAPACK is implemented (or might be directly implemented) in terms of L1 BLAS [46], it is currently designed for using to the utmost L3 BLAS [LAPACK] because these subroutines are almost the only ones with which a high performance can be achieved (closer to the maximum performance of each processor used) in the supercomputers. Thus, there is no doubt that, in terms of performance, it is essential to pay special attention to subroutines defined as Level 3 BLAS.

Why are Level 3 BLAS subroutines specially appropriate for optimizing? The answer is related to the method used for implementing the algorithms - known as block algorithms

[42] [5] [85] [104] [20] [122]. These algorithms optimize the access to memory since they maximize the operations quantity carried out for each referenced datum. In general, they organize the computation so that a data block is acceded and, thus, is implicitly assigned in cache/s memory. The necessary modifications (iteration order, "loop unroll" levels, etc.) are carried out to execute immediately all (or most) of the operations in which that data block is involved and, thus, used to the utmost. In this way, the effective memory time is reduced since the "cache hit" -or the number of times a datum referenced from the processor is immediately found in cache memory- is increased to the maximum.

Block algorithms can be adapted to each structure (or, more specifically, to each cache/memory levels hierarchy) and, for this reason, the term transportable is often used instead of portable. There is a tendency to "adapt" routines with greater optimization chances or with higher potential to achieve the maximum possible performance to the underlying architecture [46]. Most of the companies that design and commercialize processors also provide the complete BLAS-defined subroutines (and even others similar) so that they use to the utmost the processing capability [CXML] [SML] [SCSL1] [SCSL2]. These subroutines are optimized for processors even at the processor's language level (assembly language) and, thus, the effort and the cost invested in its implementation represent also a marker of these subroutines' importance.

### 2.2.2 L3 BLAS and Matrix Multiplication

Originally, L3 BLAS specification is carried out for the FORTRAN language and the defined/included subroutines are [42] [BLAS]:

a. "General" matrix products (subroutines ending in GEMM):

   $C \leftarrow \alpha\ op(A)\ op(B) + \beta C$

   where op(X) can be X, $X^T$ o $X^H$

b. Matrix products where one of the matrices is real or symmetrical complex or hermitical complex (subroutines ending in SYMM or HEMM):

   $C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$

   where A is symmetrical for SYMM or hermitical for HEMM and is located to the left or right of the multiplication depending on a subroutine parameter (SIDE).

c. Matrices products where one of them is triangular (subroutines ending in TRMM):

   $B \leftarrow \alpha\ op(A)\ B$ or $B \leftarrow \alpha\ B\ op(A)$

   where A is a triangular matrix; it is to the left or right of the multiplication depending on a subroutine parameter (SIDE), and op(A) can be A, $A^T$ o $A^H$.

d. Rank-*k* update of a symmetrical matrix (subroutines ending in SYRK):

$C \leftarrow \alpha AA^T + \beta C$ or $C \leftarrow \alpha A^T A + \beta C$

where C is symmetrical and A is to the right or left of the multiplication by $A^T$, depending on a subroutine parameter (TRANS).

e. Rank-*k* update of a hermitical matrix (subroutines ending in HERK):

$C \leftarrow \alpha AA^H + \beta C$ or $C \leftarrow \alpha A^H A + \beta C$

where C is hermitical and A is to the right or left of the multiplication by $A^H$, depending on a subroutine parameter (TRANS).

f. Rank-2*k* update of a symmetrical matrix (subroutines ending in SYR2K):

$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$ or $C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$

where C is symmetrical and A is to the right or left of the multiplication by $B^T$, depending on a subroutine parameter (TRANS).

g. Rank-2*k* update of a hermitical matrix (subroutines ending in HER2K):

$C \leftarrow \alpha AB^H + \overline{\alpha} BA^H + \beta C$ or $C \leftarrow \alpha A^H B + \overline{\alpha} B^H A + \beta C$

where C is hermitical and A is to the right or left of the multiplication by $B^H$, depending on a subroutine parameter (TRANS).

h. Solutions to triangular equations systems (subroutines ending in TRSM):

$B \leftarrow \alpha\ op(A)\ B$ or $B \leftarrow \alpha\ B\ op(A)$

where A is a triangular matrix; it is to the left or right of the multiplication depending on a subroutine parameter (SIDE), and op(A) can be $A^{-1}$, $A^{-T}$ o $A^{-H}$.

Leaving aside $O(n^2)$ operations, such as the calculation of $op(A) = A^T$, notice that every L3 BLAS subroutine has matrix multiplication as prevailing operation (as regards arithmetical operations quantity). In addition, [77] shows how the entire level 3 BLAS can be implemented in terms of the matrix multiplication operation keeping the performance closer to each computer's possible optimum. Within the market context, notice the example of Intel: it published in Internet, together with Pentium III commercialization, a document [74] explaining how to use to the utmost the processor's computing capability in terms of matrix multiplication, apart from providing the processor's users with a library of optimized matrix computing functions.

## 2.3 Matrix Multiplication as Benchmark

Computers performance characterization has been used with several purposes, such as [63]:

- Problem solution capability estimation, regarding both the size of the problems that can be solved and the necessary running time.
- Computers cost verification, not only in terms of hardware but also in terms of base and application software.
- Selection of the most appropriate computer for solving the problem or type of problems. In this case, the throughput index is implicitly used as a comparing parameter of potential computers.

Traditionally speaking, a computer numerical computing capability was characterized with the number of floating point operations per time unit (Mflop/s: millions of floating point operations per second) or by a number identifying it unambiguously [64] [SPEC]. Also, two general lines were traditionally adopted for the computation of this throughput index:

1. Processing hardware analysis: floating point unit/s, floating point units design (pipelines, internal recordings, etc), cache memory/ies (levels, sizes, etc.), main memory capacity, etc.
2. Execution of a specific computing program or set of programs called benchmarks.

In general, the processing hardware analysis results in what is known as peak throughput, or theoretical maximum performance of the computer. This performance characterization line has been adopted by computer manufacturers and is now accepted - an unusual fact for a specific application.

The usage of benchmarks became daily, due to the division that may arise between the peak performance and the real performance normally achieved by the application execution in the computers. It is really difficult to choose a set of programs that fulfils the characteristics of representing all the scope of possible applications executable on a computer. Thus, there exist many benchmarks that are used and many more proposed.

If the type of specific applications on which computers are to be used is well defined, the characterization in this specific application field without employing the most general benchmarks is still very useful. This is the case of the applications defined in terms of matrix multiplications and, thus, the same matrix multiplication performance is what can be more precisely obtained in this field and what is considered as the reference benchmark in terms of throughput.

Using a benchmark so specific and so close to the application to be solved has -in the context of parallel programs executed over heterogeneous hardware - another advantage: it accurately defines the workstations relative speed for local processing. Although this index (computing relative speed) is not so necessary nor so important within the parallel computers context with homogeneous processing elements, it is indeed essential for parallel computation with heterogeneous processing elements. Without this type of information, it is really difficult to achieve a computational load balance (at least, statically).

### 2.3.1 Level 3 BLAS Benchmark

It is evident that, if the whole level 3 BLAS is to be directly implemented in terms of matrix multiplication [77], the obtained performance will almost be that of the same matrix multiplication. But, if we choose to implement each subroutine (L3 BLAS) taking advantage of its computing features optimally and independently of the matrix multiplication, a performance very similar to that of the same matrix multiplication might still be obtained. In this way, matrix multiplication is a good "representative" (and with this, a benchmark is constituted) in terms of the throughput of level 3 BLAS routines.

A more solid argument to back up the consideration of matrix multiplication as representative in relation to the whole level 3 BLAS routines throughput would be that, at least in the sequential scope, the performance obtainable by each subroutine of level 3 BLAS is similar to the one that can be obtained with matrix multiplication [122] - a fact which is experimentally proved. In this sense, knowing the throughput obtained with matrix multiplication, a quite concrete idea of the performance obtainable with all of level 3 BLAS subroutines can be traced.

### 2.3.2 As a "General" Benchmark

In the field of benchmarks in general, i.e. of the programs that are intended to be used to identify computers computing capability (parallel or not), matrix multiplication representativeness is much more debatable. In fact, there exists a large quantity of researchers and companies that consider that the only thing that can be a benchmark is a real application [64]. However, in some benchmark distributions of free use for parallel machines [68] [94], it is still included as a "lower level benchmark".

In some way or another, results are still been reported in relation to matrix multiplication throughputs in parallel and sequential computers. One of the main reasons is that, with matrix multiplication, a near-optimal performance of the computer used can be attained. In this sense, matrix multiplication has become, in some way or another, a quality metrics of the implementation of numerical algorithms or, at least, of algorithms related to linear algebra operations. An example, academic in principle, is constituted by ATLAS [122] [ATLAS] that, only as a commercial example, [SCSL2] tries to show how good is the scientific computing library assuring that, for mono-processor machines, the performance exceeds the theoretical 95% and, for 64-processors-parallel machines, the relative global performance exceeds the theoretical 85%. The idea in this sense is that "there exists some code that solves at least one linear algebra problem with near-optimal performance of each process", with the intention of extrapolating this fact to at least a subset of the problems to be solved in the computer/s.

## 2.4 Matrix Multiplication Parallelization

For academic reasons (and simplicity), one of the first parallel algorithms explained in textbooks of parallel processings is that of matrix multiplication [56] [82] [79] [124] [10] [58] [52] [3]. However, beyond its academic relevance, the research has been updated throughout the years by its importance as a problem to be solved, and this is demonstrated by several publications with this respect, some of which are the ones previously mentioned and some others are [23] [35] [30] [120] [26] [83].

Matrix multiplication has very specific characteristics as regards the design and implementation of a parallel algorithm in the parallel algorithms context in general:

- Computation independency: each element computed from the result matrix C, $c_{ij}$, is, in principle, independent of all the other elements. This independence is utterly useful because it allows a wide flexibility degree in terms of parallelization.
- Data independence: the number and type of operations to be carried out are independent of the data. In this case, the exception is the algorithms of the so-called sparse matrix multiplication, where there exists an attempt to take advantage of the fact that most of the matrices elements to be multiplied (and thus, of the result matrix) are equal to zero.
- Regularity of data organization and of the operations carried out on data: data are organized in two-dimensional structures (the same matrices), and the operations basically consist of multiplication and addition.

The first characteristic makes matrix multiplication specially appropriate for parallel machines called multiprocessors, where a set of processors, or processing elements, share the same memory. Parallel algorithms for multiprocessors often follow the basic lines of decomposition or division of data to be computed and/or of Divide-and-Conquer recursively. In general, all of them have a previous static or dynamic period for partitioning or dividing data quantity (or parts of the multiplication result matrix) to be computed in each processor and, eventually, a subsequent utilization period of intermediate computations to compute the final result.

The last two characteristics make the proposed algorithms for parallel matrix multiplication follow SPMD (Single Program - Multiple Data) parallel computing model in general [52] [116]. In this way, a same program is executed asynchronously in each processor of the parallel machine and it is eventually synchronized and/or communicated to the other processors. It is worth to mention that SPMD is independent of whether the implementation is carried out on a multiprocessor or multicomputer parallel machine, or on a parallel computer with processing architecture distributed as a workstation network.

In general, it is really difficult to find in literature (above all in textbooks dedicated to explain how to carry out parallel processing) parallel algorithms for a certain type of parallel computing architecture. Even though algorithms can be adapted in a more or less complex way to each of the available parallel processing architectures, it is also true that there exists an adaptation and implementation cost at an algorithmical level. And even more important is the fact that, in the context of applications with great computing requirements, the cost in terms of throughput obtained may be too high. Then, in most of the cases, a close relation between each algorithm and a particular parallel computing

architecture can be found. That is why, in the following sections, each of the mentioned algorithms will be directly related to the underlying parallel computing architecture with which the best results will be obtained according to the performance that is or can obtained.

### 2.4.1 Parallel Algorithms *for* Multiprocessors

As previously stated, algorithms following computation division or decomposition principles and those of "Divide and Conquer" recursively are considered the most appropriate for shared memory parallel computers or multiprocessors. In fact, in the computation $C = A \times B$, a first attempt would be to divide directly the computation of C in as many parts as processors can be used. In this sense, the fact that matrices A and B are accessed only for reading their elements and that matrix C is only accessed for writing its elements (what was previously explained as computation independence) is highly positive.

**Direct Partitioning**. As Figure 2.1 shows, with a certain stream for each processor $P_1$, ..., $P_4$, each of them can access matrices A and B data without being synchronized with the other data (except at a physical level, depending on the shared memory organization) since the data of both matrices are accessed only for reading. In the same way, each processor can access matrix C independently of the others in order to store each of the elements to be computed in function of the elements of A and B.

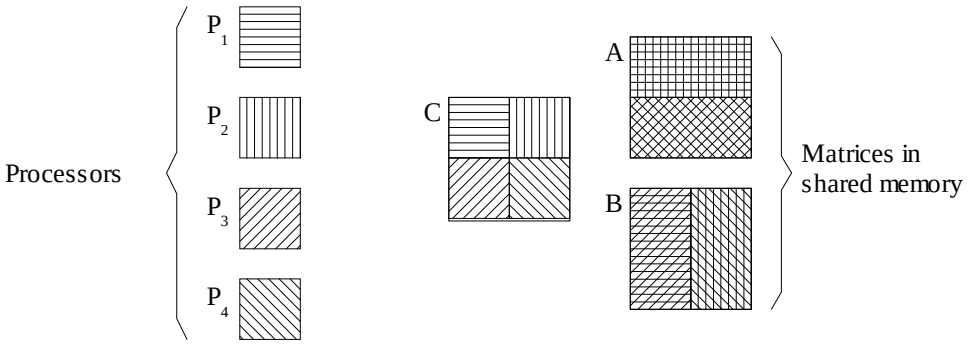



**Figure 2.1**: Multiplication Computation Division in Multiprocessors.

This method to carry out computations would at least need an initial phase to determine the part to be processed by each processor and a final synchronization to determine when all of processors have finished the computations and, thus, when the result is thoroughly computed. On the other hand, no data replication type is added despite the fact that some parts of matrices A and B are used by more than one processor, since all data are stored in the shared memory.

The fact that more than one processor has access to the same part of a matrix (A or B) may cause drawbacks in relation to the simultaneous accesses to a memory. In this sense, and depending on the shared memory design, processors can be sequentialized and, thus, penalized as regards their throughput. However, these problems can be easily solved since:

- they can be intercalated in the access to different parts of a same matrix. In the example

of Figure 2.1, for instance, processor $P_1$ might begin the access to matrix A from the first row onwards and processor $P_2$ from the last row (through which it access) towards the first one.

- the different levels of intermediate cache memory (between each processor and the main shared memory) together with block-computing algorithms highly reduce the number of access to the shared memory.

It is worth to mention the simplicity of the division, taking advantage of the same multiplication characteristics together with multiprocessors homogeneity as regards the computing capability of each computing element (processors).

**Recursive Divide-and-Conquer**. The idea of carrying out the multiplication in parts or submatrices is used up in this type of algorithms [70] [62] for the processing parallelization. The algorithm in pseudo-code can be expressed as Figure 2.2 shows [124]:

```
mat_mul(A, B, C, s)
/* A, B: matrices to multiply */
/* C: result matrix */
/* s: matrices size */
{
   if (sequential multiplication)
   {
      C = A×B;
   }
   else
   {
      mat_mul(A00, B00, C00, s/2);   /* (1) */
      mat_mul(A01, B10, C100, s/2);   /* (2) */
      mat_mul(A00, B01, C01, s/2);   /* (3) */
      mat_mul(A01, B11, C101, s/2);   /* (4) */
      mat_mul(A10, B00, C010, s/2);   /* (5) */
      mat_mul(A11, B10, C110, s/2);   /* (6) */
      mat_mul(A10, B01, C011, s/2);   /* (7) */
      mat_mul(A11, B11, C111, s/2);   /* (8) */
   }
   C00 = C00 + C100;
   C01 = C01 + C101;
   C10 = C010 + C110;
   C11 = C011 + C111;
}
```

**Figure 2.2**: Recursive Divide-and-Conquer Pseudo-Code.

where:

- Each matrix A, B, and C is divided in four equal parts, as Figure 2.3 shows. This number of parts of each matrix is directly related to the number of recursive calls that must be carried out to obtain intermediate computations.
- Most of the operations are carried out in the recursive calls to `mat_mul`, and the last four addition operations between intermediate computations (sub)matrices $C0_{ij}$ and $C1_{ij}$

($0 \leq i, j \leq 1$) are to be carried out to obtain the correct result of each of C's matrix submatrices, as Figure 2.3 shows. These last operations can be carried out in a subset of processors used for solving multiplications of recursive calls.

- Each recursive call to mat_mul numbered from (1) to (8) can be executed in a different processor, depending on the quantity of available processors and the performance obtained according to the quantity of matrices data to be multiplied.
- The condition (`sequential multiplication`) can be given in function of the size of the matrices to be multiplied (s = 1, in an extreme case) or the number of recursive calls that determine, in turn, the number of processors to be used simultaneously for partial results computation.

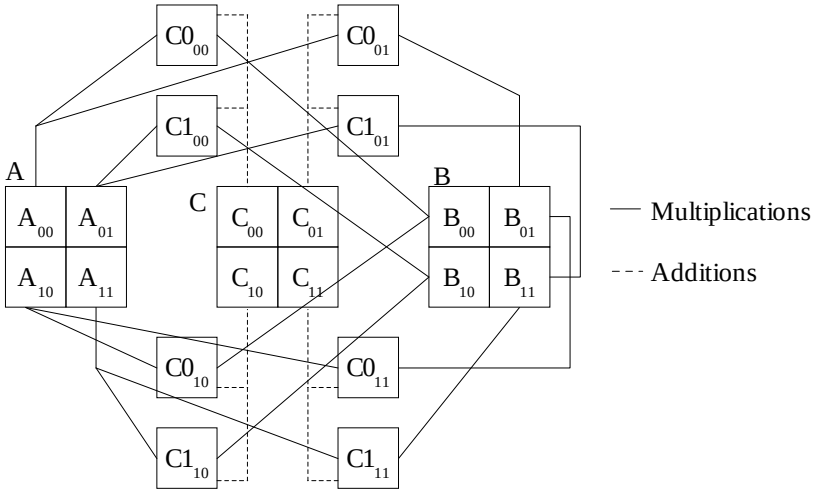



**Figure 2.3**: Recursive Divide-and-Conquer Submatrices and Computations.

For this algorithm, like for the direct partitioning algorithm, it is also interesting to notice that the computation division (and the subsequent parallelization) is highly favored by the homogeneity in the multiprocessors processing elements.

Notice also that, as it is expressed, the required space for data increases considerably, taking into account the fact that for each block of the result matrix C, $C_{ij}$, there are two blocks of intermediate data $C0_{ij}$ and $C1_{ij}$. However, with some modifications - reducing the parallelism in the quantity of recursive calls or increasing the dependency between intermediate computations with data blocks - this extra memory requirement can be avoided.

Figure 2.4 shows Figure 2.2 pseudo-code modification carried out to avoid the fact that the required memory quantity is greater than the required memory quantity for the sequential algorithm. In this way, `mat_mul` is modified to carry out a multiplication and an addition (BLAS_GEMM style) instead of a multiplication only, becoming `mat_mul_sum`. All the processing by blocks is kept, though now there are recursive calls pairs to `mat_mul_sum` that use a same block of C. These are the numbered calls, with (1) and (2), (3) and (4), (5) and (6), and (7) and (8) respectively. The use of a same block of matrix C in recursive calls pairs implies that:

- Two intermediate data blocks are no longer necessary (with respect to `mat_mul`) for a single block of the result matrix.
- Recursive calls numbered from (1) to (8) are no longer independent of each other but there exist data dependency between calls pairs and, thus, they would not be executed simultaneously. In this sense, the number of multiplications that can be simultaneously carried out is reduced from 8 to 4 (in several processors).
- Final additions appearing in Figure 2.2 are no longer necessary because they are solved directly in the same subroutine `mat_mul_sum`.

```
mat_mul_sum(A, B, C, s) /* C = A×B + C */
/* A, B: matrices to multiply */
/* C: result matrix */
/* s: matrices size */
{
   if (sequential multiplication)
   {
      C = A×B + C;
   }
   else
   {
      mat_mul_sum(A00, B00, C00, s/2);   /* (1) */
      mat_mul_sum(A01, B10, C00, s/2);   /* (2) */
      mat_mul_sum(A00, B01, C01, s/2);   /* (3) */
      mat_mul_sum(A01, B11, C01, s/2);   /* (4) */
      mat_mul_sum(A10, B00, C10, s/2);   /* (5) */
      mat_mul_sum(A11, B10, C10, s/2);   /* (6) */
      mat_mul_sum(A10, B01, C11, s/2);   /* (7) */
      mat_mul_sum(A11, B11, C11, s/2);   /* (8) */
   }
}
```

**Figure 2.4**: Recursive Divide-and-Conquer Modification.

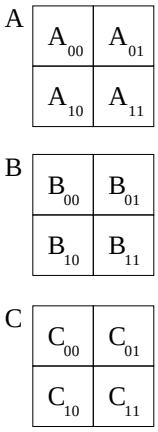
What in the direct partitioning algorithm is the initial phase of matrices division, in this algorithm would be the recursive calls solved by different processors.

**Strassen's Method Parallelization**. Strassen's method is one of the most innovative in terms of matrix multiplication sequentially solved [114]; in [59], it is also called Divide-and-Conquer algorithm and presented as a recursive algorithm. Figure 2.5-a) shows intermediate computations assuming that matrices to be multiplied are divided in four parts or submatrices or even blocks as those of Figure 2.5-b).

Although it is difficult to compute exactly the arithmetical operations quantity for this method, the multiplication operations quantity is usually computed (or estimated in terms of order of magnitude) assuming that the quantity of additions is nearly equal [59]. Under this consideration, Strassen's method has an advantage: it reduces the complexity or number of operation among floating point numbers to $O(n^{\log_2 7})$, taking as reference the multiplication conventional method that is of $O(n^3)$. We can also mention as an advantage the fact that it can be implemented by using recursion.

$P_0 = (A_{00} + A_{11}) \times (B_{00} + B_{11})$
$P_1 = (A_{10} + A_{11}) \times B_{00}$
$P_2 = A_{00} \times (B_{01} - B_{11})$
$P_3 = A_{11} \times (B_{10} - B_{00})$
$P_4 = (A_{00} + A_{01}) \times B_{11}$
$P_5 = (A_{10} - A_{00}) \times (B_{00} + B_{01})$
$P_6 = (A_{01} - A_{11}) \times (B_{10} + B_{11})$
$C_{00} = P_0 + P_3 - P_4 + P_6$
$C_{01} = P_2 + P_4$
$C_{10} = P_1 + P_3$
$C_{11} = P_0 + P_3 - P_1 + P_5$

a) Submatrices and Computing




b) Matrices Partitioning

**Figure 2.5**: Strassen's Method.

From Strassen's method sequential implementation point of view:

- There is generally a special emphasis on the fact that the operations between matrices elements are different for those conventionally defined and, thus, the effects of rounding and numerical stability can be really different depending on the values of matrices elements to be multiplied [59].
- In a similar way to what happens with the previously mentioned algorithm (recursive Divide-and-Conquer), intermediate data are necessary to reach the definitive values to be computed: blocks $P_0$, ..., $P_6$ of Figure 2.5-a). Unlike the algorithm presented as recursive Divide-and-Conquer, the removal of those intermediate data blocks is really difficult and, in fact, is not considered. That is why Strassen's method memory requirements are rather higher than those of the traditional method.

From Strassen's method parallelization point of view:

- Considering shared memory multiprocessors is immediate given that, as the recursive Divide-and-Conquer method, there exist several multiplications that can be carried out simultaneously. In the case of Strassen's method, they are seven: the computation of each $P_k$, with $0 \leq k \leq 6$.
- There exists a computational load unbalance both in the intermediate block computation $P_i$ and in the definitive blocks computation $C_{ij}$ as from $P_k$. For instance, in order to compute $P_0$, two blocks addition and one multiplication are necessary, but to compute $P_1$, one addition and one multiplication are necessary. This affects both the number of data accessed and the number of operations between scalars to be carried out. Anyhow, a special emphasis has to be placed on the fact that these differences are at an $O(n^2)$ operations level (additions and subtractions) vis-à-vis multiplication operations with complexity of $O(n^3)$ or $O(n^{\log_2 7})$.

It is important to remember that the idea of diving the matrices to be used/computed in blocks is intensively and extensively utilized in all sequential matrix algorithms as well as in the parallel ones because, as stated before, it allows the organization of the computation by blocks and, in this way, the use of the memory cache/s is considerably increased. Since

matrices to be processed are usually very large (in general, all the main available memory is used and, in some cases, the swap memory space is used as well), this type of computation organization is indispensable in order to obtain an acceptable performance. In other words, without processing by blocks, the access time to the data is, in several orders of importance, greater than what the processor needs to operate at its maximum speed or, at least, an important fraction of the possible maximum.

## 2.4.2 Parallel Algorithms *for* Multicomputers

Most of the reports on parallel algorithms for matrix multiplication (and similar problems) are those dedicated to the design taking into account the fact that the underlying computing architecture will be that of a multicomputer [59] [117]. On the one hand, multicomputers have always been considered more scalable than multiprocessor and, on the other, the design and development of multicomputers have always been constant throughout the time, reason why they have become more attractive for parallel algorithms development.

**Systolic Array or Processors Mesh**. Despite the fact that this way of matrix multiplication is originally thought for SIMD type computers - or simply for being directly implemented in hardware [82] [124]- it can be generally applied considering matrix blocks like in the previous algorithms. Figure 2.6 shows the initial display of data and processing elements for multiplying two matrices of 3×3 elements.

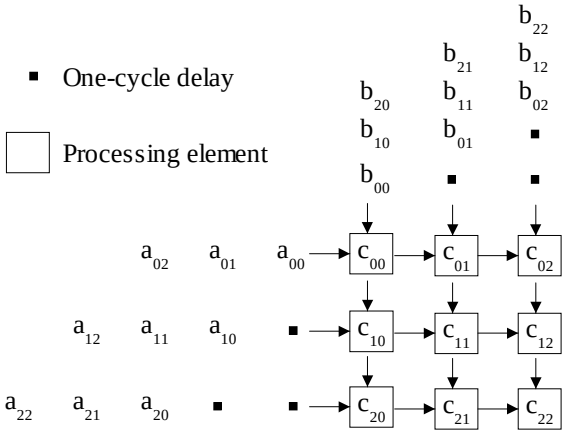



**Figure 2.6**: Multiplication of 3×3 in a Mesh Array.

It is clear that the processing or processors elements are interconnected in a mesh or bi-dimensional array. Normally, communication operations (indicated by the arrows and all of the multiplication and addition operations potentially carried out by each processing element depending on the available data) are counted as a "cycle" or "step" of the processing.

Figure 2.7-a) shows the first step of the processing where:

- all the elements of the matrices A and B "move forward" in the direction of the

corresponding arrows, and elements $a_{00}$ and $b_{00}$ reach the processor dedicated to compute $c_{00}$.
- the first operation for the computation of $c_{00}$, i.e. $a_{00}$ x $b_{00}$.

Figure 2.7-b) shows the second step of the processing where:
- all the elements of matrices A and B "move forward" once again in the direction of the corresponding arrows; elements
  - $a_{01}$ and $b_{10}$ reach the processor dedicated to compute $c_{00}$,
  - $a_{00}$ and $b_{01}$ reach the processor dedicated to compute $c_{01}$,
  - $a_{10}$ and $b_{00}$ reach the processor dedicated to compute $c_{10}$.
- the potential operations are carried out in this step for the computation of $c_{00}$, $c_{01}$ and $c_{10}$, i.e. $c_{00}^{(2)}=c_{00}^{(1)}+a_{01}\times b_{10}; c_{01}^{(2)}=a_{00}\times b_{01}; c_{10}^{(2)}=a_{10}\times b_{00}$ .

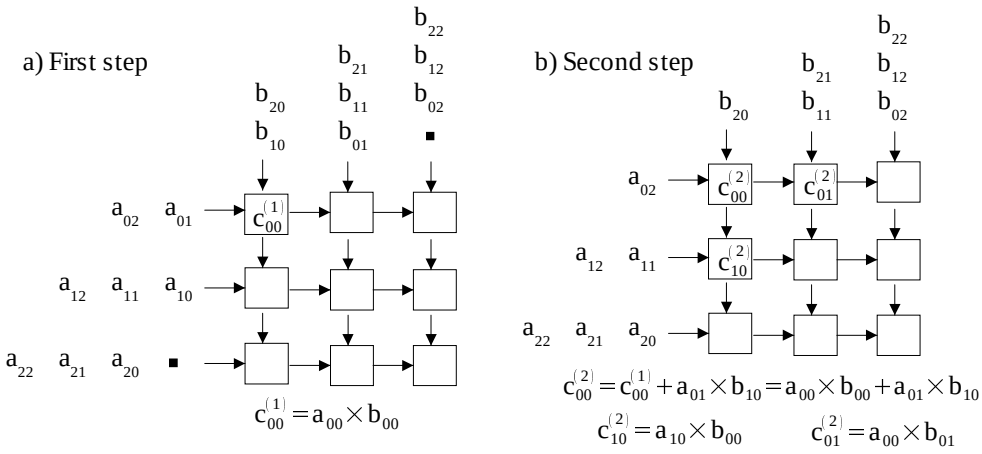



**Figure 2.7**: First two steps of a Multiplication in a Mesh.

The following conditions should be fulfilled so that this type of processing has an optimal performance:
- All the communications are carried out simultaneously. In the case of Figure 2.6 and Figure 2.7 this implies that all the processing elements may be able to carry out simultaneously (overlapped in time) up to:
  - two communications for data reception.
  - two communications for data sending.
- Overlapped computation with communications, i.e. arithmetical operations are carried out at the same time the data are being sent and received. In this case, in the steps previously explained, the computation of  is overlapped (carried out simultaneously) with the communications corresponding to the second step of the processing.
- I/O simultaneous operations, considering that the communications towards the first row and first column of the bi-dimensional array of Figure 2.6 and Figure 2.7 imply I/O. In case there exists lack of data I/O in parallel, all the matrix data should be initially in the processors' first row and first column, what would imply, in turn, different memory requirements for the different processors depending on their allocation in the mesh.

This parallel matrix multiplication algorithm is very simple and well defined for many

reasons:

- Computation distribution simplicity and naturalness: a processor or a processing element is basically and exclusively dedicated to computing a part of the result matrix, be it an element, a submatrix or a block. At this point, once more, the processing elements are assumed to be homogeneous and, thus, the distribution is trivial.
- "Fixed" and a priori known communications pattern: all data broadcasts are point-to-point (between two processors) and previously known in terms of quantity and type of broadcasts (how many data and between which processors).
- Memory requirements equal to all the processors: there is no processor that should have more or less data than the rest of them, even if it has the necessary buffers for the communications since they are the same in all the processors.

In particular, the first two characteristics turn very simple the load balance over the processors (in terms of computation to be carried out) and over the interconnection network (in terms of data broadcast to be carried out between the processors).

**Cannon's Algorithm**. It is also proposed for a two-dimensional array of processing elements [23] [79] interconnected as a mesh and with the edges of each row and column interconnected, i.e. making up a structure called torus (as Figure 2.8 shows) for 3×3 processing elements.

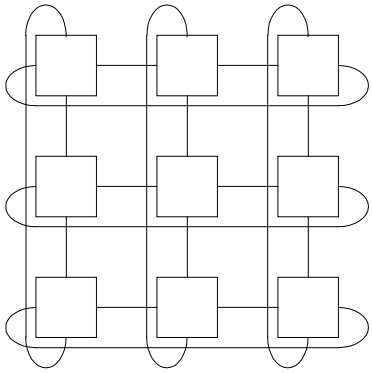
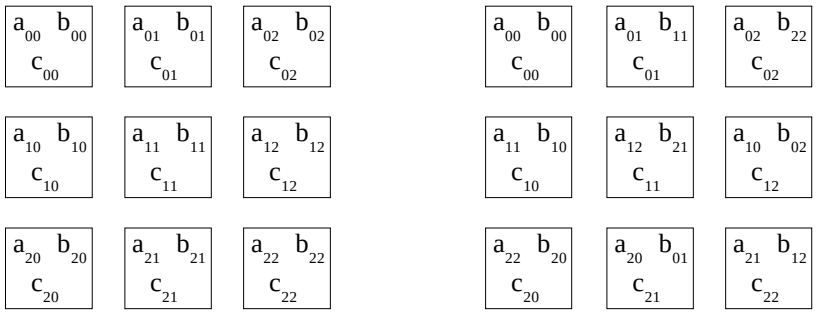


**Figure 2.8**: 3×3 Torus of Processing Elements.

Initially, matrices A, B, and C data distribution is similar to that defined previously in the mesh, i.e. if the processors are numbered according to their position in the two-dimensional array, processor $P_{ij}$ ($0 \le i, j \le P-1$) has the elements or blocks of position $ij$ ($0 \le i, j \le P-1$) of matrices A, B and C. In order to simplify the explanation, matrices elements shall be used instead of blocks. From this data distribution, matrices A and B data are "realigned" or reassigned so that, if there is a two-dimensional array of P×P processors, the element or submatrix A in row $i$ and column ($j$+i) mod P, $a_{i,(j+i)\text{mod}P}$, and also the element or submatrix of B in row ($i$+$j$) mod P and column $j$, $b_{(i+j)\text{mod}P,i}$, are assigned to processor $P_{ij}$. In other words, each data of row $i$ ($0 \le i \le P-1$) of the elements or submatrices of A are transferred or shifted $i$ times towards the left processors, and each data or column j ($0 \le j \le P-1$) of the elements or submatrices of B are transferred or shifted $j$ times towards upper processors. Figure 2.9 shows the initial assignment a), and initial relocation b), imposed by Cannon's

algorithm for matrices of 3×3 elements in a 3×3 processors torus (for simplification, processors' interconnections are not shown in the figure).




a) Initial Assignment. b) Initial Relocation.

**Figure 2.9**: 3×3 Data Location for Cannon's Algorithm.

From the initial relocation, the following steps are carried out iteratively:
- Local multiplication of data assigned in each processor for a partial result computation;
- Left rotation of the elements or submatrices of A;
- Upwards rotation of the elements or submatrices of B;

and after P of these steps, thoroughly computed values of matrix C are finally obtained.

Summarizing, the outstanding characteristics of this way to carry out matrix multiplication are:
- By the way matrices A and B data are communicated, this is an "initial alignment and rotating" algorithm.
- Load balance, both in terms of computation and communications, is assured only if the processing algorithms are homogeneous.
- As in the processing defined for the processors grid, the running time is minimized if the computation can be overlapped in time with communications.
- Matrices A and B data distribution is not the initial one when the matrix multiplication computation is finished.

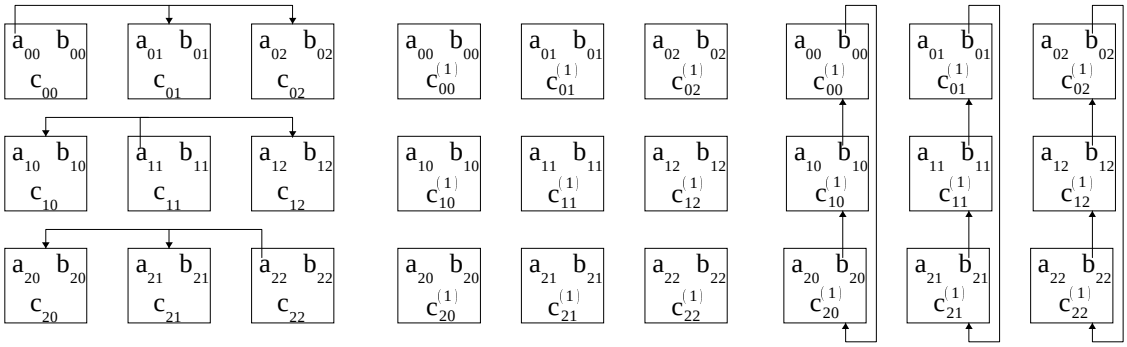
It is worth to mention that the last characteristic becomes really important since, as previously stated and as with all L3 BLAS routines, such distribution is part of some application - not necessarily the same application. Thus, any subsequent processing will have to consider this new matrix distribution or, after Cannon's algorithm, the matrices will have to be realigned in order to retrieve the initial data assignment.

**Fox's Algorithm**. It is also suggested for a two-dimensional array of processing elements [57] [56] [58] [79] interconnected as a mesh and with the edges of each row and column interconnected, i.e. making up a structure called torus. Once more, matrices A, B, and C data distribution is similar to that defined before, i.e. if the processors are numbered according to their position in the two-dimensional array, processor $P_{ij}$ ($0 \le i, j \le P-1$) has the elements or blocks of position $ij$ ($0 \le i, j \le P-1$) of matrices A, B and C - as shown in Figure 2.9-a). In this case, no initial step of data relocation is defined; instead, the

algorithm is iteratively defined from this point. In the iterations or step *k*:

- The block or datum of matrix A stored in the processor of position *i*, *i*+*k* mod P is sent to all the processors of row *i* in the two-dimensional array broadcast by rows.
- In each processor, A's received data are multiplied by B's data locally stored, thus obtaining a partial result of matrix C.
- B's data are upwardly rotated as in Cannon's algorithm.

Figure 2.10 shows the processing corresponding to the first step of Fox's algorithm in a 3×3 processors two-dimensional array, and Figure 2.11 shows the processing corresponding to the second step of Fox's algorithm in a 3×3 processors two-dimensional array.



$$c_{00}^{(1)} = a_{00} \times b_{00};\ c_{01}^{(1)} = a_{00} \times b_{01};\ c_{02}^{(1)} = a_{00} \times b_{02}$$
$$c_{10}^{(1)} = a_{11} \times b_{10};\ c_{11}^{(1)} = a_{11} \times b_{11};\ c_{12}^{(1)} = a_{11} \times b_{12}$$
$$c_{20}^{(1)} = a_{22} \times b_{20};\ c_{21}^{(1)} = a_{22} \times b_{21};\ c_{22}^{(1)} = a_{22} \times b_{22}$$

a) Broadcast. b) Local Computing. c) Shift.

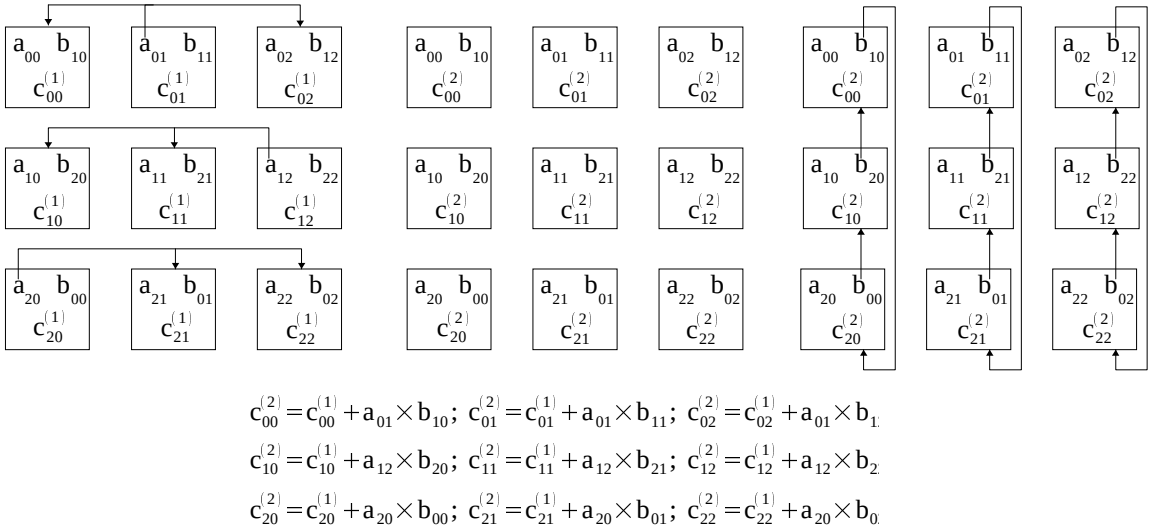
**Figure 2.10**: First step of fox's algorithm, 3×3 Matrices.

The main characteristics of Fox's algorithm are:

- As with the previous cases, the load balance in terms of the computation that each processor must carry out is really simple assuming that the processors are homogeneous.
- The load balance with respect to the intercommunication network cannot be analyzed as easily as Cannon's algorithm, since not only should point-to-point communications be bear in mind, but also collective communications: broadcast by rows. Even so, the communications network could have load balance in terms of communications by columns since they are all of point-to-point nature; and by rows, since they are all broadcast communications. From the point of view of the communications generated in each processor, they all generate the same load over the interconnection network:
  - an upwards data delivery of matrix B.
  - a data reception of matrix B from below.
  - a delivery or reception of data broadcast of matrix A towards the remaining row processors or from a processor of the same row, respectively.
- Since all the processors have the same quantity and type of communications, the buffers will be the same in all the processors - in case they are necessary.
- Memory requirements in each processor are greater in comparison with the Cannon's algorithm because all the processors should have:

- a block or a submatrix or A local matrix.
- a block or a submatrix or B local matrix.
- a block or a submatrix or C local matrix.
- another block for receiving broadcast by rows of block A communicated in each step.

- Unlike Cannon's algorithm, once matrix multiplication is finished, the data of each of the matrices remain like in the initial assignment.




a) Broadcast. b) Local Computing. c) Shift.

**Figure 2.11**: Second sep of Fox's Algorithm, 3×3 Matrices.

As regards broadcast communication by rows, the fact that the only communications that could be labeled predefined are the ones carried out between neighbor processors should be taken into account- both in a mesh and in a torus. In this sense, some additional effort is required, possibly penalized in terms of performance, in order to carry out any of the collective communications (including broadcast by rows) that can also be defined as a multicast from the point of view of a communication complete network. In this sense, there exist several publications dedicated to implementing or, in some way or another, adapting the algorithm to the torus static interconnection [30][120][26]. The basic idea consists in implementing a broadcast via multiple overlapped point-to-point communications, i.e. while a broadcast is being carried out with point-to-point communications, the point-to-point transmissions for the following broadcast can be started.

It is also possible to reduce the memory requirement, which in principle can be relatively greater than the defined in the Cannon's algorithm (more specifically, a third part greater). The key of the reduction is given in the sub-blocks data communication of matrix A assigned in each processor. Instead of carrying out a broadcast in all the data of matrix A, two broadcasts can be carried out (for instance, each with half of the data in matrix A assigned locally). This automatically reduces the extra memory requirement (taking as reference Cannon's algorithm) to half of the data of matrix A that each processor locally

has. Similarly, ten broadcasts can be carried out, each of them with the tenth of matrix A assigned locally, thus reducing to the tenth the extra quantity of necessary memory.

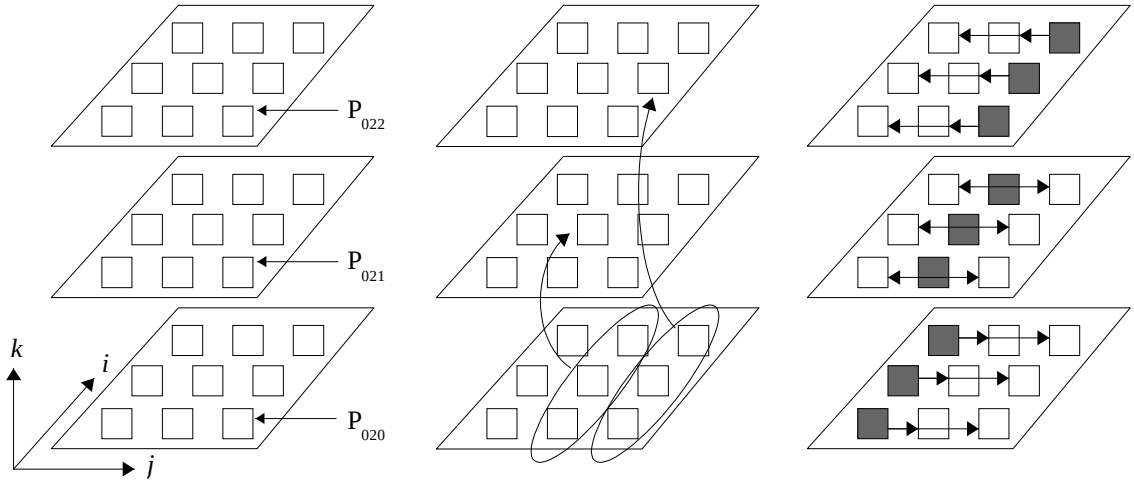
It is worth to mention the fact that when the processing is finished, the matrix data remain assigned in each processor in the same way as they were before beginning the matrix multiplication. In this sense, and recalling what was previously mentioned in the Cannon's algorithm (that matrix multiplication must be carried out in the context of other operations), this represents a substantial advantage.

In fact, both for solving the problem of making a broadcast in a torus and for reducing the necessary memory quantity in order to store the data locally, the same principle for accelerating the access to the data in the memory hierarchy is used, assuming the existence of one or more levels of cache memory: processing by blocks. Instead of handling all the data of matrices A, B and C assigned locally, each of the submatrices is considered by blocks, and these are the ones locally processed (making the multiplications of partial matrices). They are also the ones that are communicated through point-to-point connections in a same processors row making up a communications pipeline with which the broadcast is finally solved. Thus, since they are sub-blocks or parts of matrix A assigned locally, the quantity of extra memory necessary in each processor is reduced to a sub-block or set of sub-blocks that is transmitted in a same point-to-point communication between processors of the same row.

**DNS Method and Meshes of Trees**. DNS algorithm (due to the authors' surnames: Dekel, Nassimi, Sahni) [35] [100] [79] proposes the multiplication of square matrices of order $n$ in a multicomputer with its processors interconnected in a three-dimensions hypercube. Once more, the basic description of the processing will be made for matrix elements, but its extension/application to blocks or submatrices is immediate.

As starting point for the DNS algorithm, the attention is directly focused on each product of matrix elements, $a_{ik} \times b_{kj}$, which are exactly $n^3$ operations between scalars in the case of multiplying square matrices of order $n$. With $n^3$ processors, each processor is in charge of a multiplication and, then, all the values obtained from each multiplication must be accurately summed in order to obtain each of the $n^2$ elements of the result matrix. More specifically, numbering the processors as if they were organized in a three-dimensions array (i.e., n×n×n processors), processor $P_{ijk}$, $0 \leq i, j, k \leq (n\text{-}1)$, is in charge of carrying out multiplication $a_{ik} \times b_{kj}$. Then, accumulating (adding) values $P_{i,j,0}...P_{i,j,n-1}$, the value $c_{ij}$ is obtained for the result matrix.

If the matrices data (elements or submatrices) distributed in processors $P_{ij0}$ are considered (i.e., that $a_{ik}$, $b_{kj}$ and $c_{ij}$ are assigned in processor $P_{ij0}$), data must be distributed/replicated before carrying out the multiplications. For this step, it is also convenient to visualize processors as they were organized in a three-dimensional array - such as Figure 2.12-a) shows- for nine processors, i.e. as successive planes (corresponding to indexes $ij$) overlapped for different values of $k$.




a) Processors Identification. b) A Columns Distribution. c) Broadcast by Planes

**Figure 2.12**: Data in DNS over an Array of 3×3×3 Processors.

From the initial assignment of matrices elements to the processor $P_{ij0}$, in the inferior plane of Figure 2.12-a), the *j*-th column of matrix A is sent to the corresponding processors (*j*-th column of processors) in the *j*-th plane, i.e. from processors $P_{ij0}$ to processors $P_{ijj}$ ($0 \le i, j \le n$-1), as Figure 2.12-b) shows. As final step in the data distribution of matrix A, the processors in each plane with the data of A, $P_{ijj}$ ($0 \le i, j \le n$-1), make a broadcast by rows of processors such as Figure 2.12-c) shows and, in this way, it is possible to have element $a_{ik}$ in each $P_{ijk}$. The way of distributing data of matrix B is similar, but instead of distributing by columns, the distribution is carried out by rows. Once all processors have the data, they can execute the multiplication and, then, the results are summed in processors $P_{ij0}$ ...$P_{ijn-1}$, such as it was previously mentioned.

The most relevent characteristics of this method to compute matrix multiplication can be summarized as follows:

- It can multiply matrices of order *n* in $O(\log(n))$ steps.
- It uses (up to) $n^3$ processors, and though it is explained for matrix elements, it can be applied to submatrices or blocks.
- A same datum of each matrix A is used and, thus, copied in all the processors' row, such as Figure 2.12-c) shows; and, similarly, a same data of matrix B is copied in all the processors' column and, thus, several replicated data are obtained.
- It is clearly oriented to parallel computers with interconnected processors as if they were in a three-dimensional structure and, in fact, this produces the reduction in the computing quantity to $O(\log(n))$ and, at the same time, data replication in several processors.

These characteristics are similar to those of the matrix multiplication method over a mesh of processors trees [82] [98]. In fact, this method also follows the sequence:

- Initial assignment of $n^2$ elements of each matrix to $n^2$ processors or processing elements.
- Communication/replication of matrices elements so that each processor has the necessary data for making the multiplications $a_{ik} \times b_{kj}$.
- Addition of the multiplication results in order to obtain the matrix C elements.

## 2.5 Chapter Summary

The most important issues that have been introduced and/or explained in this chapter, apart from the very definition of matrix multiplication and the number of operations needed for its resolution, are:

- The identification of the linear algebra applications context, given by the LAPACK library, the basic operations included in BLAS, and, more specifically, the relation of matrix multiplication. Not only is the matrix multiplication similar to the rest of L3 BLAS operations -in terms of computing and memory requirements-, but also it has been demonstrated that all L3 BLAS operations can be implemented using matrix multiplication.
- The use of matrix multiplication as bechmark. Even though its validity is limited, we can indeed assert that it represents a good benchmark of the whole L3 BLAS. In addition, it is used in order to show the optimization quality that can be attained when the performance obtained can be related to the theoretical maximum of the hardware used.
- Parallel algorithms for carrying out matrix multiplication in multiprocessors have been described. In general, they are really simple both in terms of description and their theoretical analysis of performance.
- Parallel algorithms for matrix multiplication in multicomputers have also been described. Normally, we can clearly identify the relation of each algorithm with the underlying processors interconnection (in which the algorithm obtains its best performance). The adaptations of Fox's algorithm to static and bidimensional (grids and tori) processors interconnection networks in particular have also been described (the bidimensional *adapted algorithm* algorithm is the one implemented in the ScaLAPACK library)
- All the parallel algorithms share common characteristics relatively important at a conceptual level:
  - They adopt the SPMD processing model.
  - They assume that processing nodes are homogeneous and take this homogeneity as an advantage to obtain load balance.
  - They make use of processing by blocks in order to optimize the memory hierarchy of computers, which are specifically oriented to optimization in the access to data in cache memory/ies.

Each of these points should be reviewed in order to be able to analyze, at least, whether the parallel computing algorithms for multiplying matrices are suitable or not. Specifically, this step should be carried out after making the description of the parallel processing hardware provided in local networks.