ALGORITHMIC REDISTRIBUTION METHODS FOR BLOCK CYCLIC DECOMPOSITIONS

A Dissertation

Presented for the

Doctor of Philosophy Degree

The University of Tennessee, Knoxville

Antoine Petitet

December 1996

To my parents

Acknowledgments

The writer expresses gratitude and appreciation to the members of his dissertation committee, Michael Berry, Charles Collins, Jack Dongarra, Mark Jones and David Walker for their encouragement and participation throughout my doctoral experience.

Special appreciation is due to Professor Jack Dongarra, Chairman, who provided sound guidance, support and appropriate commentaries during the course of my graduate study.

I also would like to thank Yves Robert and R. Clint Whaley for many useful and instructive discussions on general parallel algorithms and message passing software libraries. Many valuable comments for improving the presentation of this document were received from L. Susan Blackford. Finally, I am grateful to the Department of Computer Science at the University of Tennessee for allowing me to do this doctoral research work here.

A special debt of gratitude is owed to Joanne Martin, IBM POWERparallel Division, for awarding me an IBM Corporation Fellowship covering the tuition as well as a stipend for the 1994-96 academic years. This work was also supported by the Defense Advanced Research Projects Agency under contract DAAH04-95-1-0077, administered by the Army Research Office.

The author acknowledges the use of the Intel Paragon XP/S 5 computer, located in the Oak Ridge National Laboratory Center for Computational Sciences (CCS), funded by the Department of Energy's Mathematical, Information, and Computational Sciences (MICS) Division of the Office of Computational and Technology Research.

This research was also conducted using the resources of the Cornell Theory Center, which receives major funding from the National Science Foundation (NSF) and New York State, with additional support from the Advanced Research Projects Agency (ARPA), the National Center for Research Resources at the National Institutes of Health (NIH), IBM Corporation, and other members of the center's Corporate Partnership Program.

Abstract

This research aims at creating and providing a framework to describe algorithmic redistribution methods for various block cyclic decompositions. To do so properties of this data distribution scheme are formally exhibited. The examination of a number of basic dense linear algebra operations illustrates the application of those properties. This study analyzes the extent to which the general two-dimensional block cyclic data distribution allows for the expression of efficient as well as flexible matrix operations. This study also quantifies theoretically and practically how much of the efficiency of optimal block cyclic data layouts can be maintained.

The general block cyclic decomposition scheme is shown to allow for the expression of flexible basic matrix operations with little impact on the performance and efficiency delivered by optimal and restricted kernels available today. Second, block cyclic data layouts, such as the purely scattered distribution, which seem less promising as far as performance is concerned, are shown to be able to achieve optimal performance and efficiency for a given set of matrix operations. Consequently, this research not only demonstrates that the restrictions imposed by the optimal block cyclic data layouts can be alleviated, but also that efficiency and flexibility are not antagonistic features of the block cyclic mappings. These results are particularly relevant to the design of dense linear algebra software libraries as well as to data parallel compiler technology.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Il y a trois savoirs, le savoir proprement dit, le savoir-vivre et le savoir-faire.

Les deux derniers dispensent assez bien du premier.

Charles Maurice de Talleyrand (1754-1838)

In the past several years, the emergence of Distributed Memory Concurrent Computers (DMCCs) and their potential for the numerical solution of Grand Challenge problems [28, 62, 76, 77] has led to extensive research. As a result, DMCCs have become not only indispensable machines for large-scale engineering and scientific applications, but also common and viable platforms for commercial and financial applications. Many DMCCs, such as the IBM Scalable POWERparallel SP-2, the Intel Paragon, the Cray T3D, the nCube-2/3, Networks and Clusters of Workstations (NoWs and CoWs) have achieved *scalable* performance in these domains.

These scalable parallel computers comprise an ensemble of Processing Units (PUs) where each unit consists of a processor, local memories organized in a hierarchical manner, and other supporting devices. These PUs are interconnected by a point-to-point (direct) or switch-based (indirect) network. Without modifying the basic machine architecture, these distributed memory systems are capable of proportional increases in performance as the number of PUs, their memory capacity and bandwidth, and the network and I/O bandwidth are increased. As of today, DMCCs are still being produced and their success is apparent when considering how common they have become. Still, their limitations have been revealed and their successors have already appeared. The latter are constructed from a small number of nodes, where each node is a small DMCC featuring a virtual shared memory. These nodes are interconnected by a simple bus- or crossbar-based interconnection network. Programming these machines as well as their production is facilitated by the relative simplicity of the interconnection network. In addition, increasing the computational capabilities of the PUs appears to be an easier task than increasing the performance of the network. As opposed to large scale DMCCs where all processors are much less powerful than the whole, the collection of nodes of this hierarchical architecture is only slightly more powerful than its components. The SGI SMP Power Challenge is an existing example of such an architecture. The scalability of these machines can simultaneously take advantage of the progresses made by the processor and network technologies as well as the hardware and/or software mechanisms implementing the virtual shared memory. It is still unclear how these machines will be programmed. Whether these machines will in the future completely replace DMCCs is also a question that is difficult to answer today. In this dissertation, these machines will also be considered as DMCCs.

In order to fully exploit the increasing computational power of DMCCs, the application software must be *scalable*, that is, able to take advantage of larger machine configurations to solve larger problems with the same efficiency. Thus, the design of parallel algorithms and their software implementations should at the very early stages plan for larger, maybe not even existing, hardware platforms. These concerns usually cause the complexity of this software to become an important limiting factor and constraint. Indeed, the application software should also be "easy to produce and maintain". Ideally, one would like to automatically produce a parallel scalable executable from an existing serial program. In reality, programming DMCCs has been a major challenge impeding the greater success of these systems. In order to alleviate this difficulty, *parallel programming models* have been specifically designed for DMCCs. A programming model is a collection of program abstractions providing a programmer with a simplified and transparent view of the computer hardware/software system [58]. The basic computational units in a running parallel program are *processes* corresponding to operations performed by related code segments on the process's data set. A running program can then be defined as a collection of processes [58]. Inter-process communication defines what is called a running *parallel program*. In general there may be several processes executed by one physical processor; therefore, without loss of generality, the underlying DMCC will be regarded henceforth as a set of processes rather than physical processors.

This dissertation is primarily focused on two parallel programming models, the message passing model and the data parallel model. In the message passing model, two processes may communicate with each other by passing messages through the interconnection network. This model is usually based on the send and receive paradigm that requires matching operations by sender and receiver. Such a semantic is often referred to as a two-sided communication. It has been observed, however, that the coding of some applications can be facilitated when using a one-sided communication semantic. Remote memory access allows one process to specify all communication parameters, both for the sending side and the receiving side. Such a semantic is usually based on the put and get paradigm. The one-sided communication primitives can be implemented in terms of asynchronous send and receive primitives as it is suggested in the current draft of the extensions to the Message-Passing Interface [44]. Independently from the semantic of the message passing model, this programming paradigm is tedious, time-consuming, and error-prone for programmers in general, as it is ultimately based on separate name spaces. In the data parallel model, parallelism, i.e., inter-process communication, is explicitly handled by hardware synchronization and flow control. Data parallelism can be implemented on DMCCs; however, because the communication delay caused by message passing is much longer than that caused by accessing shared variables in a common memory, DMCCs are considered to be loosely-coupled multiprocessors. In order to avoid global synchronization after each instructions, the same program can be executed by each processor asynchronously. Synchronization takes place only when processors need to exchange data. This programming model is referred to as the Single Program Multiple Data (SPMD) programming model [67]. This model is based on distinct name spaces and loosely synchronous parallel computation with a distinct data set for each process. Thus, data parallelism can be exploited on DMCCs by using the data parallel programming model or the SPMD programming model. Finally, the data parallel programming model requires a regular distribution of the data as well as the tasks to be performed concurrently. These requirements considerably facilitate the design of data parallel languages. In practice, such a language can be directly derived from standard serial programming languages such as C or Fortran. It follows that data parallel programs are easier to write and debug. However, when the problem's data or even the tasks to be performed are irregular, the data parallel programming model may not be a viable and/or useful abstraction.

In this dissertation, DMCCs will be regarded as MIMD computers from the architectural point of view according to Flynn's classification [42]. DMCCs will however be considered as SPMD multicomputers from the programming point of view; that is, the same program is executed by all of the processes simultaneously. This parallel program operates on multiple data streams, and more precisely each process of the parallel program operates on its own data set. Finally, the computations performed by these processes are loosely coupled.

1.1 The Parallel Programming Problem : A Concrete Example

The tremendous commercial success of micro (personal) computing technology can be attributed to a large extent to the early development and availability of spreadsheet software products (VisiCalc). Indeed, preparing such worksheets is a very common task in nearly all businesses. Likewise, to a lesser extent however, the development of numerical linear algebra software has played a similar role for the scientific supercomputing community, since linear algebra – in particular, the solution of linear systems of equations – lies at the heart of a very large number of calculations in scientific computing. The well-known BLAS [36, 35] and LAPACK [5] numerical linear algebra software libraries are typical examples of such useful and successful software packages for shared-memory vector and parallel processors.

The programming languages used to encode VisiCalc and LAPACK respectively, have ultimately been essential building tools for the existence and success of these software packages. Indeed, in a serial model of computation the translation of basic linear algebra expressions into a procedural programming language is a well understood and relatively easy operation. The basic data structures and flow of control constructs available in such programming languages generally match the concise mathematical notation reasonably well. Consequently, one can develop *general, flexible and re-usable* numerical software in a reasonable amount of time concentrating on its design, quality and efficiency.

In a distributed memory computational environment these basic algebra expressions become *meta-expressions*. The simple global mathematical notation does not adequately describe the actual operations that the individual processes must perform. The differences between the translation of a concise mathematical formula in a serial, versus distributed, computational environment cannot solely be reduced to the addition of a few message exchanges across the interconnection network. The local operations differ to a large extent too, and can be best illustrated through the use of a simple example.

Let us consider the computation of the trace of a matrix. This operation is trivially expressed in a serial model of computation using a simple loop construct available in most programming languages. However, in a distributed memory computational environment, because the data is distributed among the processes memories, difficulty arises in locating the diagonal entries of the matrix owned by each process, rather than in combining or computing the local results.

In the last decade three main approaches to designing dense linear algebra libraries for DMCCs computers have been followed. These three approaches are presented below and focus on different optimality criteria.

1. optimal data layout and efficiency,
2. data decomposition independence,
3. software reuse via high-level language support for data parallel programming.

First, because the data decomposition largely determines the performance and scalability of a concurrent algorithm [21, 46, 48], a great deal of research [7, 12, 55] has aimed at determining optimal data distributions [27, 54, 57, 72]. This approach tacitly makes two assumptions worth reiterating about the user's data and the target architecture. First, the user's data may need to be redistributed to match this optimal distribution [6]. Second, the target architecture is such that all processes can be treated equally in terms of local performance, and, the communication rate between two processes is independent of the processes considered. Efficiency is the primary consideration justifying any restriction or requirement that an implementation may have. As a result, the two-dimensional block cyclic distribution [67] (see Chapter 2) has been suggested as the basic decomposition for parallel dense linear algebra libraries due to its scalability [38, 67], load balance and communication [54] properties. Let us illustrate the implications of this approach on the simple trace computation example. First, one would likely restrict the trace computation to the leading submatrix of the initial distributed matrix to simplify somewhat the index computations. Second, the usual and simple trace computation algorithm clearly suggests viable data decomposition decisions for its efficient parallel implementation. Therefore, one could for example require either a square block cyclic data layout onto a rectangular process grid such that the numbers of process rows and columns are relatively prime, or perhaps the use of a two-dimensional block Hankel wrapped storage scheme without any restrictions on the process grid. A definition of the block Hankel wrapped distribution scheme can be found in [55] and references therein. For sufficiently small distribution blocking factors, both of these distribution choices ensure that the diagonal blocks of the submatrix operand will be evenly distributed among all processes, and thus a perfect load balance. With these restrictions, one is guaranteed to produce an optimal trace computation implementation. Obviously, the blocking factor used by the distribution would affect the performance of such an implementation, and the optimal value of this factor is also likely to be machine dependent. Nevertheless, such an optimal value exists for each possible target architecture.

Another approach focused on flexible and general-purpose library routines. Determining an appropriate decomposition that maximizes program performance is inherently difficult due to the very large number of distribution and alignment possibilities. The above example is a good illustration of the distribution choice dilemma, since the block cyclic or the block Hankel wrapped storage schemes are indeed large families of distributions. As a matter of fact, the problem of determining an optimal data distribution for one- or two-dimensional arrays has been proven to be NP-complete [73]. Similarly, the problem of finding a set of alignments for the indices of multiple program arrays that minimizes data movement among processes is also NP-complete [71]. A number of heuristics for determining suitable distributions and alignments [71] have been proposed in the literature. In addition to these theoretical results, it is intuitively clear that a unique data distribution or alignment selection for an entire program may not be enough to achieve the best possible performance. In other words, a particular data decomposition that is well suited for one phase of a given algorithm may not be as good, as far as performance is concerned, for the other phases. These results have motivated this second approach where the user's decomposition is generally not changed but passed as an argument and a suboptimal algorithm is used. This approach is usually referred to as decomposition independent [41, 84]. The suboptimality of a routine must be weighted against the possibly large cost of redistributing the input data.

Finally, the most potentially ambitious approach attempts to provide high-level language support for data parallel programming. In the last few years, several data parallel Fortran languages have been proposed, such as Fortran D [45] and Vienna Fortran [88]. More recently, the High Performance Fortran (HPF) language [66] has been developed as a community effort to standardize data parallel Fortran programming for DMCCs. HPF includes many of the concepts originally proposed in

Fortran D, Vienna Fortran, and other data parallel Fortran languages. HPF supports an abstract model of parallel programming in which users annotate single-threaded program with data-alignment and distribution directives. The compiler uses these directives to partition the program's computation and distribute the data as the basis to derive a SPMD program to be executed on each PU of the parallel machine. Today, the first HPF compilers are slowly becoming available on a wide range of machines, and it is unclear yet if those compilers will fulfill their highly difficult goals in the near future. C-based data parallel extensions have also been proposed, such as Data Parallel C [51] and pC++ [11]. The long term objective here is to design a data parallel language to generate efficient parallel code from any serial code fragment.

1.2 Motivation

There are multiple sources of redistribution in a data parallel program, even when it is not *explicitly* invoked by the programmer or a requirement imposed by a parallel software library. For instance when the distribution or alignment of actual parameters does not match the distribution of the dummy arguments in the subprogram interface, an *implicit* redistribution or realignment phase must take place at the procedure boundary. Moreover, even if the actual parameters are distributed accordingly to the requirements imposed by a subprogram interface, this subprogram faces a dilemma. It can either use the physical distribution characteristics of its parameters as a guideline to sequence the computation and communication phases. Or, it may choose to reorganize logically and physically these phases for efficiency reasons. Substantial performance gains may be achieved by changing array data decompositions, but only if the overhead of such operations remains relatively small. Therefore, the efficient and scalable implementation of the *explicit*, *implicit* and *algorithmic* data redistribution mechanisms is important to the overall performance of data parallel programs on DMCCs. Equally important is the ability to avoid a redistribution phase that would lead to a degradation in performance.

Many issues need to be considered to design efficient and scalable data redistribution operations. First, because redistribution is a communication dominant task, the efficiency of the selected communication patterns is essential. In addition, the redistribution may have to occur within a larger context. This will be the case for example if this operation is part of a larger task taking advantage of pipelined communication phases. It is then important to redistribute in the same pipelined fashion whenever possible. Second, the total execution time of the operation is rapidly proportional to the amount of data communicated. Consequently, fast local indexing, packing and sorting techniques will benefit redistribution performance. Third, the scalability of these redistribution operations must be quantified in order to either evaluate their performance as a function of the amount of data and/or the number of processors or compare different techniques. Finally, redistribution mechanisms should be as independent as possible from a particular architecture to allow for their portability across a wide range of DMCCs. This last issue has been considerably simplified by the recent development of the Message Passing Interface (MPI) [43] and its adoption by a large majority of machine vendors.

Most of the Fortran- and C-based data parallel languages incorporate *explicit* and *implicit* redistribution capabilities. The Kali language [75] was one of the first to do so. DINO [79] addresses the implicit redistribution of data at procedure boundaries. The Hypertasking compiler [8] for data parallel C programs, Vienna Fortran [14] and Fortran D [50] additionally specify data redistribution primitives. For efficiency purposes as well as simplicity of the compiler, these redistribution operations are often implemented in a library of intrinsics [61].

Explicit and implicit data redistribution operations are necessary but not quite sufficient. These expensive operations change the distribution of an entire operand at once, where in a number of cases some of it can be delayed. In those cases, only a more simple redistribution operation on a suboperand is required which can be overlapped with other computational and/or communication phases. Such operations are more efficient in terms of space and time. They are called *algorithmic* redistribution methods because they in essence attempt to reorganize logically and physically the computations and communications within a algorithmic context.

To derive algorithmically redistributed operations, it is first necessary to analyze the reasons why the formulation of more general distributed operations is difficult for a given data decomposition. Such difficulties are intertwined with the given data layout, even though one can usually define this underlying data-process mapping in a simple form. Indeed, this definition is by itself of little use unless it is accompanied by a list of the mapping's properties. For instance being able to tell in which process's memory the matrix entry a_{44} resides is undoubtly a valuable piece of information; it is, however, far more useful to be able to determine in which process's memories the diagonal blocks reside and how far those are from each other. Indeed, if one wants to access the diagonal entries of a matrix, one must first know how to access a_{11}, and second, how to access $a_{i+1,i+1}$ from a_{ii}. The formulation of general distributed matrix operations must be derived from the properties that a given data layout may possess as opposed to the data layout itself. These inherent properties should therefore be brought to the fore. One aspect of this dissertation is to exhibit these properties along with a formal proof as well as illustrate how they characterize a given data distribution scheme. Algorithmic redistribution methods are then suggested for the general two-dimensional block cyclic distribution.

The two-dimensional block cyclic data decomposition scheme is a natural candidate for such a study for multiple reasons. First, it provides a simple general purpose way of distributing a block-partitioned matrix on DMCCs. It encompasses a large number of more specific data distributions such as the blocked or purely scattered cases (see Chapter 2), and it has been incorporated into the High Performance Fortran language [66]. Second, some of this mapping's properties can be expressed in a simpler way if one restricts oneself to the square block cyclic case (see Chapter 2). Some of these simplified corollaries have already been indirectly illustrated and applied in [22, 23, 25]. Laborious debugging sessions of code fragments that were in fact relying on these corollaries are at the origin of the development of this more formal approach. Finally, the way in which a matrix is distributed over a set of processes has a major impact on the load balance and communication characteristics of the concurrent algorithm, and hence largely determines its performance and scalability. There is considerable evidence that the square block cyclic mapping can lead to very efficient implementations of more complex matrix operations such as the LU, Cholesky or QR factorizations [20, 38] or even the reductions to Hessenberg, tridiagonal and bidiagonal forms [19, 24].

The encouraging performance results mentioned above were obtained for particular process grid shapes and empirically chosen distribution parameters on specific hardware platforms. It is natural to ask if restricting the supported data layouts and providing basic operations with little flexibility are reasonable decisions, even at the early design stages of a general purpose dense linear algebra software library for DMCCs. Indeed, as the development progresses, these restrictions become more burdensome, eventually to the point where they prevent from the formulation of more complicated algorithms. Examples of such algorithms are out of core linear solvers, divide and conquer methods and algorithms involving a large number of distributed matrices such as the generalized least squares solvers, or even direct sparse linear solvers.

On the one hand, supporting the most general block cyclic decompositions will not allow for any performance increase or decrease as far as the restricted and optimal cases are concerned. On the other hand, allowing for more flexible operations suggests a more intensive use of algorithmic blocking features, which somewhat attenuate the communication overhead induced by the most general data layouts. Consequently, algorithmic redistribution methods logically balance the communication and computation operations, and, therefore, allow for improved transportable performance. This dissertation quantifies and models these effects and discusses the possible tradeoffs between efficiency and flexibility. Experimental results are also reported to evaluate the performance model.

1.3 Problem Statement

Basic algorithmically redistributed matrix operations on distributed memory architectures allow for the expression of efficient and flexible general matrix operations for various block cyclic mappings. This dissertation first investigates distinct logical blocking techniques, as well as their impact on the scalability of these opera-

tions, and second to what extend flexibility and efficiency are antagonistic features for the general family of two-dimensional block cyclic data distributions.

This dissertation is distinguished as the earliest known research to propose a portable and scalable set of flexible algorithmically redistributed operations, as well as a framework for expressing these complicated operations in a modular fashion. Their scalability is quantified on distributed memory platforms for various block cyclic mappings.

1.4 Organization of This Dissertation

Chapter 2 defines the two-dimensional block-cyclic data distribution. Elementary results of the theory of integers are systematically brought to the fore. They fundamentally characterize the properties of the two-dimensional block cyclic data distribution. In addition, these properties are the basis of efficient algorithms for address generation, fast indexing techniques and consequently efficient data redistribution and manipulation. Some of these algorithms are described in detail along with the properties from which they are deduced. Related work is also summarized. Chapter 3 presents various flexible and general basic algorithmic redistribution operations. Different blocking techniques particularly well-suited for the implementation of these basic dense linear algebra operations on DMCCs are presented and discussed in detail. In addition to the easier case of general rectangular matrix operands, specific general blocking techniques for triangular and symmetric matrices are shown. A *minimal* amount of data is exchanged among process memories during these redistribution operations. These techniques feature a variable logical blocking factor for efficiency purposes and are independent of the underlying machine architecture. The properties shown earlier ensure the portability of these techniques among distributed memory platforms. The optimality of these techniques with respect to minimizing the amount of data exchanged is shown in Chapter 4, along with a discussion of the importance of the variable logical blocking factor. Chapter 4 also presents a framework for *quantifying* the scalability of the algorithmic redistribution operations previously presented. This framework is also used to assess the theoretical performance impact of the logical blocking factor. This parameter is shown to allow for high performance tuning and its theoretical relationship with some machine parameters is exhibited. Chapter 5 validates the previously established performance model by comparing its theoretical predictions with actual and experimental performance data. Chapter 6 concludes this dissertation by explaining how algorithmically redistributed operations can be used in the context of even more complex linear algebra computations, such as matrix transposition, matrix-matrix multiplication, triangular solve, classic matrix factorizations and reductions. This last chapter finally summarizes the major contributions of this dissertation and suggests potential future research directions.

Chapter 2

Properties of The Block Cyclic Data Distribution

2.1 Introduction

Due to the non-uniform memory access time of distributed memory concurrent computers, the performance of data parallel programs is highly sensitive to the adopted data decomposition scheme. The problem of determining an appropriate data decomposition is to maximize system performance by balancing the computational load among processors and by minimizing the local and remote memory traffic. The data decomposition problem involves *data distribution*, which deals with how data arrays should be distributed among processor memories, and *data alignment*, which specifies the collocation of data arrays. Since the data decomposition largely determines the performance and scalability of a concurrent algorithm, a great deal of research [21, 46, 48, 55] has aimed at studying different data decompositions [7, 12, 57]. As a result, the two-dimensional block cyclic distribution [67] has been suggested as a possible general purpose basic decomposition for parallel dense linear algebra libraries [27, 54, 72] due to its scalability [38], load balance and communication [54] properties.

The purpose of this chapter is to present and define the two-dimensional block cyclic data distribution. The contributions of this chapter are two-fold. First, elementary results of the theory of integers are systematically brought to the fore. They fundamentally characterize the properties of the two-dimensional block cyclic data distribution. Second, these properties are the basis of efficient algorithms for address generation and fast indexing techniques, leading to consequently efficient data redistribution and manipulation. Some of these algorithms are described in detail along with the properties from which they are deduced.

The two-dimensional block cyclic data distribution or decomposition is formally defined. Its fundamental properties are then formally proved and presented along with direct applications. The next two chapters illustrate how these properties can be applied to address and solve data alignment problems, i.e., to generate and implement more complicated algorithms for data redistribution and logically blocked operations. The correctness of these operations and the robustness of their implementation rely on these properties.

2.2 Definitions

Definition 2.2.1 The mapping of an algorithm's data over the processes of a distributed memory concurrent computer is called a data distribution. The block-cyclic data distribution is one of these mappings.

In general there may be several processes executed by one processor, therefore, without loss of generality, the underlying concurrent computer is regarded as a set of *processes,* rather than physical processors. Consider a $P \times Q$ grid of processes, and let , denote the set of all the process coordinates (p, q) in this grid:

$$, \ = \{(p, q) \in \{0 \ldots P \Leftrightarrow 1\} \times \{0 \ldots Q \Leftrightarrow 1\}\}.$$

Figure 2.1 illustrates a 2×3 process grid and the elements of , .

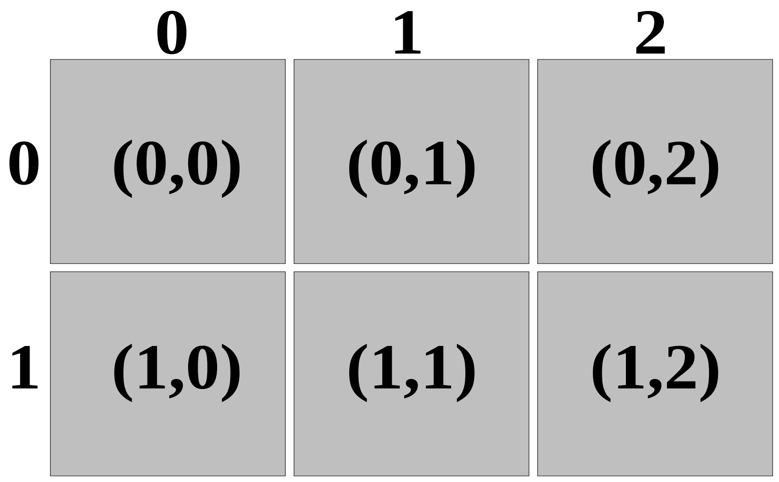


Figure 2.1: A 2×3 process grid

Consider an $M_b \times N_b$ array of blocks. Each block is uniquely identified by the integer pair (i_b, j_b) of its row and column indexes. Let Δ_b be the set constructed from all these pairs:

$$\begin{aligned} \Delta_b = \quad & \{(i_b, j_b) \in \{0 \ldots M_b \Leftrightarrow 1\} \times \{0 \ldots N_b \Leftrightarrow 1\}\} \\ = \quad & \{(l\ P + p, m\ Q + q) \text{ with } ((p, q), (l, m)) \in \ , \ \times \Lambda\} \end{aligned}$$

with $\Lambda = \{(l, m) \in \{0 \ldots \lfloor \frac{M_b - 1}{P} \rfloor\} \times \{0 \ldots \lfloor \frac{N_b - 1}{Q} \rfloor\}$.

Definition 2.2.2 The block cyclic distribution is a mapping of Δ_b onto , associating to block coordinates the coordinates of the process into which it resides:

$$\left\{ \begin{array}{c} \Delta_b \Leftrightarrow\!\rightarrow \ , \\ (i_b, j_b) = (l\ P + p, m\ Q + q) \Leftrightarrow\!\!\rightarrow (p, q). \end{array} \right. \qquad (2.2.1)$$

An $M \times N$ matrix A partitioned into blocks of size $r \times s$ is an $M_b \times N_b$ array of blocks. The total number M_b (respectively N_b) of row blocks or blocks of rows (respectively column blocks) of A as well as their size are easy to determine:

$$M_b = \frac{M \Leftrightarrow 1}{r} + 1 = \lceil \frac{M}{r} \rceil \text{ and } N_b = \frac{N \Leftrightarrow 1}{s} + 1 = \lceil \frac{N}{s} \rceil \text{ with } M, N \geq 1. \quad (2.2.2)$$

All the blocks are of size $r \times s$ with the exception of the ones of the last row and column of blocks. If $M \bmod r = 0$, the last row blocks contain the last r matrix rows, and the last $M \bmod r$ rows otherwise, where mod denotes the positive modulo operator of two positive integers. The last column blocks contain the last s matrix columns if N mod $s = 0$, and the last N mod s columns otherwise. An example of a block-partitioned matrix is shown in Figure 2.2. Figure 2.3 illustrates the mapping of this example onto a particular process grid.

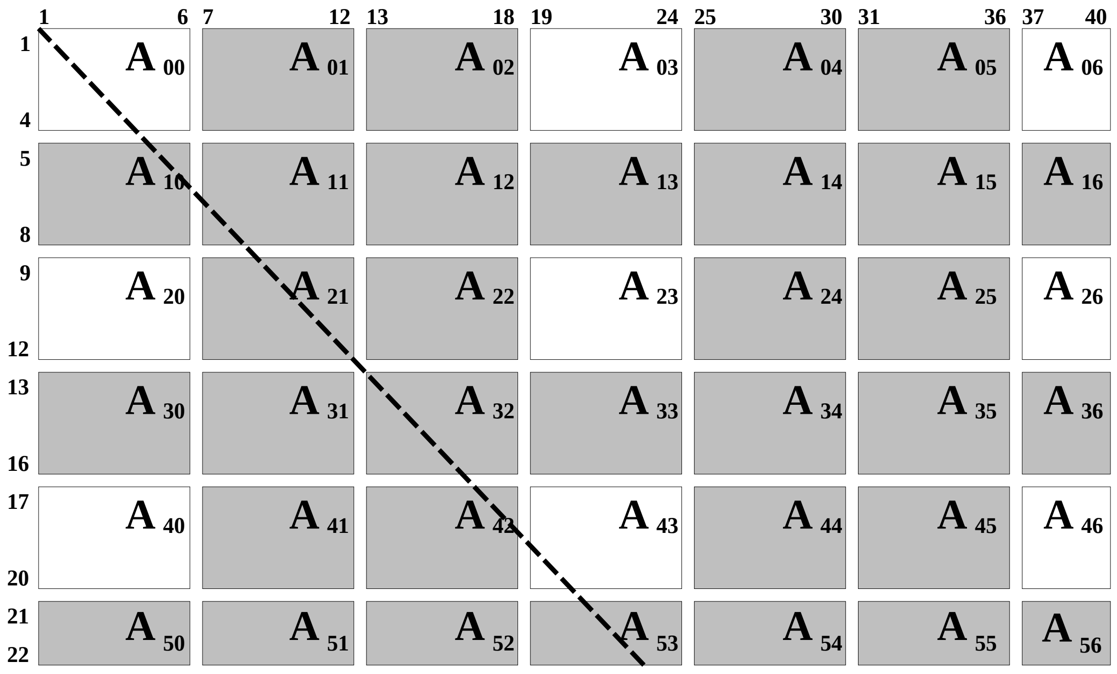
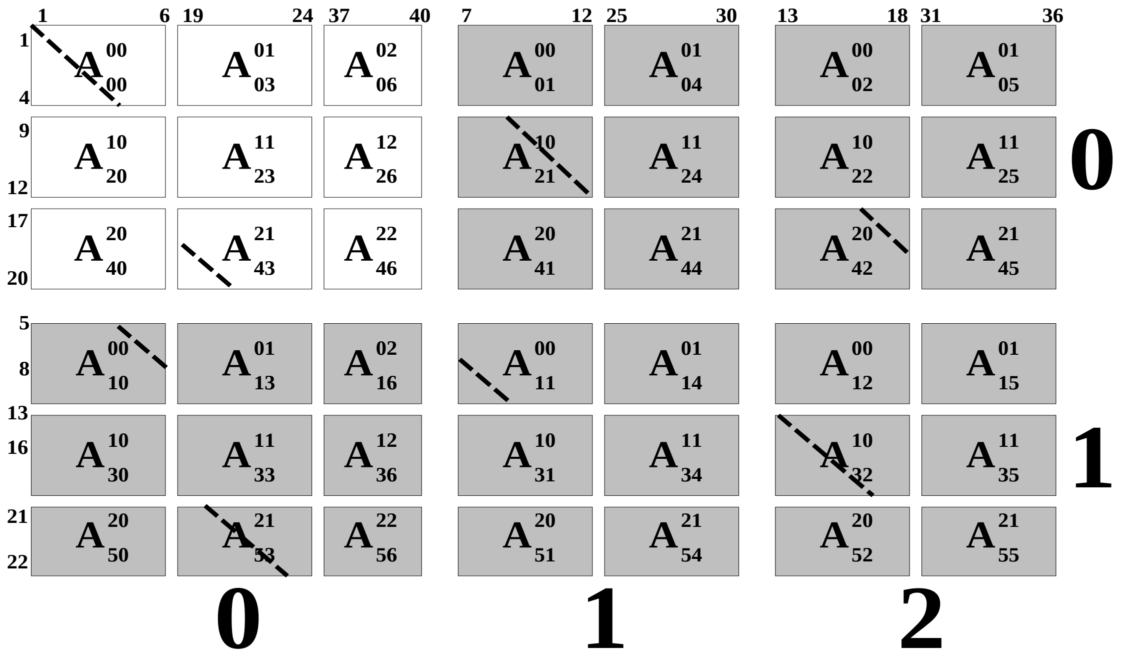


Figure 2.2: A block-partitioned matrix, with $M = 22$, $N = 40$, $r = 4$, $s = 6$.

Definition 2.2.3 Let Λ be the set of all possible pairs (l, m) as defined above. Consider the adjoint mapping from Δ_b onto Λ that associates to a global block coordinate pair its local coordinate pair:

$$\left\{ \begin{array}{c} \Delta_b \Leftrightarrow\!\!\!\rightarrow \Lambda \\ (i_b, j_b) = (l\ P + p, m\ Q + q) \Leftrightarrow\!\!\!\rightarrow (l, m) \end{array} \right. \tag{2.2.3}$$

REMARK. Mapping (2.2.3) transforms the coordinates of a matrix block into local values, i.e., the matrix block of coordinates $(i_b, j_b) = (l\ P + p, m\ Q + q)$ is the local block indexed by (l, m) into the process (p, q). This fact is illustrated in Figure 2.3, where $A^{lm}_{i_b j_b}$ denotes a matrix block of global coordinates (i_b, j_b) and local coordinates (l, m). At this point, it is useful to refine the above Definitions (2.2.1) and (2.2.3).



1 6 19 24 37 40 7 12 25 30 13 18 31 36
1 4 9 12 17 20 5 8 13 16 21 22
A 00 00, A 01 03, A 02 06, A 00 01, A 01 04, A 00 02, A 01 05
A 10 20, A 11 23, A 12 26, A 10 21, A 11 24, A 10 22, A 11 25
A 20 40, A 21 43, A 22 46, A 20 41, A 21 44, A 20 42, A 21 45
A 00 10, A 01 13, A 02 16, A 00 11, A 01 14, A 00 12, A 01 15
A 10 30, A 11 33, A 12 36, A 10 31, A 11 34, A 10 32, A 11 35
A 20 50, A 21 53, A 22 56, A 20 51, A 21 54, A 20 52, A 21 55
0
1
0
1
2

Figure 2.3: The previous block-partitioned matrix mapped onto a 2×3 process grid.

Definition 2.2.4 Consider Δ_b, , and Λ as defined above. The block cyclic distribution is defined by the following two related mappings associating to the global block coordinates:

- the coordinates of the process in which this block resides

$$\begin{cases} \Delta_b \Leftrightarrow\!\!\rightarrow \;, \\ (i_b, j_b) = (l\ P + p, m\ Q + q) \Leftrightarrow\!\!\rightarrow (p, q) \end{cases} \tag{2.2.4}$$

- the corresponding local coordinates of this block

$$\begin{cases} \Delta_b \Leftrightarrow\!\!\rightarrow \Lambda \\ (i_b, j_b) = (l\ P + p, m\ Q + q) \Leftrightarrow\!\!\rightarrow (l, m). \end{cases} \tag{2.2.5}$$

Furthermore, this previous definition can be restated in terms of each matrix entry a_{ij} instead of the block $A_{i_b j_b}$ to which it belongs. This definition is the most appropriate and will best serve our purpose:

Definition 2.2.5 Consider a $P \times Q$ grid of processes, where , denotes the set of all process coordinates (p, q) in this grid:

$$, \ = \{(p, q) \in \{0 \ldots P \Leftrightarrow 1\} \times \{0 \ldots Q \Leftrightarrow 1\}\}.$$

Consider an $M \times N$ matrix partitioned into blocks of size $r \times s$. Each matrix entry a_{ij} is uniquely identified by the integer pair (i, j) of its row and column indexes. Let Δ be the set constructed from all these pairs:

$$\begin{aligned} \Delta = \;& \{(i, j) \in \{0 \ldots M \Leftrightarrow 1\} \times \{0 \ldots N \Leftrightarrow 1\}\} \\ = \;& \{((l\ P + p)r + x, (m\ Q + q)s + y), ((p, q), (l, m), (x, y)) \in \ , \ \times \Lambda \times \Theta\} \end{aligned}$$

$$\text{with } \Lambda = \{(l,m) \in \{0 \ldots \lfloor \frac{\frac{M-1}{r}}{P} \rfloor\} \times \{0 \ldots \lfloor \frac{\frac{N-1}{s}}{Q} \rfloor\}\} \text{ and}$$

$$\Theta = \{(x,y) \in \{0 \ldots r \Leftrightarrow 1\} \times \{0 \ldots s \Leftrightarrow 1\}.$$

The block cyclic distribution is then defined by the three following mappings associating to a matrix entry index pair (i,j):

- the coordinates (p,q) of the process into which the matrix entry resides

$$\left\{ \begin{array}{c} \Delta \Leftrightarrow\!\rightarrow , \\ (i,j) = ((l\ P+p)\ r+x, (m\ Q+q)\ s+y) \leftarrow\!\!\!\Leftrightarrow\!\rightarrow (p,q) \end{array} \right. \tag{2.2.6}$$

- the coordinates (l,m) of the local block in which the matrix entry resides

$$\left\{ \begin{array}{c} \Delta \Leftrightarrow\!\rightarrow \Lambda \\ (i,j) = ((l\ P+p)\ r+x, (m\ Q+q)\ s+y) \leftarrow\!\!\!\Leftrightarrow\!\rightarrow (l,m) \end{array} \right. \tag{2.2.7}$$

- the local row and column offsets (x,y) within this local block (l,m)

$$\left\{ \begin{array}{c} \Delta \Leftrightarrow\!\rightarrow \Theta \\ (i,j) = ((l\ P+p)\ r+x, (m\ Q+q)\ s+y) \leftarrow\!\!\!\Leftrightarrow\!\rightarrow (x,y) \end{array} \right. \tag{2.2.8}$$

Definition 2.2.6 The blocked decomposition is defined by Definition 2.2.5 with $r = \lceil \frac{M}{P} \rceil$ and $s = \lceil \frac{N}{Q} \rceil$, i.e., $\Lambda = \{(0,0)\}$.

Definition 2.2.7 The purely scattered or cyclic decomposition is defined by Definition 2.2.5 with $r = s = 1$, i.e., $\Theta = \{(0,0)\}$.

Definition 2.2.8 The square block cyclic distribution is a special case of the general two-dimensional block cyclic distribution (2.2.5) with $r = s$.

2.3 Block Properties

In this section, we state and prove some properties of the general block cyclic data decomposition as defined in (2.2.5).

2.3.1 Notation and Elementary Theorems

The positive modulo of two positive integers a and b has been denoted above by mod. When b evenly divides a, i.e., $a \bmod b = 0$, we equivalently write b div a. The least common multiple and greatest common divisor of x and y are respectively denoted by $\operatorname{lcm}(x, y)$ and $\gcd(x, y)$. A few elementary theorems that will be used in the next sections are stated below. These theorems are direct implications of Euclid's division algorithm and their proof can be found in any elementary integer theory textbook.

Theorem 2.3.1 $\forall x, y \in \mathbb{N},\ x\ y = \operatorname{lcm}(x, y)\ \gcd(x, y)$.

Theorem 2.3.2 $\forall\ a,\ b \in \mathbb{Z}$, the set of all linear integral combinations $a\ t + b\ u$ with t and u in $\mathbb{Z}$ is exactly the set of all integral multiples of $\gcd(a, b)$. In other words, for all t and u in $\mathbb{Z}$, there exists k in $\mathbb{Z}$ such that $a\ t + b\ u = k\ \gcd(a, b)$ and conversely.

Definition 2.3.1 The least common multiple and greatest common divisor of $P\, r$ and $Q\, s$ are denoted by *lcmb* and *gcdb* respectively, i.e.,

$$lcmb \equiv \mathrm{lcm}(P\ r, Q\ s) \text{ and } gcdb \equiv \gcd(P\ r, Q\ s).$$

With these notational conventions, it follows that $P\ Q\ r\ s = lcmb\ gcdb$.

2.3.2 Properties

In order to achieve an even distribution of the load in a data parallel program, one has to first distribute evenly the data. Equivalently one must know the smallest piece of data that is evenly distributed. The size of the *smallest* $r \times s$ block-partitioned matrix evenly mapped onto a $P \times Q$ process grid according to the block cyclic scheme is $P\ r \times Q\ s$. This is trivially achieved by distributing one $r \times s$ block per process. This matrix is in general rectangular as opposed to square, and as such it is not a convenient partitioning unit for operations on triangular or symmetric matrices. A more appropriate unit size is given by *lcmb* which is the size of the *smallest* $r \times s$ block-partitioned *square* matrix similarly mapped onto the same process grid. This square matrix is called an *LCM block*. Each process owns exactly $lcmb/P \times lcmb/Q$ entries of this LCM block. This concept has been originally introduced in the restricted context of square block cyclic mappings in [22, 23, 25]. The purpose of this section is surely to formally exhibit properties of the block cyclic distribution. More importantly, this collection of properties aims at determining an elegant and convenient data structure that encapsulates and reveals the essential features of the LCM block partitioning unit when used in the context of algorithmic redistributed operations.

In preparation of the more general properties presented later in this chapter, it is useful to first characterize the properties of the individual blocks $A_{i_b j_b}$ with (i_b, j_b) in Δ_b. More precisely, the blocks $A_{i_b j_b}$ such that $i_b = j_b$ are of particular importance in the development of the more general properties of interest.

Definition 2.3.2 A matrix block $A_{i_b j_b}$ with $(i_b, j_b) = (l\ P + p, m\ Q + q)$ such that $i_b = j_b$ with $((l, m), (p, q)) \in \Lambda \times$, is called a *D-block.*

Definition 2.3.3 A process of coordinates (p, q) in , , such that there is a pair (l, m) in Λ verifying the equation: $l\ P + p = m\ Q + q$ is called a *D-process.*

Remark. A D-block or a D-process does not necessarily contain diagonal entries of the matrix, e.g., the D-block A_{33} in Figure 2.2.

Property 2.3.1 There are exactly $\mathrm{lcm}(P, Q)$ D-processes.

Proof. Consider the D-blocks A_{ii} and A_{jj}, with $i \neq j$ and $i, j \geq 0$. A_{ii} and A_{jj} reside in the memory of the process of coordinates (p, q) if and only if

$$\begin{cases} (i \Leftrightarrow p) \bmod P = 0, \\ (j \Leftrightarrow p) \bmod P = 0, \end{cases} \text{and} \begin{cases} (i \Leftrightarrow q) \bmod Q = 0, \\ (j \Leftrightarrow q) \bmod Q = 0. \end{cases} \qquad (2.3.9)$$

Since congruences for the same modulus may be added or subtracted, the previous conditions can be rewritten as

$$\left.\begin{array}{l}(i \Leftrightarrow j) \bmod P = 0 \\ \\ (i \Leftrightarrow j) \bmod Q = 0\end{array}\right\} \Leftrightarrow (i \Leftrightarrow j) \bmod \operatorname{lcm}(P, Q) = 0. \tag{2.3.10}$$

It follows that the set of pairs (A_{ii}, A_{jj}) such that A_{ii} and A_{jj} are D-blocks owned by the same process is an equivalence relation having exactly $\operatorname{lcm}(P, Q)$ equivalence classes. ∎

Remark. A fundamental consequence of this proof is that the sequence of D-blocks or D-processes is periodic and the smallest period is $\operatorname{lcm}(P, Q)$. This proves that the LCM block introduced above is indeed the *smallest* square partitioning unit. Second, all $r \times s$ blocks of relative coordinates say (r, s) with respect to the LCM block to which they belong to are residing in the same process of coordinates $(r \bmod P, s \bmod Q)$. Finally, two of these blocks are adjacent if and only if their corresponding LCM blocks are adjacent.

Property 2.3.2 The set of the D-processes of coordinates (p, q) in , is given by the following equation

$$(p \Leftrightarrow q) \bmod \gcd(P, Q) = 0, \text{ i.e., } \gcd(P, Q) \text{ div } (p \Leftrightarrow q). \tag{2.3.11}$$

Proof. The coordinates of a D-block verify $l\ P + p = m\ Q + q$, i.e.,

$$p \Leftrightarrow q = m\ Q \Leftrightarrow l\ P. \tag{2.3.12}$$

A necessary and sufficient condition for this linear Diophantine equation to have a solution in integers for l and m is that $\gcd(P, Q)$ divides $p \Leftrightarrow q$. $\blacksquare$

By setting p (or q) in Equation (2.3.11) to a constant value, it follows that the distance between two consecutive D-processes in the same process row (or column) is equal to $\gcd(P, Q)$. Moreover, the extended Euclid's algorithm [29] can be used to solve the linear Diophantine Equation (2.3.12). The solution pairs depend on the local process information (p, q). First, a particular solution (l_*, m_*) of the equation

$$\gcd(P, Q) = m_*\ Q \Leftrightarrow l_*\ P \tag{2.3.13}$$

is found by computing $\gcd(P, Q)$. The set Λ_{pq} of all solutions of the Equation (2.3.12) is given by:

$$\Lambda_{pq} = \{(l, m) = (l_* + t\ lcmp, m_* + t\ lcmq), t \in \mathbb{Z}\}, \tag{2.3.14}$$

with $lcmp = lcmb/(P\ r)$ and $lcmq = lcmb/(Q\ s)$.

Property 2.3.3 If A_{ii} is a D-block residing in the process of coordinates (p, q), the next D-block A_{kk} residing in this same process with $k = i + \mathrm{lcm}(P, Q)$ is locally separated from A_{ii} by a fixed stride in the column and row directions, namely $lcmp$ and $lcmq$ respectively.

PROOF. This is a direct conclusion of the block cyclic decomposition Definition (2.2.5) and the above definition of Λ_{pq}. ∎

Property 2.3.4 The local coordinates of the first D-block residing in the process of coordinates (p, q) are determined by the smallest positive pair $(\tilde{l}, \tilde{m})$ in Λ_{pq}. This pair also provides an ordering of the D-processes.

PROOF. By definition of the block cyclic distribution and since the Mapping (2.2.3) is an increasing function of l and m, it is sufficient to prove this result by reasoning on the matrix blocks. The set $\widetilde{\Lambda_{pq}}$ of the D-block coordinates residing in a D-process is given by

$$\widetilde{\Lambda_{pq}} = \{(l, m) \in \Lambda_{pq}, \text{such that } l, m \geq 0\} \subset I\!\!N^2. \qquad (2.3.15)$$

$\widetilde{\Lambda_{pq}}$ is a discrete set that is bounded below; therefore, it has and contains its smallest element $(\tilde{l}, \tilde{m})$. Furthermore, the D-block $(\tilde{l}\ P + p, \tilde{m}\ Q + q)$, with $\tilde{l}\ P + p = \tilde{m}\ Q + q$ is the first D-block residing in the process (p, q). This implies that this particular process is the $(\tilde{l}P + p + 1)^{th}$ D-process. ∎

Figure 2.4 shows a 4×6 process grid, the D-processes are highlighted as darker squares. The upper left and lower right corners of the process grid are labeled by A and Ā respectively. A simple graphical procedure to determine these D-processes is to draw a diagonal starting from A. This diagonal is represented by a bold dashed line on the figure. When the diagonal reaches an edge of the process grid, it should be continued on the opposite edge of the grid. For example the diagonal starting from A first reaches an edge of the grid at B. The diagonal should therefore be continued from the opposite edge of the grid precisely from

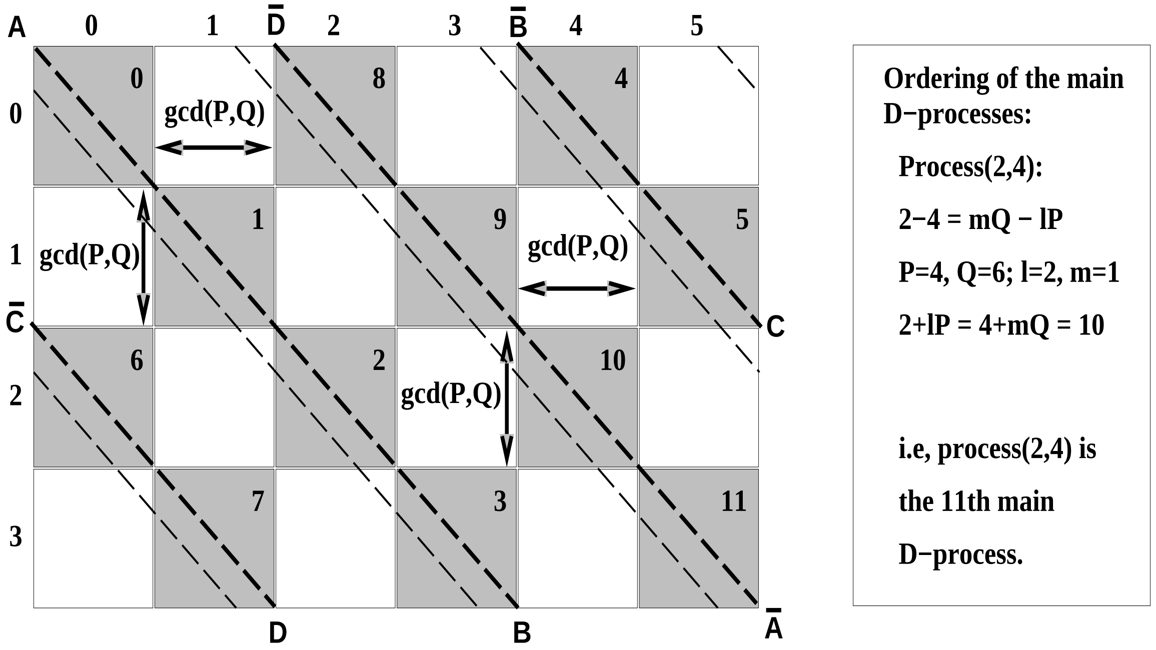


Figure 2.4: A $P \times Q$ process grid with $P = 4$, $Q = 6$, $\mathrm{lcm}(P,Q) = 12$, $\mathrm{gcd}(P,Q) = 2$.

B̄ and so on. Ultimately, the diagonal will reach Ā since the grid is finite. One may also picture the process grid folded onto a torus. In this case, A and Ā label the same point on the surface of the torus. This is also true of all other conjugate pairs. Thus, finding the D-processes can be achieved by drawing a "straight" line on a torus surface. The dashed lines represent two matrix diagonals of a square block-partitioned matrix mapped onto this grid. As noted above, the bold dashed line is a D-diagonal, i.e., in one-to-one correspondence with the D-blocks. The corresponding D-processes are represented by darker squares. The total number of D-processes owning these D-blocks is $\mathrm{lcm}(4, 6) = 12$. The distance between two consecutive D-processes is equal to $\gcd(P, Q) = \gcd(4, 6) = 2$ as explicitly noted in this figure. In addition, this example also illustrates how to determine the ordering of these D-processes by finding the smallest positive solution of the Equation (2.3.12). This computation is of importance when one wants to compute the local number of D-blocks owned by a particular process. All of these results were given by the above properties.

Figure 2.5 illustrates how to compute the local distance between D-blocks. It represents a portion of a square block-partitioned matrix distributed over a 2×3 process grid. The dashed lines materialize three matrix diagonals D0, D1 and D2 of this square block-partitioned matrix. The bold dashed diagonal D0 is a D-diagonal, i.e., in one-to-one correspondence with the D-blocks. The blocks residing in the process of coordinates (p, q) are represented by darker squares. Note that the global coordinates of the blocks residing in this process (p, q) represented on the figure have the form $((l+u)\,P+p, (m+v)\,Q+q)$ with $u = 0 \ldots 3$ and $v = 0 \ldots 2$. The blocks of global coordinates $(l\,P+p, m\,Q+q)$ and $((l+3)\,P+p, (m+2)\,Q+q)$ are labeled by B0 and B3 respectively. The figure illustrates the fact that the two

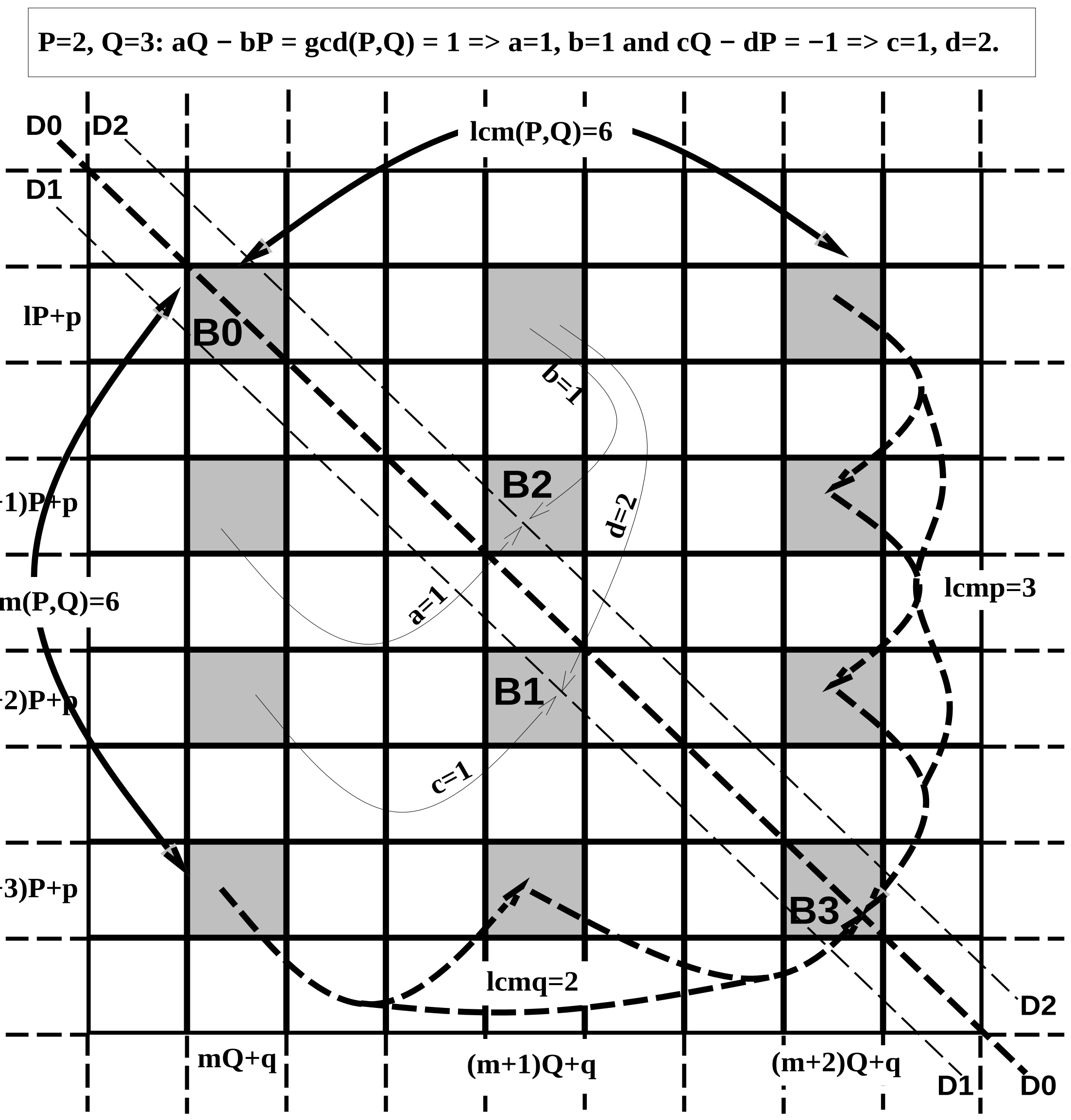


Figure 2.5: A square block-partitioned matrix distributed over a 2×3 process grid.

consecutive D-blocks B0 and B3 are globally $\text{lcm}(P, Q) = 6$ blocks away from each other in both row and column directions. This fact is indicated on the figure by the bold arrows. Locally within this process of coordinates (p, q) these same two consecutive D-blocks B0 and B3 are $lcmp = 3$ blocks in the column direction and $lcmq = 2$ blocks in the row direction distant from each other. Similarly, this fact is indicated on the figure by the bold dashed arrows. The two diagonals D1 and D2 illustrated in the figure by simple dashed lines do not not match exactly the D-blocks. For these diagonals D1 and D2, this figure shows the existence of other blocks than the D-blocks that not only reside in the process of coordinates (p, q), but also own diagonal entries. These blocks are labeled by B1 and B2. They lie between the two previous consecutive D-blocks B0 and B3. The block labeled B1 (respectively B2) owns diagonals when the diagonal D1 (respectively D2) to be considered is below (respectively above) the D-diagonal D0. It is interesting to notice that the local jumps between blocks owning diagonal entries, i.e., between the blocks B0, B1 and B3 or the blocks B0, B2 and B3, are solutions of the linear Diophantine equations $\pm \gcd(P, Q) = l\ P \Leftrightarrow m\ Q$ depending on the position of the diagonal. In the figure, the two solution pairs are denoted by (a, b) and (c, d). Simple arcs illustrate these local jumps. For example to go locally from B0 to B1, one has to go $d = 2$ blocks south, and $c = 1$ block east. Finally, to go locally from B1 to B3, one has to go $a = 1$ block south, and $b = 1$ block east. Reversing this procedure allows one to explicitly find the path from B0 to B3 via B2. All of these results were also given by the above properties.

Assume that the only problem one is interested in is to locate the diagonal entries of a block-partitioned distributed matrix. Within this context, assume that one is willing to restrict oneself to the purely scattered decomposition as defined in (2.2.7). With these assumptions, subject to a renumbering of the processes, a D-block is just a diagonal entry and conversely. Since the above properties completely characterize the D-blocks and the D-processes, the problem of interest is solved. These assumptions have been made by some researchers [12, 13, 54] to implement the LINPACK benchmark and related dense linear algebra kernels on distributed vector computers. These are also specifications data parallel languages such as HPF are leaning towards.

Assume now that one is willing to restrict oneself to the square ($r = s$) block cyclic decomposition as defined in (2.2.8) and is only interested by the D-diagonals made of matrix entries a_{ij} such that $|i \Leftrightarrow j| \bmod s = 0$. In this case as well, the diagonals of the D-blocks are the D-diagonal entries and conversely. Similarly as above, the problem of locating the diagonals is solved. The square block cyclic data decomposition is a particular case of the distributions HPF supports standardly [66]. This approach has also been chosen for the design of the dense routines in the ScaLAPACK software library [16, 21, 22, 23, 25, 38]. The above properties assume that the diagonals a_{ij} of interest are such that s evenly divides $|i \Leftrightarrow j|$. When this is not the case, the properties can easily be adapted as indicated in Table 2.1. This table shows how the above properties are generalized to all possible diagonals for the square block cyclic distribution.

Table 2.1: Generalization of the block properties to the square block cyclic distribution

$r = s$	D-Diagonals $\lvert i \Leftrightarrow j \rvert \bmod r = 0$	Other Diagonals $\lvert i \Leftrightarrow j \rvert \bmod r \neq 0$
Blocks owning diagonals	$p \Leftrightarrow q = m\ Q \Leftrightarrow l\ P$	$\begin{cases} p \Leftrightarrow q = m\ Q \Leftrightarrow l\ P \\ p \Leftrightarrow q \pm 1 = m\ Q \Leftrightarrow l\ P \end{cases}$
Processes owning diagonals	$\gcd(P,Q)$ div $(p \Leftrightarrow q)$	$\begin{cases} \gcd(P,Q) \text{ div } (p \Leftrightarrow q) \\ \gcd(P,Q) \text{ div } (p \Leftrightarrow q \pm 1) \end{cases}$
Number of such processes	$\mathrm{lcm}(P,Q)$	$\min(2, \gcd(P,Q)) \times \mathrm{lcm}(P,Q)$

In all of the other cases the previous properties are insufficient to solve the problem of locating diagonals. More powerful techniques presented in the next two sections should be used instead.

2.4 Solving Linear Diophantine Equations

The algorithm to solve linear Diophantine equations is described in this section. It is historically attributed to the greek Diophantos (perhaps A.D. 250). The method is nevertheless presented below, as it is still the best known method to solve completely these equations and also one way to establish properties of the block cyclic distribution. In order to locate the diagonals by solving directly a linear Diophantine equation, one first consider the following equation:

$$m\ Q\ s \Leftrightarrow l\ P\ r = p\ r \Leftrightarrow q\ s + x \Leftrightarrow y \tag{2.4.16}$$

for $(l, m) \in \Lambda$ and $(x, y) \in \Theta$. This equation is deduced from the block cyclic mapping defined in (2.2.5). To begin, one can instead solve

$$m_*\ Q\ s \Leftrightarrow l_*\ P\ r = gcdb = \gcd(P\ r, Q\ s). \tag{2.4.17}$$

The solution (l_*, m_*) of this equation can be found in $O(\log(\max(P\ r, Q\ s)))$ time and space by using the extended Euclid's algorithm [29] for computing *gcdb*. Then, one computes $\gcd(r, s)$ and rewrites Θ as a disjoint union of intervals $\Theta_h = [h\ \gcd(r,s) \ldots (h+1)\ \gcd(r,s))$ with $h \in \mathbb{Z}$. The expression $x \Leftrightarrow y$ is rewritten as $\delta + \gamma$ with $\delta \bmod \gcd(r,s) = 0$ and $0 \leq \gamma < \gcd(r,s)$. Because $\gcd(r,s)$ divides δ, there exist α and β such that $\alpha\ r + \beta\ s = \delta$. Thus, the Equation (2.4.16) can be rewritten as

$$m\ Q\ s \Leftrightarrow l\ P\ r = (p + \alpha)\ r \Leftrightarrow (q + \beta)\ s + \gamma \tag{2.4.18}$$

This equation has a solution if and only if *gcdb* divides the right hand side. In this case, a solution pair (l_h, m_h) is obtained from the particular solution (l_*, m_*) previously computed. There is no guarantee that this solution pair will belong to the correct interval, so some scaling may be necessary. It is then easy to recover x_h and y_h. In addition, since the quadruplet solutions (l_h, m_h, x_h, y_h) may be found in any order, sorting may also be necessary. The Equation (2.4.16) is thus completely solved, and one precisely knows in every process which block owns diagonal entries and how to find those diagonals within each block. Each quadruplet solution corresponds uniquely to a multiple of *gcdb* in Θ as illustrated in Figure 2.6. This one-to-one mapping implies that the quadruplet solutions are

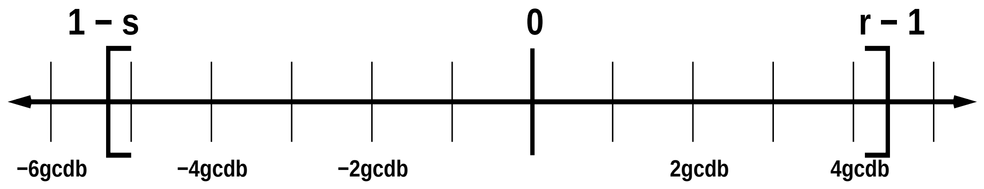


Figure 2.6: The quadruplet solution intervals Θ_h

periodic and will be successively found by the above method, so that one can stop as soon as a quadruplet solution has been found twice.

The approach of solving a set of linear Diophantine equations to determine index sets, process sets and so on is recommended by data parallel compiler designers as one way to proceed [15, 56, 64, 82]. Binary algorithms are available [65] to solve these equations. This method is thus very general and relatively inexpensive in terms of time. Still, this approach is the most powerful and expensive method in terms of memory requirements. Dynamic storage facilities are needed for the quadruplet solutions [63]. It can be adapted to accommodate variations of the block cyclic distributions that are supported by the HPF language.

2.5 LCM Tables

Definition 2.5.1 The k-diagonal of a matrix is the set of entries a_{ij} such that $i \Leftrightarrow j = k$.

Remark. With this definition the 0-diagonal is the "main" diagonal of a matrix. The first subdiagonal and superdiagonal are respectively the 1-diagonal and the $\Leftrightarrow 1$-diagonal.

Definition 2.5.2 Given a k-diagonal, the k-LCM table (LCMT) is a two-dimensional infinite array of integers local to each process (p, q) defined recursively by

$$\begin{cases} LCMT_{0,0}^{p,q} \quad = q\ s \Leftrightarrow p\ r + k, \\ \forall l \in \mathbb{N}, \quad LCMT_{l,*}^{p,q} = LCMT_{l-1,*}^{p,q} \Leftrightarrow P\ r, \\ \forall m \in \mathbb{N}, \quad LCMT_{*,m}^{p,q} = LCMT_{*,m-1}^{p,q} + Q\ s. \end{cases}$$

An equivalent direct definition is

$$\forall (l, m) \in \mathbb{N}^2 \quad LCMT_{l,m}^{p,q} = m\ Q\ s \Leftrightarrow l\ P\ r + q\ s \Leftrightarrow p\ r + k.$$

The equation for the 0-diagonal (2.4.16) is generalized to the k-diagonal by

$$LCMT_{l,m}^{p,q} = x \Leftrightarrow y, \tag{2.5.19}$$

with (x, y) in Θ. Thus, blocks owning the k-diagonal entries are such that

$$1 \Leftrightarrow s \leq LCMT^{p,q}_{l,m} \leq r \Leftrightarrow 1. \qquad (2.5.20)$$

In addition the value of $LCMT^{p,q}_{l,m}$ specifies where the diagonal starts within a block owning diagonals as illustrated in Figure 2.7.

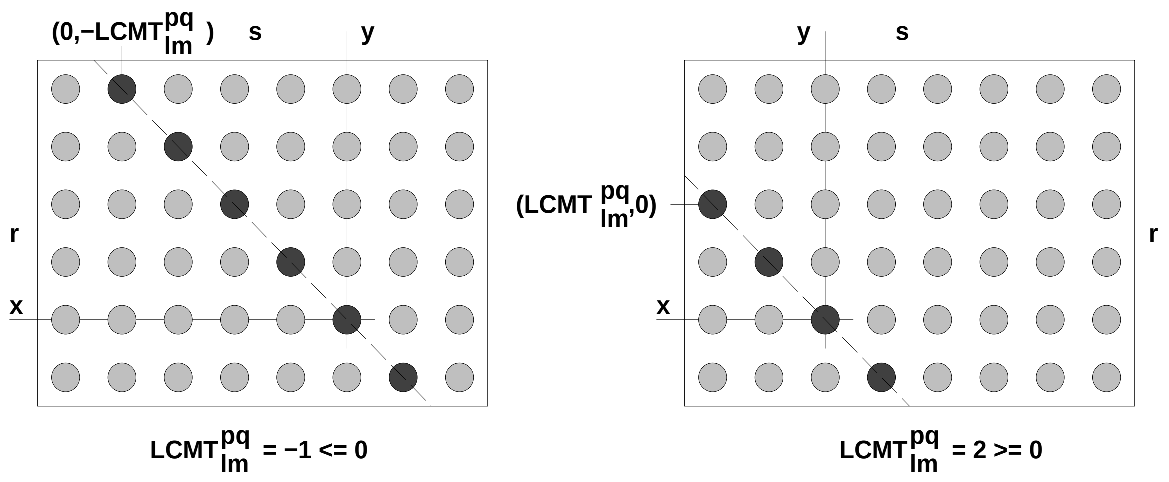


Figure 2.7: Meaning of different values of $LCMT^{p,q}_{l,m}$ with $r = 6$, $s = 8$

Property 2.5.1 The local blocks in process (p, q) such that $LCMT^{p,q}_{l,m} \leq 0$ own matrix entries a_{ij} that are globally below the k-diagonal.

Property 2.5.2 The local blocks in process (p, q) such that $LCMT^{p,q}_{l,m} \geq 0$ own matrix entries a_{ij} that are globally above the k-diagonal.

Property 2.5.3 The local blocks in process (p, q) such that $LCMT^{p,q}_{l,m} \leq \Leftrightarrow s$ correspond globally to strictly lower blocks of the matrix.

Property 2.5.4 The local blocks in process (p, q) such that $LCMT^{p,q}_{l,m} \geq r$ correspond globally to strictly upper blocks of the matrix.

Property 2.5.5 Within each process, if the $r \times s$ block of local coordinates (l, m) owns k-diagonal entries, the block of local coordinates $(l + 1, m)$ (respectively $(l, m + 1)$) owns either k-diagonals or matrix entries that are strictly below (respectively above) the k-diagonal.

Property 2.5.6 Within each process, if the $r \times s$ blocks of local coordinates (l, m) and $(l + 1, m)$ (respectively $(l, m + 1)$) own k-diagonals, then the block of local coordinates $(l, m+1)$ (respectively $(l+1, m)$) owns matrix entries that are strictly above (respectively below) the k-diagonal.

Proof. These last properties are direct implications of the LCM table definition. ∎

Figure 2.8 shows an LCM block-partitioned matrix and the $r \times s$ blocks of this matrix that reside in the process of coordinates (p, q). Depending on their relative position to the k-diagonal, these blocks are identified by a different shade of color. The arrangement of these blocks in process (p, q) is also represented and denoted by the local array in process (p, q). This figure illustrates the previous properties and demonstrates that the essential piece of information necessary to locate the diagonals locally in process (p, q) is contained in the diagonal LCM blocks. These diagonal LCM blocks separate the upper and lower parts of the matrix. Moreover, because of the periodicity of the distribution mapping mentioned earlier in this chapter, only one diagonal LCM block is needed in order to locate the k-diagonals in every process of the grid. This implies that only a very small fraction of the LCM table needs to be computed to solve the problem of interest. Furthermore, the size of the meaningful part of the LCM table can be computed in $O(\sqrt{lcmp^2 + lcmq^2})$ time. Thus, this information is very cheap to obtain and one can afford to recompute it when needed as opposed to what was done for the linear Diophantine equation method discussed in the previous section.

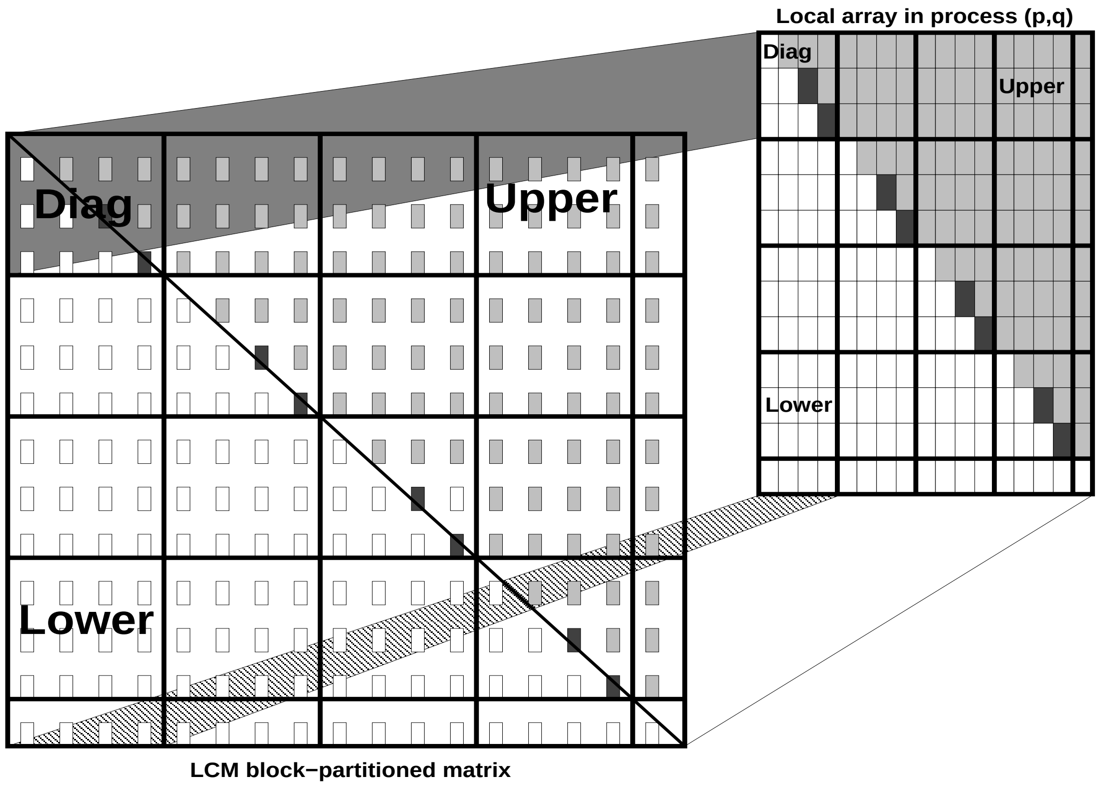


Figure 2.8: LCM template

Figure 2.9 shows a 1-LCM block for a given set of distribution parameters P, r, Q and s. Figure 2.10 shows the associated 1-LCM tables corresponding to the 1-LCM block shown in Figure 2.9. Each of these tables is associated to a distinct process of coordinates (p, q). These coordinates are indicated in the upper left corner of each table. Examine for example the table corresponding to

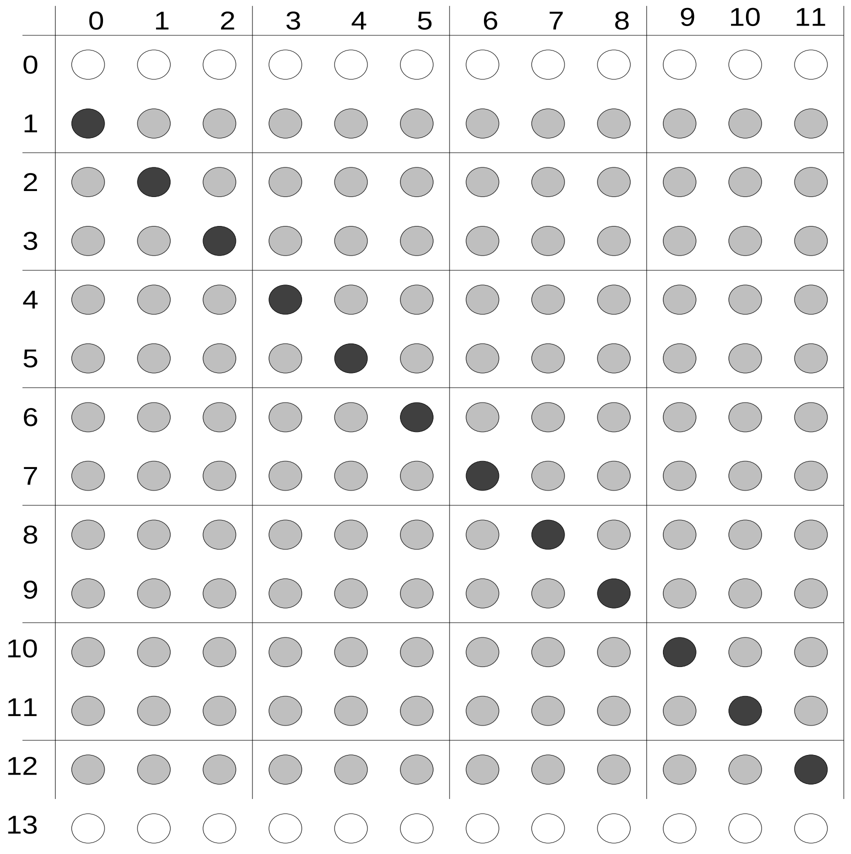


Figure 2.9: The 1-LCM block obtained for $P = 2$, $r = 2$, $Q = 2$ and $s = 3$.

process $(0, 0)$. The value of the LCM table entry $(0, 0)$ is 1. Since this value is greater than $\Leftrightarrow s = \Leftrightarrow 3$ and less than $r = 2$, it follows that this block $(0, 0)$ owns diagonals. Moreover, locally within this block the diagonal starts in position

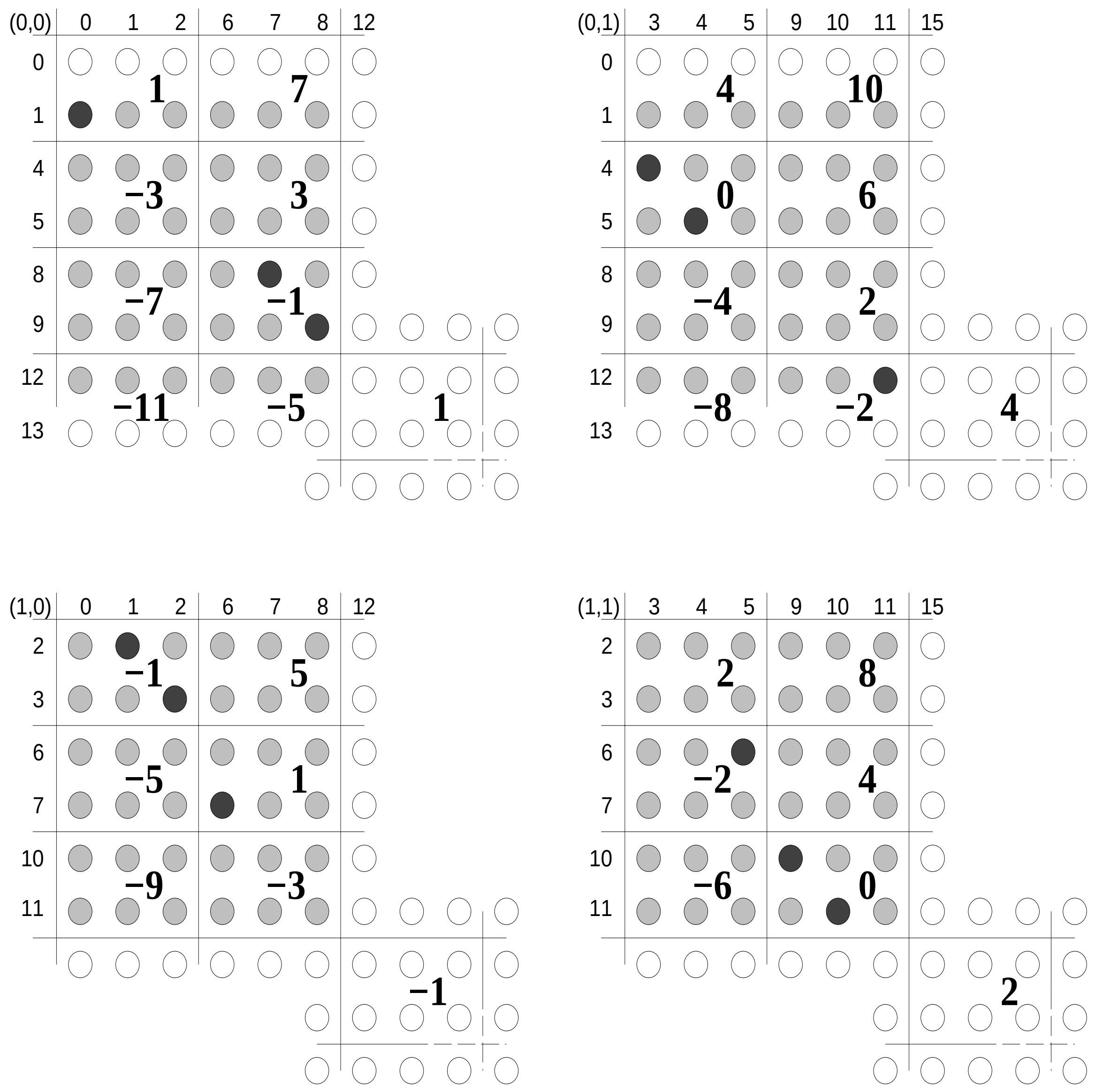


Figure 2.10: The 1-LCM tables obtained for $P = 2$, $r = 2$, $Q = 2$ and $s = 3$.

$(LCMT^{00}_{00}, 0) = (1, 0)$. The periodicity in this table is shown by the block of coordinates $(3, 2)$ which is such that $LCMT^{00}_{00} = LCMT^{00}_{32} = 1$. One can also verify that a block of local coordinates (l, m) in this table corresponds to a strictly lower (respectively upper) block in the original 1-LCM block (see Figure 2.9) if and only if $LCMT^{00}_{lm} \leq \Leftrightarrow s$ (respectively $LCMT^{00}_{lm} \geq r$). These same remarks apply to all of the other LCM tables shown in Figure 2.10.

Property 2.5.7 The number of $r \times s$ blocks owning k-diagonal entries is given by

$$\begin{cases} lcmb\ (\dfrac{r + s \Leftrightarrow \gcd(r, s)}{r\ s}) & \text{if } \gcd(r, s) \text{ divides } k, \\ lcmb\ (\dfrac{r + s}{r\ s}) & \text{otherwise.} \end{cases}$$

Proof. (sketch) First note that one can assume $\Leftrightarrow s < k < r$ without loss of generality by renumbering the processes with their relative process coordinates. Second, consider an array of $r \times s$ blocks of size $\mathrm{lcm}(r, s)$. If k divides $\gcd(r, s)$, there is exactly one $r \times s$ block such that its $(r \Leftrightarrow 1, s \Leftrightarrow 1)$ entry belongs to the k-diagonal. Otherwise, such a block does not exist. Third, the column (respectively row) edges of the blocks will be cut exactly $\mathrm{lcm}(r, s)/s$ (respectively $\mathrm{lcm}(r, s)/r$) times by the k-diagonal. To see that $lcmb/\mathrm{lcm}(r, s)$ is indeed an integer, one may observe that this quantity can be rewritten as $((u\ Q)\ P\ r + (t\ P)\ Q\ s)/gcdb$ with u and t in $\mathbb{Z}$. Finally, there are exactly $lcmb/\mathrm{lcm}(r, s)$ such blocks in an LCM block. ∎

Property 2.5.8 If $\gcd(r,s)$ divides k, a sufficient condition for all $P \times Q$ processes to own k-diagonals is given by $r + s \Leftrightarrow \gcd(r,s) \geq gcdb$. Otherwise, i.e., when $\gcd(r,s)$ does not divide k, a sufficient condition for all $P \times Q$ processes to own k-diagonals is given by $r + s \geq gcdb$.

Proof. Remark that $\gcd(r,s)$ divides $gcdb$. If $\gcd(r,s)$ divides k (note that this will always be the case if $\gcd(r,s) = 1$), the number of multiples of $\gcd(r,s)$ in the interval $I_{p,q} = (p\,r \Leftrightarrow (q \Leftrightarrow 1)\,s \ldots (p+1)\,r \Leftrightarrow q\,s)$ is $\dfrac{r+s}{\gcd(r,s)} \Leftrightarrow 1$. The number of multiples of $gcdb$ in the interval $I_{p,q}$ is $\dfrac{gcdb}{\gcd(r,s)}$. Thus, the inequality $\dfrac{r+s}{\gcd(r,s)} \Leftrightarrow 1 \geq \dfrac{gcdb}{\gcd(r,s)}$ is a sufficient condition for a multiple of $gcdb$ to be in this interval $I_{p,q}$. Otherwise, i.e., when $\gcd(r,s)$ does not divide k, Equation 2.5.20 can be rewritten as

$$p\ r \Leftrightarrow (q+1)\ s < m\ Q\ s \Leftrightarrow l\ P\ r + k < (p+1)\ r \Leftrightarrow q\ s. \qquad (2.5.21)$$

For any given process of coordinates (p,q), there must exist a $t \in \mathbb{Z}$ such that $m\ Q\ s \Leftrightarrow l\ P\ r = t\ gcdb$ verifying Inequality 2.5.21. Moreover, the interval of interest $I_{p,q}$ is of length $r + s \Leftrightarrow 1$. A sufficient condition for all processes to have k-diagonals is given by $r + s \Leftrightarrow 1 \geq gcdb$. Since $\gcd(r,s) \neq 1$ and $\gcd(r,s)$ divides $gcdb$, this sufficient condition can be equivalently written as $r + s \geq gcdb$. ▮

Property 2.5.9 If $\gcd(r,s)$ divides k, a necessary condition for all $P \times Q$ processes to own k-diagonals is given by $r + s \Leftrightarrow \gcd(r,s) \geq gcdb$. Otherwise, when gcd$(r, s)$ does not divide k, a necessary condition for all $P \times Q$ processes to own k-diagonals is given by $r + s \geq gcdb$.

Proof. Suppose there exists a process (p, q) having two distinct blocks owning k-diagonals. Then, $r + s \Leftrightarrow \mathrm{gcd}(r, s) \geq gcdb$ if gcd(r, s) divides k, and $r + s \geq gcdb$ otherwise. Indeed, there are two multiples of $gcdb$ in some interval $I_{p,q} = (p\,r \Leftrightarrow (q \Leftrightarrow 1)\,s \ldots (p+1)\,r \Leftrightarrow q\,s)$. Otherwise, each process owns at most one $r \times s$ block in which k-diagonals reside. Therefore, the number of blocks owning k-diagonals is equal to the number of processes owning these diagonals. The result then follows from Property 2.5.7. ∎

Corollary 2.5.1 A necessary and sufficient condition for every process to own k-diagonal entries is given by $r + s \Leftrightarrow \mathrm{gcd}(r, s) \geq gcdb$ if gcd(r, s) divides k and $r + s \geq gcdb$ otherwise.

PROOF. This result directly follows from the two preceding properties. ∎

Corollary 2.5.2 The number of processes owning k-diagonal entries is equal to the maximum of $P \times Q$ and the number of blocks owning k-diagonals. This number is given by

$$\begin{cases} \max(P\,Q\,(\dfrac{r + s \Leftrightarrow \mathrm{gcd}(r, s)}{gcdb}), P\ Q) & \text{if } \mathrm{gcd}(r, s) \text{ divides } k, \\ \max(P\,Q\,(\dfrac{r + s}{gcdb}), P\ Q) & \text{otherwise.} \end{cases}$$

PROOF. The result follows from the fact that $lcmb/(r\ s) = (P\ Q)/gcdb$ and Properties 2.5.7, 2.5.8 and 2.5.9. ∎

These last properties generalize the results presented in Table 2.1. They are summarized below in Table 2.2. The end of this section aims at determining the probability that the quantities $r+s$ or $r+s \Leftrightarrow \mathrm{gcd}(r,s)$ are greater or equal to $gcdb$, that is, the probability that every process owns k-diagonals entries. The result

Table 2.2: Properties of the k-diagonal for the block cyclic distribution

Blocks owning k-diagonals	$\Leftrightarrow s < m\ Q\ s \Leftrightarrow l\ P\ r + q\ s \Leftrightarrow p\ r + k < r$
Processes owning k-diagonals	$\begin{cases} \exists t \in \mathbb{Z}, \text{ such that} \\ p\ r \Leftrightarrow (q+1)s < t\ gcdb + k < (p+1)r \Leftrightarrow q\ s \end{cases}$
Number of such processes	$\begin{cases} \min(\frac{P\,Q\,(r+s)}{gcdb}, P\,Q) \text{ if } k \bmod \mathrm{gcd}(r,s) \neq 0, \\ \min(\frac{P\,Q\,(r+s \Leftrightarrow \mathrm{gcd}(r,s))}{gcdb}, P\,Q) \text{ otherwise.} \end{cases}$

obtained is particularly interesting because it quantifies the complexity of general redistribution operations as a function of the distribution parameters, namely the perimeter $r+s$ of the $r \times s$ partitioning unit and the quantities $\mathrm{gcd}(r,s)$ and $gcdb = \mathrm{gcd}(P\,r, Q\,s)$.

Property 2.5.10 If P, r, Q and s are integers chosen at random, the probability $P1$ that $gcdb = \gcd(P\ r, Q\ s) = 1$ is

$$P1 = \prod_{p\ prime} (1 \Leftrightarrow (\frac{2\,p \Leftrightarrow 1}{p^2})^2) \approx 0.21778\ldots.$$

Proof. A precise formulation of this property as well as its proof should carefully define what is meant here by "chosen at random". This precision however, does complicate the following argumentation in a useless manner. More information on the random generation of integers for the purpose of a proof can be found in [65]. For the sake of simplicity, only the essential arguments are presented below. We first state and prove two useful lemmas.

Lemma 2.5.1 Let p be a prime and α a positive integer. The probability q_{p^α} that p^α divides the product $P\,r$ where P and r are integers chosen at random is given by

$$q_{p^\alpha} = \frac{(\alpha + 1)\,p \Leftrightarrow \alpha}{p^{\alpha+1}}.$$

Proof. If p^α divides the product $P\,r$, then either p^α divides P, or, $p^{\alpha-k}$ divides P and $p^{\alpha-k+1}$ does not divide P and p^k divides r for all values of k in $[1\ldots\alpha]$. It follows that

$$q_\alpha = \frac{1}{p^\alpha} + \sum_{k=1}^{\alpha} \frac{1}{p^{\alpha-k}}(1 \Leftrightarrow \frac{1}{p})\frac{1}{p^k} = \frac{1}{p^\alpha}(1 + \alpha \Leftrightarrow \frac{\alpha}{p}) = \frac{(\alpha+1)\,p \Leftrightarrow \alpha}{p^{\alpha+1}}. \qquad \blacksquare$$

Lemma 2.5.2 Let d be an integer and $d = \prod_{i \in I_d} p_i^{\alpha_i}$ its decomposition into prime factors. The probability q_d that d divides the product $P\,r$ where P and r are integers chosen at random is given by

$$q_d = \prod_{i \in I_d} \frac{(\alpha_i + 1)\, p_i \Leftrightarrow \alpha_i}{p_i^{\alpha_i + 1}}.$$

Proof. This result directly follows from the application of Lemma 2.5.1 to each of the prime factors of d. ∎

The probability that p prime divides $P\,r$ and $Q\,s$ is thus $(\frac{2\,p \Leftrightarrow 1}{p^2})^2$ by application of Lemma 2.5.1. This probability is equal to the probability that p divides $gcdb = \gcd(P\ r, Q\ s)$. However, the probability $P1$ is the probability that there is not a prime dividing $gcdb$. Therefore, one considers the sequence of partial products

$$P_n = \prod_{first\ n\ primes\ p} (1 \Leftrightarrow (\frac{2\,p \Leftrightarrow 1}{p^2})^2).$$

The partial products P_n form a positive strictly decreasing quadratically convergent sequence. Therefore, $P1 = \lim_{n \to \infty} P_n$ exists. An easy computation show that an approximate value of $P1$ is $0.21778\ldots$. ∎

Remark. With the help of a computer, it is possible to enumerate for a given n all the 4-tuples (P, r, Q, s) lying in the finite range $1 \leq P, r, Q, s \leq n$ such that $gcdb = 1$. Figure 2.11 shows the obtained results for different values of $n \leq 500$.

First, note that 500 corresponds to a 250000 process grid, which is by far larger than any existing DMCC. It is also remarkable that the finite ratio is always larger than $P1 \approx 0.21778\ldots$. Second, the "convergence" rate of the probability is incredibly rapid for small values of n. The value of this probability can therefore be considered as almost exact or at worst a very accurate lower bound for all possible values of the distribution parameters.

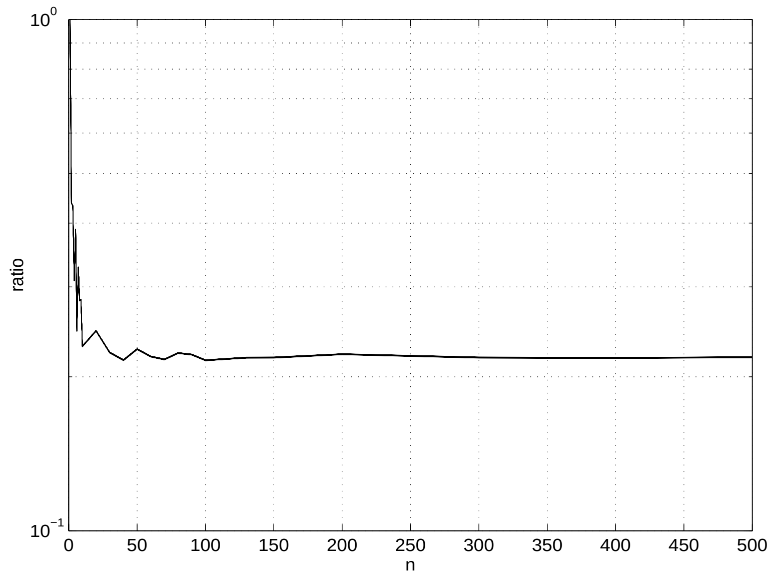


Figure 2.11: Ratio of tuples $(P, r, Q, s) in [1..n]^4$ such that $\gcd(P\,r, Q\,s) = 1$.

Finally, an analogous result known as Dirichlet's theorem [34] states that the probability that $\gcd(u, v) = 1$ for u and v integers chosen at random exists and is $\frac{6}{\pi^2}$. This result does not directly apply to the above property since in this latter case one is interested in the number of divisors of a product of integers. The proof techniques are however very similar.

Property 2.5.11 Let d be an integer and $d = \prod_{i \in I_d} p_i^{\alpha_i}$ its decomposition into prime factors. If P, r, Q and s are integers chosen at random, the probability Pd that $gcdb = \gcd(P\ r, Q\ s) = d$ is

$$Pd = q_d^2 \prod_{p\ prime} (1 \Leftrightarrow q_{pd}^2)$$

where q_d (respectively q_{pd}) is the probability that d (respectively $p\,d$ with p prime) divides $P\,r$. Furthermore, we have

$$q_d = \prod_{i \in I_d} \frac{(\alpha_i + 1)p_i \Leftrightarrow \alpha_i}{p_i^{\alpha_i + 1}} \text{ and } q_{pd} = \begin{cases} \dfrac{(\alpha_i + 2)p_i \Leftrightarrow (\alpha_i + 1)}{((\alpha_i + 1)p_i \Leftrightarrow \alpha_i)p_i} q_d \text{ if } \exists i \in I_d | p = p_i, \\ \dfrac{2\,p \Leftrightarrow 1}{p} q_d \text{ if } p \text{ does not divide } d. \end{cases}$$

Proof. The probability Pd is the product of the probability q_d that d divides $gcdb$ and the probability that there is no prime factor p such the product $p\,d$ divides $gcdb$. The probability q_d is obtained by a direct application of Lemma 2.5.2. The probability that the product $p\,d$ divides $gcdb$ is more delicate to compute since it depends on whether p is a prime factor of d or not. The result however directly follows from Lemmas 2.5.1 and 2.5.2. The existence and convergence of the product follows from the fact that $|q_{pd}| \leq |q_d|$. ∎

The above properties do not lead in a straightforward manner to the desired probability that all processes will own k-diagonal entries. However, these properties illustrate the fact that this probability, if it exists, is likely to converge at a high rate. Once again, it is possible to rely on a computer to enumerate all 4-tuples in a finite and practical range such that the quantities $r + s \Leftrightarrow \text{gcd}(r, s)$ or $r + s$ are greater or equal to *gcdb*. The results are presented in Figure 2.12. It is important

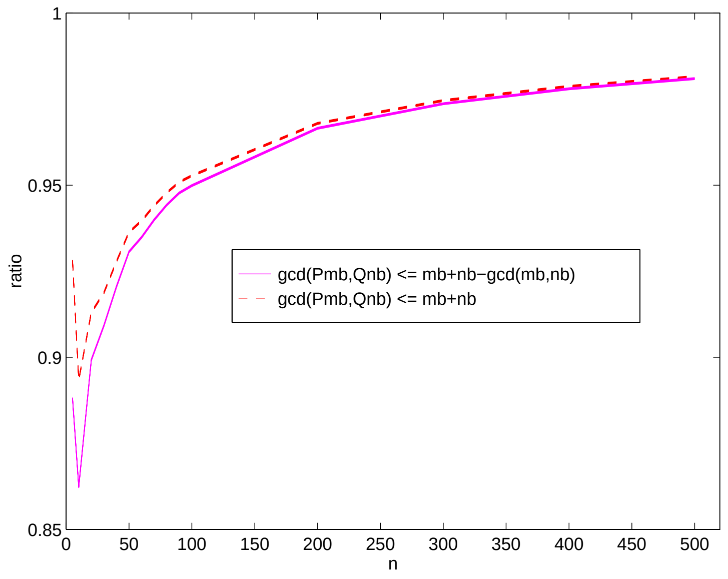


Figure 2.12: Ratios of tuples $(P, r, Q, s) in [1..n]^4$ such that $r + s - \text{gcd}(r, s)$ or $r + s$ is greater or equal to $\text{gcd}(P\,r, Q\,s)$.

to notice that in practice, i.e., for a finite range of values $(1 \leq P, r, Q, s \leq n)$, there is almost no difference between the finite ratios of all 4-tuples verifying these inequalities over n^4. This figure does not prove the existence of the limit and therefore of the probability. However, if it exists, its value is likely to be very close to 1. In other words, if one picks random and realistic distribution parameters, it is very likely that all processes in the grid will own k-diagonals.

2.6 Rationale

First, Figure 2.12 not surprisingly shows that the ratios of distribution parameters such that k-diagonals are evenly distributed tends towards one. More interesting is the fact that this function increases very rapidly ($r(10) \approx .88$, $r(20) \approx .90$, $r(50) \approx .93$). Therefore, it is very likely that all processes in the grid will own k-diagonals. The Corollaries (2.5.1) and (2.5.2) say that the distribution of the k-diagonals essentially depends on the perimeter of the $r \times s$ partitioning unit as opposed to its shape. This says that restricting the data decomposition to a square block cyclic mapping does not affect in any way the problem of locating the k-diagonals, and consequently the complexity of redistribution operations. Finally, assume that the complexity of redistribution operations in terms of the number of messages exchanged for the same volume of data to be communicated grows with the number of processes owning k-diagonals. The next two chapters will confirm the validity of this assumption. In this case, it follows that small blocking factors are favorable for interconnection networks featuring a large startup time or latency, but high bandwidth. Conversely, small startup time and lower bandwidth are more well-suited for medium and large blocking factors, as far as the performance of redistribution operations is concerned. Consequently transportable efficiency for redistribution operations requires the support of the parameterized family of block cyclic mappings.

The definitions of an LCM table given in this chapter can easily be generalized to a block cyclic distribution with a partial first block. In other words, the first block of rows (respectively columns) is of size ir (respectively is) instead of r (respectively s). This more general definition, as well as a number of examples of LCM tables, have been added to Appendix A of this document. An alternate definition of an LCM table entry would be the global number of columns up to the blocks of local coordinates $(*, m)$ minus the global number of rows up to the blocks of local coordinates $(l, *)$. This constructive definition is more general than the one used in this dissertation. It encompasses the entire family of Cartesian mappings [9].

The algorithmic redistributed operations described later in this dissertation can be expressed in terms of locating diagonals of a distributed matrix. The next chapters also illustrate the fundamental role played by LCM tables and the properties presented above in the formulation of these operations. Moreover, the implications of these properties are analyzed in greater detail as these operations are specified in this document. Still, the correctness of these operations and the robustness and reliability of their implementation depend entirely on the material presented in this chapter.

Chapter 3

Algorithmic Redistribution

3.1 Introduction

In a serial computational environment, *transportable efficiency* is the essential motivation for developing blocking strategies and block-partitioned algorithms [3, 5, 33, 60]. The linear algebra package (LAPACK) [5] is the archetype of such a demarche. The LAPACK software is constructed as much as possible out of calls to the BLAS (Basic Linear Algebra Subprograms). These kernels confine the impact of the machine architecture differences within a small number of routines. The efficiency and portability of the LAPACK software are then achieved by combining native and efficient BLAS implementations with portable high-level components. The BLAS are subdivided in three levels, each of which offering increased scope for exploiting parallelism. This classification criterion happens to also correspond to three different kinds of basic linear algebra operations:

- Level 1 BLAS [68]: for vector operations, such as $y \leftarrow \alpha x + y$,
- Level 2 BLAS [36]: for matrix-vector operations, such as $y \leftarrow \alpha Ax + \beta y$,
- Level 3 BLAS [35]: for matrix-matrix operations, such as $C \leftarrow \alpha AB + \beta C$.

Here, A, B, and C are matrices, x and y are vectors, and α and β are scalars. The performance potential of the three levels of BLAS is strongly related to the ratio of floating point operations to memory references, and the reuse of data when it is stored in the higher levels of the memory hierarchy. Consequently, the Level 1 BLAS cannot achieve high efficiency on most modern supercomputers. The Level 2 BLAS can achieve near-peak performance on many vector processors. On RISC microprocessors, however, their performance is limited by the memory access bandwidth bottleneck. The greatest scope for exploiting the highest levels of the memory hierarchy as well as other forms of parallelism is offered by the Level 3 BLAS [5].

The previous reasoning applies to distributed memory computational environments in two ways. First, in order to achieve overall high performance, it is necessary to express the bulk of the computation local to each process in terms of Level 3 BLAS operations. Second, developing a set of BLAS for DMCCs should lead to a straightforward port of the LAPACK software. This is the path followed by the ScaLAPACK research project [16, 39] as well as others [1, 13, 26, 41].

Such a design sounds simple and reasonable, even if little is said on the adequate blocking strategies for a distributed memory hierarchy. One answer is given by the *physical blocking* approach, where the distribution blocking factors are used as computational blocking units, hence inducing alignment restrictions on the operands. Most of the parallel algorithms proposed in the literature are physically blocked [25, 26, 74, 83]. High performance is achievable on a wide range of DMCCs, but usually depends on the distribution blocking factors. The alignment restrictions simplify the expression and implementation of these algorithms, but also limit their application scope in a way that does not satisfy general purpose library requirements. High performance can be maintained across platforms by parameterizing the user's data distribution or across library function calls by using general redistribution packages [78].

The purpose of this chapter is to propose alternatives to the physical blocking strategy. The originality of the algorithms presented here is their systematic derivation from the properties of the underlying mapping. These blocking strategies are expressed within a single framework using LCM tables. The resulting blocked operations are appropriate for library software. They indeed feature potential for high performance without any specific alignment restrictions on their operands. This says that the antagonism between efficiency and flexibility is not a property of the block cyclic mapping, but merely a characteristic of the algorithms that have been so far proposed to deal with a distributed memory hierarchy.

3.2 Terminology

This section defines some basic objects and terms that are heavily used in the rest of this chapter. An effort has been made to maintain consistency with the notations used in (2.2.5). The objects defined here are common and sometimes very intuitive. Hence, their definition may seem a little obscure. It corresponds however to the usual data structure used for their storage in a computer.

Definition 3.2.1 An $M \times N$ matrix A is a two-dimensional array of elements indexed by their relative row and column coordinates

$$A = \{a_{ij} \text{ with } (i,j) \in \Delta_A = \{0 \ldots M \Leftrightarrow 1\} \times \{0 \ldots N \Leftrightarrow 1\}\}.$$

Δ_A is called a virtual matrix (VM) or the index set associated with the matrix A.

Definition 3.2.2 An $M \times N$ (block cyclic) distributed matrix (DM) A is a $P \times Q$ matrix of matrices:

$$\begin{cases} A = \{A^{pq} \text{ with } (p,q) \in \, , \} \\ A^{pq} = \{a_{ij}^{pq} \text{ with } (i,j) \in \Delta_{A^{pq}}\} \end{cases}$$

with $\Delta_{A^{pq}} = \{(l\,r + x, m\,s + y) \text{ such that } (l,m,x,y) \in \Lambda \times \Theta\}$. The set $\Delta_A = \{\Delta_{A^{pq}} \text{ with } (p,q) \in \, , \}$ is called a distributed virtual matrix (DVM) or the index set associated with the distributed matrix A.

Remark. When Δ_A is empty, A is denoted by $\emptyset$ and called the null (distributed) matrix. In a computer, the null matrix is represented by a valid address in memory pointing to no data.

Definition 3.2.3 An $M \times N$ 1-dimensional column distributed matrix A is a distributed matrix such that there exists q_0 in $\{0 \ldots Q \Leftrightarrow 1\}$ and

$$\begin{cases} \forall p \in \{0 \ldots P \Leftrightarrow 1\},\ \forall q \in \{0 \ldots Q \Leftrightarrow 1\} \setminus \{q_0\},\ \Delta_{A^{pq}} = \emptyset \text{ and} \\ \Delta_{A^{pq_0}} = \{(l\,r + x, m\,s + y) \,|\, (l, m, x, y) \in \{0 \ldots \lfloor \frac{\frac{M-1}{r}}{P} \rfloor\} \times \{0 \ldots \frac{N-1}{s}\} \times \Theta\}. \end{cases}$$

Definition 3.2.4 An $M \times N$ 1-dimensional row distributed matrix A is a distributed matrix such that there exists p_0 in $\{0 \ldots P \Leftrightarrow 1\}$ and

$$\begin{cases} \forall q \in \{0 \ldots Q \Leftrightarrow 1\},\ \forall p \in \{0 \ldots P \Leftrightarrow 1\} \setminus \{p_0\},\ \Delta_{A^{pq}} = \emptyset \text{ and} \\ \Delta_{A^{p_0 q}} = \{(l\,r + x, m\,s + y) \,|\, (l, m, x, y) \in \{0 \ldots \frac{M-1}{r}\} \times \{0 \ldots \lfloor \frac{\frac{N-1}{s}}{Q} \rfloor\} \times \Theta\}. \end{cases}$$

Definition 3.2.5 An $M \times N$ local matrix A is a distributed matrix such that there exists $(p_0, q_0) \in ,$ and

$$\begin{cases} \forall (p, q) \in ,\ \setminus \{(p_0, q_0)\},\ \Delta_{A^{pq}} = \emptyset \text{ and} \\ \Delta_{A^{p_0 q_0}} = \{(l\,r + x, m\,s + y) \,|\, (l, m, x, y) \in \{0 \ldots \frac{M-1}{r}\} \times \{0 \ldots \frac{N-1}{s}\} \times \Theta\}. \end{cases}$$

There is another type of distributed matrix that occurs in a large number of distributed dense linear algebra computations. These are the *replicated* variants of the last three definitions. It is useful to incorporate these distributed matrices in our general reflection because they frequently contain intermediate results needed to avoid unnecessary communication phases.

Definition 3.2.6 An $M \times N$ 1-dimensional column replicated distributed matrix A is a distributed matrix such that there exists q_0 in $\{0 \ldots Q \Leftrightarrow 1\}$ and

$$\begin{cases} \forall p \in \{0 \ldots P \Leftrightarrow 1\},\ \forall q \in \{0 \ldots Q \Leftrightarrow 1\} \setminus \{q_0\}, \\ A^{pq} = A^{pq_0} = \{a_{ij}^{pq_0} \text{ with } (i,j) \in \Delta_{A^{pq_0}}\} \text{ and} \\ \Delta_{A^{pq_0}} = \{(l\,r + x, m\,s + y) \,|\, (l,m,x,y) \in \{0 \ldots \lfloor \frac{\frac{M-1}{r}}{P} \rfloor\} \times \{0 \ldots \frac{N-1}{s}\} \times \Theta\}. \end{cases}$$

Definition 3.2.7 An $M \times N$ 1-dimensional row replicated distributed matrix A is a distributed matrix such that there exists p_0 in $\{0 \ldots P \Leftrightarrow 1\}$ and

$$\begin{cases} \forall q \in \{0 \ldots Q \Leftrightarrow 1\},\ \forall p \in \{0 \ldots P \Leftrightarrow 1\} \setminus \{p_0\}, \\ A^{pq} = A^{p_0 q} = \{a_{ij}^{p_0 q} \text{ with } (i,j) \in \Delta_{A^{p_0 q}}\} \text{ and} \\ \Delta_{A^{p_0 q}} = \{(l\,r + x, m\,s + y) \,|\, (l,m,x,y) \in \{0 \ldots \frac{M-1}{r}\} \times \{0 \ldots \lfloor \frac{\frac{N-1}{s}}{Q} \rfloor\} \times \Theta\}. \end{cases}$$

Definition 3.2.8 An $M \times N$ local column replicated matrix A is a distributed matrix such that there exists $(p_0, q_0) \in$, and

$$\forall p \in \{0 \ldots P \Leftrightarrow 1\},\ \forall q \in \{0 \ldots Q \Leftrightarrow 1\} \setminus \{q_0\},\ \Delta_{A^{pq}} = \emptyset \text{ and } A^{pq_0} = A^{p_0 q_0}.$$

Definition 3.2.9 An $M \times N$ local row replicated matrix A is a distributed matrix such that there exists $(p_0, q_0) \in$, and

$$\forall q \in \{0 \ldots Q \Leftrightarrow 1\},\ \forall p \in \{0 \ldots P \Leftrightarrow 1\} \setminus \{p_0\},\ \Delta_{A^{pq}} = \emptyset \text{ and } A^{p_0 q} = A^{p_0 q_0}.$$

Definition 3.2.10 An $M \times N$ local replicated matrix A is a distributed matrix such that there exists $(p_0, q_0) \in$, and

$$\forall p \in \{0 \ldots P \Leftrightarrow 1\} \setminus \{p_0\},\ \forall q \in \{0 \ldots Q \Leftrightarrow 1\} \setminus \{q_0\}\ A^{pq} = A^{p_0 q_0}.$$

Finally, the notion of equality and equivalence of two distributed matrices with respect to the block cyclic distribution can be defined as follows:

Definition 3.2.11 Two $M \times N$ distributed matrices A and B are said to be equal with respect to their distribution if and only if $\Delta_A = \Delta_B$.

Definition 3.2.12 Two $M \times N$ distributed matrices A and B are said equivalent, noted $\Delta_A \equiv \Delta_B$, with respect to their distribution if and only if there are two integers u and t such that

$$\forall (p, q) \in\ ,\ , \Delta_{A^{pq}} = \Delta_{B^{vw}} \text{ with } v = (p + u) \bmod P \text{ and } w = (q + t) \bmod Q.$$

Property 3.2.1 If $\Delta_A = \Delta_B$, then $\Delta_A \equiv \Delta_B$.

3.3 One-Dimensional Redistribution

The operations described in this section involve only $M \times N$ one-dimensional distributed matrices as defined in (3.2.3) and (3.2.4). Let X and Y be such matrices. Let P_X and r_X (respectively Q_Y and s_Y) be the distribution parameters associated with X (respectively Y). In order to redistribute X into Y, one considers the DVM induced by X and Y and specified by the distribution parameters P_X, Q_Y, r_X and s_Y. The Figure 3.1 shows the block-partitioned operands X and Y

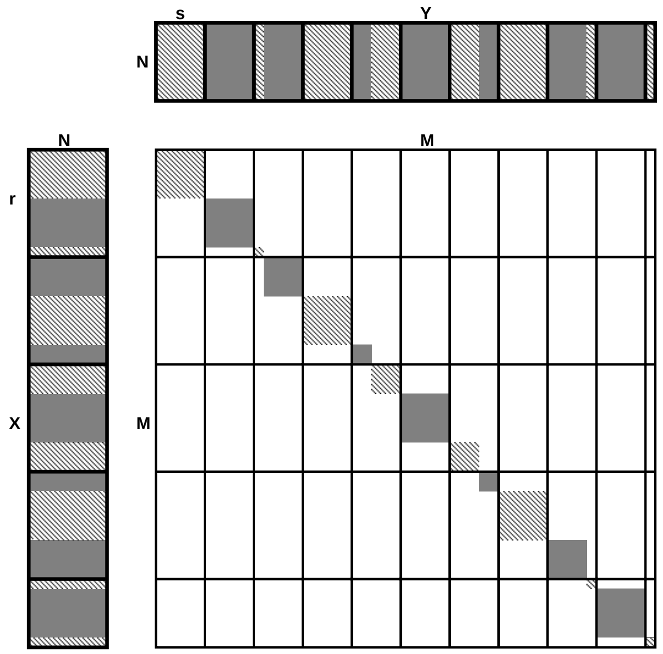


Figure 3.1: Global view of one-dimensional redistribution

as well as the induced DVM. For a given row of X, the corresponding diagonal entry of this DVM determines the corresponding column of Y. A block of rows of X that could be packed together is represented in the figure by a gray rectangle.

The corresponding diagonal block of the DVM and the corresponding columns of Y are colored with the same shade of gray. Figure 3.1 illustrates the importance of locating the diagonals in the context of one-dimensional redistribution. The process grid $P_X \times Q_Y$ is called a virtual process grid. Without loss of generality, one can assume that X (respectively Y) resides in column (respectively row) 0

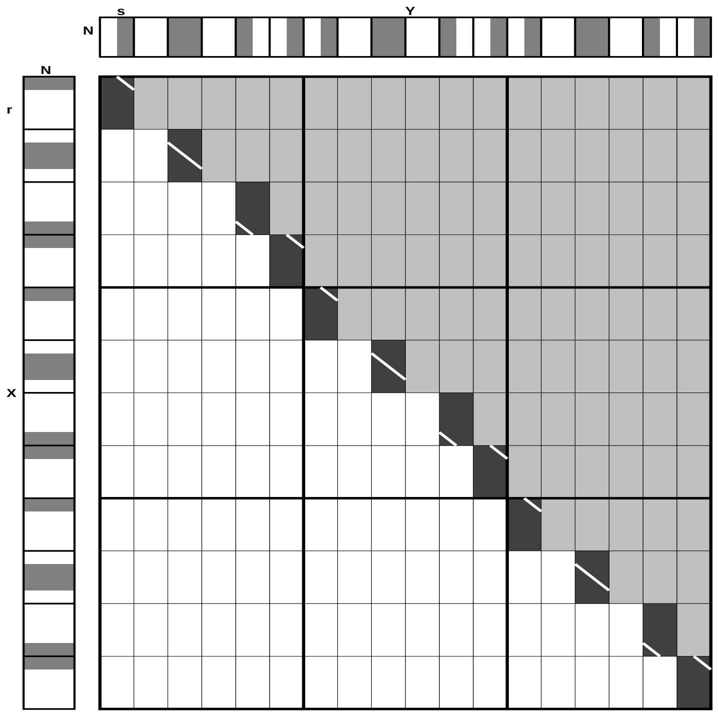


Figure 3.2: Local view in process (p_X, q_Y) of one-dimensional redistribution

of this virtual process grid. If a process of coordinates (p_X, q_Y) owns diagonal entries of the associated $M \times M$ virtual matrix, then some data residing in the process $(p_X, 0)$ should be sent to the process of coordinates $(0, q_Y)$. Figure 3.2 shows the local viewpoint of one-dimensional distribution in the virtual process of coordinates (p_X, q_Y). The blocks of the DVM owning diagonals are represented by darker rectangles in which the diagonals are symbolized by a white segment. Moreover, the rows of X marked in gray can be packed in a single message and sent to process $(0, q_Y)$. The knowledge of the source process coordinates allows the receiving process to determine a priori the size of the message to be received as well as its packed form and the location of each message entry. Consequently, unless X and Y are equivalent in the sense of (3.2.12), the number of messages to be exchanged is equal to the number p_d of processes owning diagonals of the DVM. The average size of each message is thus $M\,N/p_d$. This approach using LCM tables allows the handling of the shift and transpose operations in the same framework. Indeed, when X and Y are distributed along the same axis of the actual process grid, the operation shifts X into Y along this axis. Otherwise X is physically transposed into Y. The implementation of such an operation should take advantage of such a savings opportunity. In addition, since it is possible to detect via a simple test on the distribution parameters distribution equivalence, this scheme can be easily made optimal for the simpler cases. Note that the operands X and Y could be distributed on distinct process grids without affecting the packing strategy induced by the LCM tables. Moreover, if one uses the LCM table definition to handle a first partial block, this scheme naturally accommodates non-aligned operands. Finally, it is straightforward to handle replicated operands by taking replication into account when computing the LCM table entries. The next chapter will discuss the possible communication patterns associated with the one-dimensional redistribution as well as their complexity. Pseudo algorithms will then be presented.

3.3.1 Non-Unit Stride

The computation of the LCM table can be easily adapted if one wants to access the entries of a $M \times 1$ one-dimensional distributed matrix separated by a non unit stride s. The block cyclic distribution Definition (2.2.5) gives

$$(l\, P_X + p)\, r_X + x = k\, s \text{ with } k \in I\!N, \text{i.e., } p\, r_X \leq k\, s \Leftrightarrow l\, P_X\, r_X < p\, r_X.$$

Therefore, the processes p having entries of this scattered one-dimensional matrix are such that

$$p\, r_X \leq m \gcd(s, P_X\, r_X) < (p+1)\, r_X. \tag{3.3.1}$$

This problem is then a particular instance of locating the diagonals of a DVM distributed over a $P_X \times 1$ process grid and partitioned into $r_X \times s$ blocks. The construction of the LCM tables is sufficient if one only wants to access the data. However, it may be necessary to redistribute this scattered operand in order to perform some computation with it. This redistribution phase is facilitated by noticing that the column index of the LCM table uniquely identifies the global index of the entries of the scattered operand.

This scheme can be extended to two-dimensional distributed matrices. This corresponds to a stride different from the leading dimension in a serial Fortran environment. The implementation is, however, slightly tricky because of the multiple addressing space of the target machines. Indeed, the local leading dimension of the array storing the local pieces of a distributed matrix plays a role when two consecutive scattered entries belong to two different columns of this array. In this case, the local stride or offset within a block given by the LCM table may need to be augmented by some local value that depends on the effective number of rows stored in this local array and the leading dimension of this Fortran array.

3.4 Blocking Strategies

This section presents different kinds of blocking strategies for distributed memory hierarchies. Each of them exploits specific and sometimes antagonistic features of different operation contexts. They can all be formulated in terms of "LCM-operations", i.e., operations relying on LCM tables for their expression and implementation. Most of the blocking strategies presented below are known and their use has mainly been illustrated in specific applications. The originality of this section is mainly the presentation of these distinct techniques within a single framework, making them suitable for their integration into a software library. For some of these strategies little is known in terms of their impact on efficiency and/or ease of modular implementation. To our knowledge, no practical experiments have been so far reported in the literature. This dissertation is the earliest known document to present the results of such experiments. The same example operation called a rank-K update is used to illustrate the differences between all blocking strategies presented below. This operation produces an $M \times N$ matrix C by adding to itself the product of an $M \times K$ matrix A and a $K \times N$ matrix B

$$C \leftarrow C + A\,B.$$

3.4.1 Static Blocking

The *static blocking* strategy deals only with purely local computational phases. It is assumed that the operation has reached a stage where the operands have already been redistributed if necessary by other techniques. Only local remaining computations need to be performed. It may, however, be the case that a local output operand has to be redistributed subsequently. Within this context, the rank-K update operation is easy to describe. The matrix A has been replicated in every process column and the matrix B replicated in every process row. The update is performed by a single call to the matrix multiply subprogram. In this particular case, the simplicity of the operation is due to the fact that the local and global point of view are identical as illustrated in Figure 3.3. This figure shows the LCM block-partitioned matrices A, B and C. The blocks residing in the process of coordinates (p, q) are indicated by gray rectangles. The arrangement of these blocks in process (p, q) is also represented and denoted by the local arrays in process (p, q).

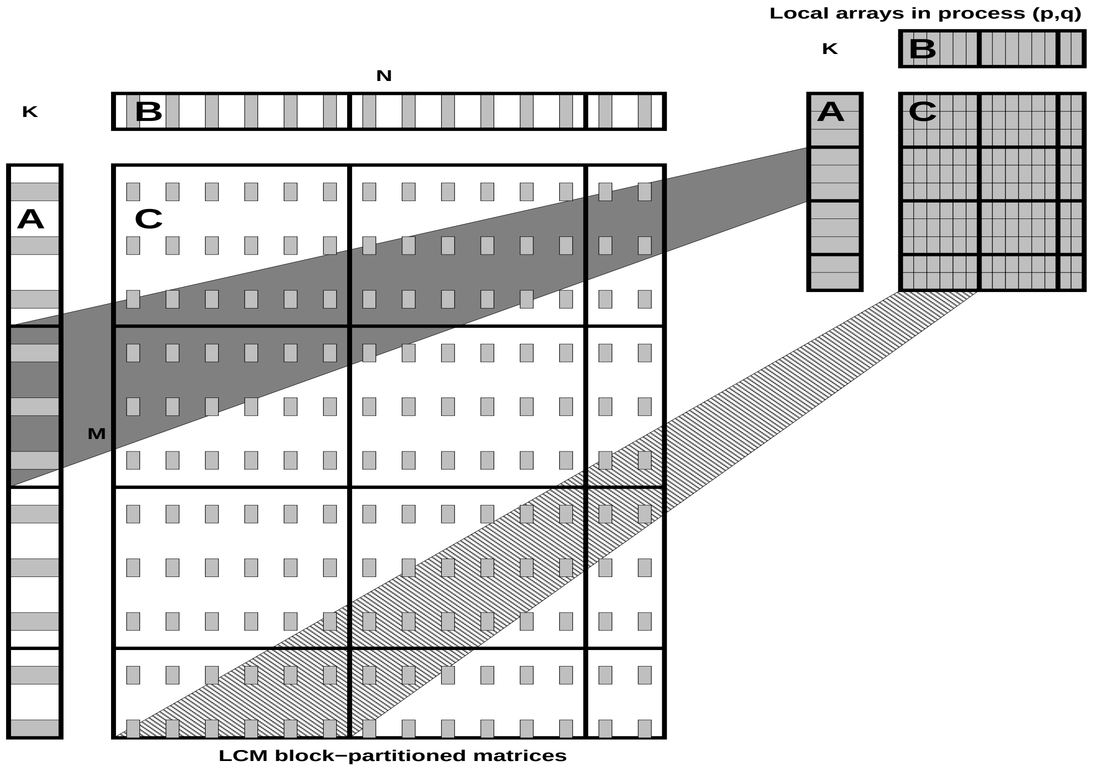


Figure 3.3: Static general rank-K update

The static blocking strategy becomes more interesting when the operation involves a triangular or symmetric matrix C, for which only the upper or lower triangle should be referenced. When C is symmetric, M is equal to N and B is A^T. As before, it is assumed that A and A^T have already been replicated across process columns and rows respectively. The distributed matrix C is partitioned into diagonal and strictly upper or lower LCM blocks as shown in Figure 3.4. This figure shows the LCM block-partitioned matrices A and C and the $r \times s$, $r \times K$

and $K \times s$ blocks of these matrices that reside in the process of coordinates (p, q). The arrangement of these blocks in process (p, q) is also represented and denoted by the local arrays in process (p, q). Depending on their relative position to the

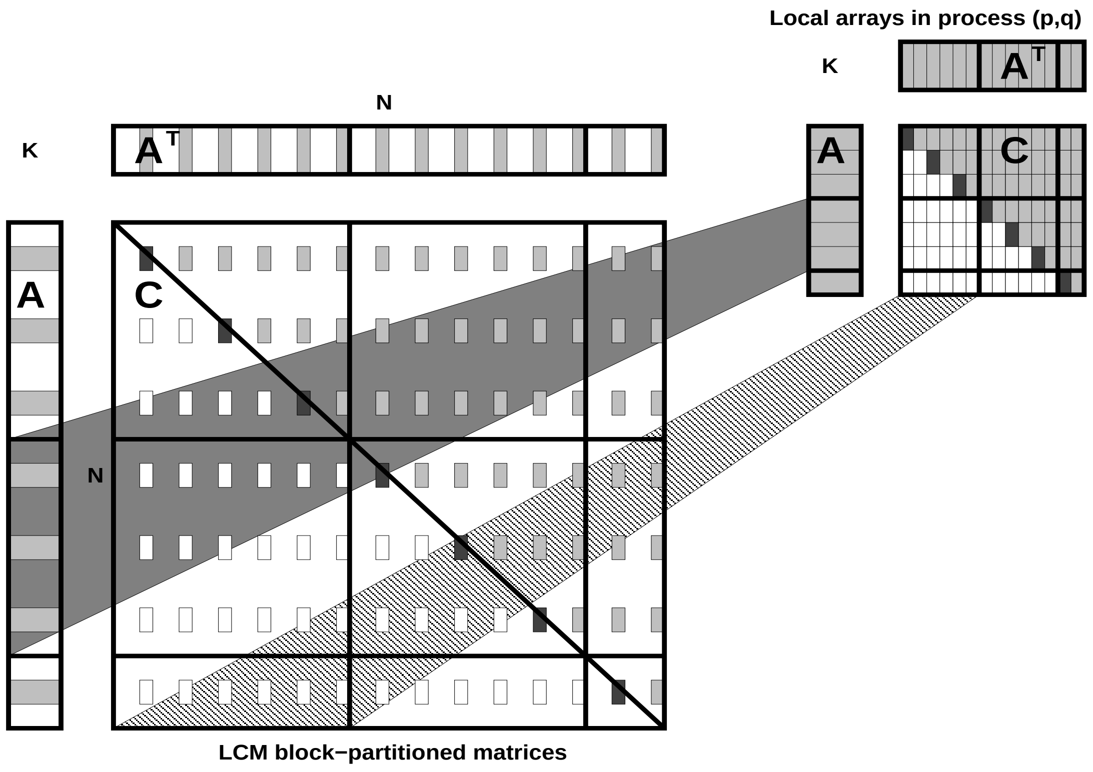


Figure 3.4: Static symmetric rank-K update

diagonal, the $r \times s$ blocks of C are identified by a different shade of color. It is usually easy to deal with the strict upper or lower part using the BLAS matrix-matrix multiply. The diagonal LCM block requires however particular attention. Two options are possible. First, one copies the part to be referenced into a buffer padding the rest of this buffer with zeros. It is then possible to issue one single call to the adequate BLAS kernel to deal with this block as well. More floating point operations than needed are obviously performed and some workspace is required. The amount of workspace available triggers how much of the LCM block can be handled in a single BLAS operation. It may also be necessary to copy back the meaningful part of this buffer when the triangular or symmetric distributed matrix is an output parameter. This strategy is exactly what is done in the current version of the ScaLAPACK software library [16, 25]. There is little evidence, however, demonstrating the superiority of this strategy over the second option. The latter avoids the unnecessary floating point operations as well as the data copy, and thus does not require any workspace to store these diagonal LCM blocks. To do so, each $r \times s$ block owning diagonals of such a diagonal LCM block is treated separately. The other blocks strictly below or above the diagonals are grouped together for the computations. Still, it is likely that more BLAS calls will be issued on smaller matrix operands. In both options described above, it is necessary to locate the $r \times s$ blocks owning the diagonals of C. This can be achieved by using the LCM tables described in the previous chapter.

The static blocking strategy, even in its simplest form, imposes strong restrictions on the alignment and distribution of the operands. This is, nevertheless the last opportunity for a large operation to rearrange the computations. This suggests the development of two sets of building blocks. The first one contains appropriate operations for dealing with *trapezoidal- symmetric and triangular* matrix blocks. It is referred to as the trapezoidal BLAS. The second set is comprised of operations specifically designed to manipulate the diagonal LCM blocks. It is referred to as the LCM BLAS. Both of these sets of building blocks are BLAS extensions.

The upper trapezoidal symmetric rank-K update operation is illustrated in Figure 3.5. From this figure, it is fairly easy to generalize the derivation of other

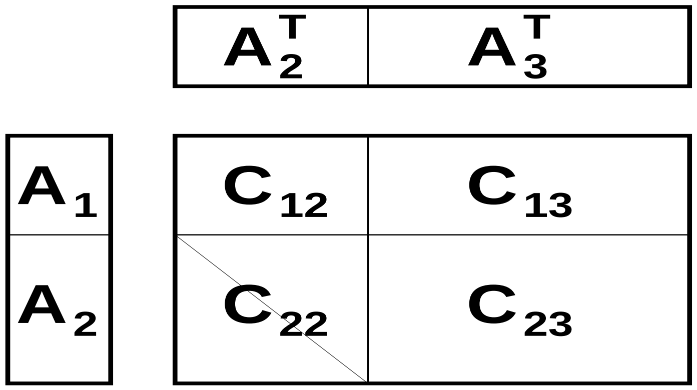


Figure 3.5: Trapezoidal symmetric rank-K update ($C_{22} = C_{22}^T$)

basic trapezoidal operations. The trapezoidal symmetric rank-K update shown in Figure 3.5 can be solely expressed in terms of BLAS operations.

$$\begin{cases} [C_{12}C_{13}] & \leftarrow [C_{12}C_{13}] & + A_1\,[A_2^T A_3^T] & \text{(matrix}\Leftrightarrow\text{multiply)} \\ C_{22} & \leftarrow C_{22} & + A_2\,A_2^T & \text{(symmetric rank}\Leftrightarrow K \text{ update)} \\ C_{23} & \leftarrow C_{23} & + A_2\,A_3^T & \text{(matrix}\Leftrightarrow\text{multiply)} \end{cases}$$

The Levels 2 and 3 BLAS dealing with symmetric and triangular matrices can be extended to the trapezoidal cases. Native implementations of the latter can take advantage of better data reuse than what is suggested in the above pseudo code.

Or, a mitigated improvement could be achieved by using low overhead BLAS kernels.

In order to handle diagonal LCM blocks, one needs to inspect the LCM table for the blocks owning the diagonals by using the Properties (2.5.20), (2.5.1), (2.5.2), (2.5.3) and (2.5.4). These blocks are trapezoidal blocks. At this point, it is remarkable that the size of an LCM block is irrelevant and can be replaced by any logical blocking factor NB_{log}. It is now possible to express a statically blocked symmetric rank-K update in terms of these basic LCM operations. The pseudo code for the upper case is presented below.

```
for    $i = 1,\ N,\ NB_{log}$
       $ib = \min(N \Leftrightarrow i + 1,\ NB_{log});$
       $C(i : i + ib \Leftrightarrow 1, i : i + ib \Leftrightarrow 1) \leftarrow C(i : i + ib \Leftrightarrow 1, i : i + ib \Leftrightarrow 1) +$
                 $A(i : i + ib \Leftrightarrow 1, :)\, A(i : i + ib \Leftrightarrow 1, :)^T;$
       $C(i : i + ib \Leftrightarrow 1, i + ib : N) \leftarrow C(i : i + ib \Leftrightarrow 1, i + ib : N) +$
                 $A(i : i + ib \Leftrightarrow 1, :)\, A(i + ib : N, :)^T;$
end    for
```

This pseudo-code suggests a global index interface for the LCM BLAS similar to the one described in [18] for a set of parallel BLAS. Finally, it is possible to reuse existing serial GEMM-based implementations of the Levels 2 and 3 BLAS [33, 60].

3.4.2 Cyclic Ordering

The *cyclic ordering* strategy is distinguished by the fact that the computations are cyclically distributed as opposed to the data. The block cyclic data distribution allocates the data in a cyclic fashion. The computation then proceeds in consecutive order just like a conventional serial algorithm. For example, the usual LU factorization algorithm [49] handles first the first column of the matrix, then the second and so on. The dual of this framework can be described as follows. First, the data is allocated or distributed in consecutive order, i.e., according to the blocked distribution defined in (2.2.6). Second, the computation proceeds in cyclic fashion. This approach is called *cyclic ordering*. It has been used throughout the CMSSL library [81]. It is shown in [72] that block cyclic order elimination can be used effectively on distributed memory architectures to achieve load balance as an alternative to block cyclic data allocation. The Connection Machine system compilers were designed to use consecutive data allocation as a default, or blocked data distribution as defined in (2.2.6). Thus, the designers of the CMSSL library chose to use cyclic order elimination to achieve good load balance. As suggested at the beginning of this section cyclically ordered algorithms proceed differently than the equivalent serial algorithms. It has been therefore necessary to develop these cyclic elimination algorithms, without being able to reuse much of the existing software. Nevertheless, this approach has been proven to lead to efficient and scalable parallel algorithms [72].

The rank-K update operation [74] can easily be expressed within such a framework. The matrices A, B and C are assumed to be allocated in consecutive order over the process grid, that is blocked distributed. The pseudo code is presented below.

```
for    $kk = 1,\ K,\ NB_{log}$
       $kb = \min(K \Leftrightarrow kk + 1,\ NB_{log})$;
       Broadcast $A(:, kk : kk + kb \Leftrightarrow 1)$ within process rows;
       Broadcast $B(kk : kk + kb \Leftrightarrow 1, :)$ within process columns;
       $C \leftarrow C + A(:, kk : kk + kb \Leftrightarrow 1) * B(kk : kk + kb \Leftrightarrow 1, :)$;
end  for
```

This algorithm is extremely efficient for three reasons. First, the logical blocking factor NB_{log} can be empirically chosen to be optimal for a given hardware platform. Second, it is possible to pipeline the communication phases in both dimensions of the process grid. Finally, a given process broadcasts all of its columns of A or rows of B before its east or south neighbor broadcasts. The communications can then be "perfectly" pipelined, that is, at all stages of the pipeline some computation is performed. In theory, communications can be completely overlapped with computations. When a process broadcasts its last piece, some attention is required to maintain the communication pipelines, but that is a minor and solvable detail. For more complicated operations such as a triangular solve or the LU factorization with partial pivoting, it is necessary to permute the cyclically ordered output into its original consecutive order. This somewhat complicates the hierarchical design of building blocks. However, the restrictions on the data layout simplifies the specification of these permutation operations. Misaligned data can occur, but redistributing from one specific data decomposition into itself can be achieved by simple and efficient algorithms. Finally, the cost of designing new algorithms based on cyclic elimination should be weighted against the gains in terms of simplicity and efficiency for the compilers, operating and run-time systems. This approach seems to be one of the most reasonable and viable software designs if one wants to develop the entire software collection that a given hardware platform needs to be operational.

3.4.3 Physical Blocking

The *physical blocking* strategy uses the distribution blocking factors as a unit for the computational blocks. In other words, the computations are partitioned accordingly to the data distribution. The blocks used to decompose the matrix are the same as those used to partition the computation. No attempts are made to either gather rows or columns residing in distinct processes, or scatter rows or columns residing in a single process row or column. It is assumed that the distribution parameters have been determined a priori presumably by the user.

Optimally, the latter should take into account the implications of physical blocking. This strategy is used in most of the parallel algorithms presented in the literature [2, 7, 9, 10, 26, 27, 39, 47, 48, 52, 57, 69, 70, 85].

The rank-K update operation shown in Figure 3.6 is based on a physical blocking strategy and is relatively easy to express. Just as for the static blocking strategy, strong alignment and distribution assumptions are made on the matrix operands. The pseudo code is almost the same as the one given above for cyclic ordering. The only modification to be made is to replace the logical blocking factor NB_{log} by the physical blocking factor used to decompose the columns of

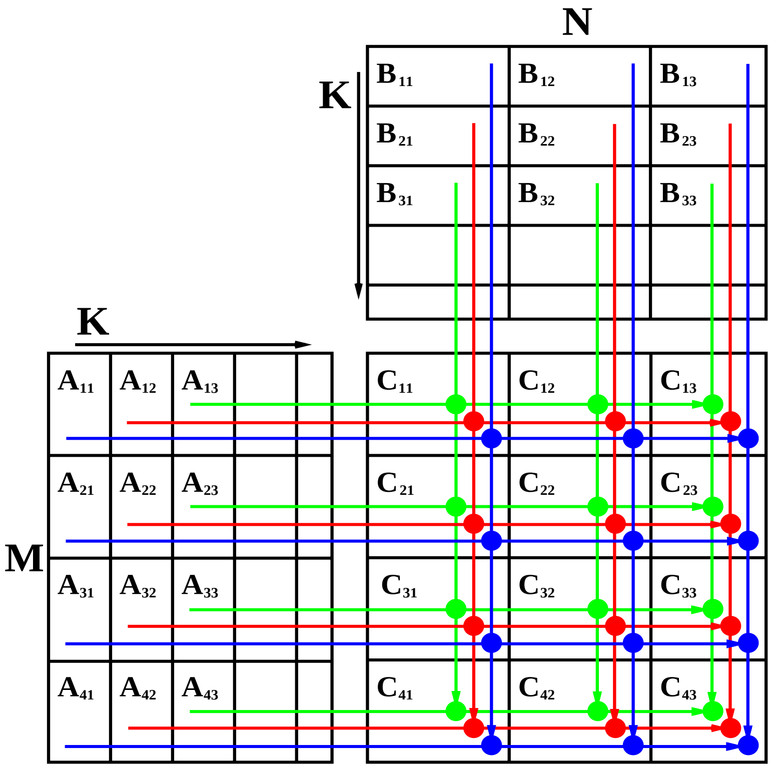


Figure 3.6: Physically blocked rank-K update

A and the rows of B. Similarly as in the cyclic ordering algorithm, it is possible to take advantage of communication pipelines in both directions of the process grid. However the cyclic data allocation imposes that the source process of the broadcasts changes at each iteration in a cyclic fashion. That is, a given process broadcasts all of its columns of A or rows of B in multiple pieces of size proportional to the value of the physical blocking factor. The smaller this value is, the larger the number of messages and the lower the possible data reuse during the following computational phase. In other words, the performance degrades as the value of the physical blocking factor is decreased. If the value of this factor is very large, the communication computation overlap decreases causing a performance degradation. Moreover, the stages of the communication pipelines are longer, and the pipeline startup cost is larger than for cyclic ordering. This is because the processes issuing the broadcasts are the south and east neighbors of the processes that have issued the previous broadcasts. These source processes change at every iteration of the loop. High performance and efficiency can still be achieved for a wide range of different values of the blocking factors. This has been reported in [2, 38, 74, 83].

The use of physical blocking in conjunction with static blocking can lead to a comprehensive and scalable dense linear algebra software library. Existing serial software such as LAPACK [5] can be reused. The ScaLAPACK software library is the result of this reasoning. As suggested above, if one limits oneself to static and physical blocking, strong alignment restrictions must be met by the matrix operands. It is argued that these restrictions are reasonable because, first, general redistribution software is available. Second, the user is ultimately responsible for choosing the initial data layout. Finally, the majority of practical cases are covered by this approach.

3.4.4 Aggregation and Disaggregation

The *aggregation or algorithmic blocking* strategy operates on a panel of rows or columns that are globally contiguous. The local components of this panel before aggregation are also contiguous. The size of this panel is a logical blocking unit factor that depends on the target machine characteristics. If this logical value is equal to the physical distribution blocking factors, then algorithmic and physical blocking are the same. Otherwise, a few rows or columns which are globally contiguous and residing in distinct processes, are aggregated into a single process row or column and this panel becomes the matrix operand. This strategy is required for efficiency if the physical blocking factor is so small that Level 3 BLAS performance cannot be achieved locally on each process. Obviously, the aggregation phase induces some communication overhead. However, this must be weighted against the local computational gain. The problem is then to determine a logical blocking factor NB_{log} that keeps this overhead as low as possible and simultaneously optimizes the time spent in local computation. The feasibility and performance characteristics of this approach have been illustrated for the numerical resolution of a general linear system of equations and the symmetric eigenproblem in [12, 13, 53] for the purely scattered distribution as defined in (2.2.7). Similarly, it is sometimes beneficial to disaggregate a panel into multiple panels in order to overlap communication and computation phases. When applicable, this last strategy also presents the advantage of requiring a smaller amount of workspace. The pseudo code of the rank-K update operation using aggregation follows.

```
for    $kk = 1,\ K,\ NB_{log}$
       $kb = \min(K \Leftrightarrow kk + 1,\ NB_{log})$;
       Aggregate $A(:, kk : kk + kb \Leftrightarrow 1)$ in one process column;
       Broadcast $A(:, kk : kk + kb \Leftrightarrow 1)$ within process rows;
       Aggregate $B(kk : kk + kb \Leftrightarrow 1, :)$ in one process row;
       Broadcast $B(kk : kk + kb \Leftrightarrow 1, :)$ within process columns;
       $C \leftarrow C + A(:, kk : kk + kb \Leftrightarrow 1) * B(kk : kk + kb \Leftrightarrow 1, :)$;
end  for
```

The aggregation and disaggregation techniques are attempts to address the cases where the physical blocking strategy is not very efficient, i.e., for very small or large distribution blocking factors. In both techniques, the consecutive order of matrix columns or rows is preserved. It is therefore possible to use both techniques for algorithms that feature dependent steps such as a triangular solve or the LU factorization with partial pivoting. The disaggregation technique however can only be applied efficiently for operations that do not feature any dependence between steps, such as a matrix-multiply. The disaggregated data remains consecutively ordered. Therefore, it cannot improve significantly the load imbalance caused by consecutive allocation and consecutive elimination [59].

3.4.5 LCM Blocking

The *LCM blocking* strategy operates on a panel of rows or columns that are locally contiguous. The size of this panel is also a logical blocking unit factor that depends on the target machine characteristics. However, one packs rows or columns that may not be locally contiguous according to an external criterion, typically the distribution parameter of another operand.

Consider the rank-K update operation illustrated in Figure 3.7. The LCM blocking strategy proceeds as follows. One is interested in finding the columns of A residing in a particular process column q and the rows of B residing in a particular process row p that could be multiplied together in order to update the matrix C. In Figure 3.7, these columns of A and rows of B are indicated in gray. To accomplish this, one can consider the virtual matrix, denoted VM in the figure, defined by the column distribution parameters of A and the row distribution parameters of B. Locating the 0-diagonals of this VM in the process of coordinates (p, q) exactly solves the problem as illustrated in the figure. This can be realized by using LCM tables as shown in Chapter 2. As opposed to the physical blocking strategy, this technique does not assume the distribution equivalence of the columns of A and rows of B as suggested in Figure 3.7. Moreover, the packing of these columns of A and rows of B is a local data copy operation, i.e., without communication overhead. For a given q, one just needs to go over all process rows and thus treat all of the columns of A residing in this process column q. This algorithm presents multiple advantages over the physically blocked version. First,

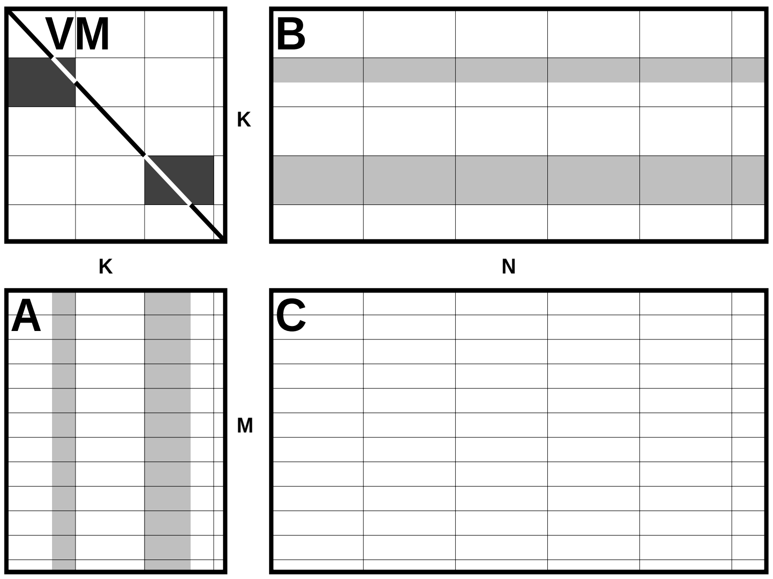


Figure 3.7: Global view of the LCM blocked rank-K update

as mentioned above, it does not assume an equivalent distribution of the columns of A and rows of B. Second, the communication overhead of the physically blocked variants has been partially replaced by a local data copy into a buffer that was needed anyway. The communication pipeline stages in the row direction have been shortened. The cost of this pipeline startup has also been reduced considerably by having the process column emitting the broadcasts remaining fixed as long as possible. Finally, one has the opportunity to overlap communications and computations in the process column direction as well. Indeed, the packing of the rows of B in the process row $(p+1)$ can be performed in advance, so that this communication pipeline is cheaper. This operation can also be logically blocked by limiting the number of columns of A in process column q and corresponding rows of B in the process row p that will be locally packed and broadcast.

This approach presents the advantage that the cost of aggregation phase is put on the processor as opposed to the interconnection network. However, it cannot be used for algorithms where each step depends on the previous one. Typically, LCM blocking is well-suited for multiplying two matrices, where each contribution to resulting matrix entries can be added in any order. To a certain extent solving a triangular system can take advantage of such a blocking strategy. The algorithm proceeds in an ordered sequence of steps that depend on each other. It is, however, possible to block this algorithm and and express it in terms of triangular solves and matrix multiplies [33, 60]. The LCM blocking strategy is a typical algorithmic redistribution operation since it rearranges logically and physically the communication and computation phases for increased efficiency and flexibility.

3.4.6 Aggregated LCM Blocking or Hybrid Schemes

The *aggregated LCM blocking* strategy is an hybrid scheme that combines the aggregation and LCM blocking strategies. In the aggregation scheme described earlier, the blocks to be aggregated were globally contiguous. It is, however, possible to use the same strategy for the local blocks obtained via LCM blocking. Furthermore, disaggregated LCM blocking is also possible as noted above.

3.5 Two-Dimensional Redistribution

The operations described in this section involve $M \times N$ two-dimensional distributed matrices as defined in (3.2.2). They generalize the one-dimensional redistribution technique presented earlier. Let A and B be such matrices. Let (P_A, r_A, Q_A, s_A) (respectively (P_B, r_B, Q_B, s_B)) be the distribution parameters associated with A (respectively B). In order to redistribute A into B, one considers the DVM induced by the columns of A and rows of B, as well as the one induced by the rows of A and columns of B. Figure 3.8 shows the block-partitioned operands A and B as well as the induced DVMs $VM1$ and $VM2$. For a given row of A, the corresponding diagonal entry of the distributed virtual matrix $VM2$ determines the corresponding columns of B. A block of rows of A that could be packed together is represented in the figure by a gray rectangle. The corresponding diagonal block of the distributed virtual matrix $VM2$ and the corresponding columns of B are colored with the same gray. The same reasoning is applied to the columns of A and rows B using the distributed virtual matrix $VM1$. The gray intersections in A determines the entries that could be packed together in one message. Figure 3.8 illustrates the importance of locating the diagonals in the context of two-dimensional redistribution. As opposed to the one-dimensional

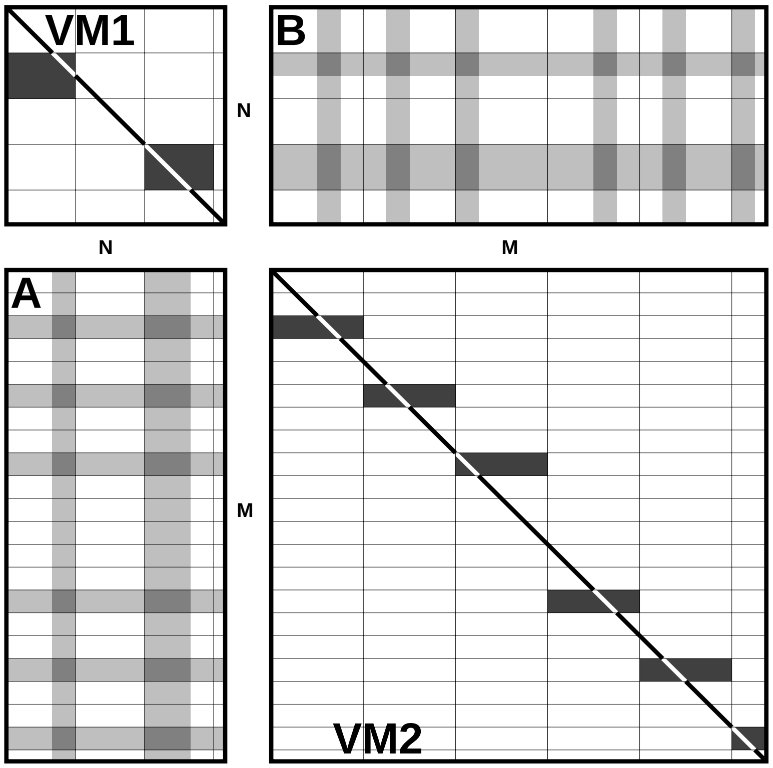


Figure 3.8: Global view of two-dimensional redistribution

case, there are two virtual process grids defined by P_A, Q_B and Q_A, P_B. If a process of coordinates (p_A, q_B) owns diagonal entries of the associated $M \times M$ virtual matrix $VM2$, and if a process of coordinates (p_B, q_A) owns diagonal entries of the associated $N \times N$ virtual matrix $VM1$, then this says that some data residing in process (p_A, q_A) should be sent to the process of coordinates (p_B, q_B). The blocks of the DVMs $VM1$ and $VM2$ owning diagonals are represented by darker rectangles in which the diagonals are symbolized by a white segment. The knowledge of the source process coordinates allows the receiving process to calculate a priori the size of the message to be received as well as its packed form and the location of each message entry. Consequently, unless A and B are equivalent in the sense of (3.2.12), the number of messages to be exchanged is equal to the product of the number of processes owning diagonals in the DVMs $VM1$ and $VM2$. This approach using LCM tables allows one to express the copy and transpose operations in a single framework. The implementation of such operations should take advantage of such a savings opportunity. In addition, since it is possible to detect via a simple test on the distribution parameters distribution equivalence, this scheme can easily be made optimal for the simpler cases. Note that the operands A and B could be distributed on distinct process grids without affecting the packing strategy induced by the LCM tables. Moreover, if one uses the LCM table definition handling a partial first block, this scheme naturally supports non-aligned operands. Finally, it is straightforward to handle replicated operands by taking it into account when computing the LCM table entries. The next chapter will discuss the possible communication patterns and their complexity associated with the two-dimensional redistribution just described as well as their complexity. Pseudo algorithms will also be presented.

3.6 Conclusions

This chapter summarized different blocking strategies for block cyclic mappings. It also introduced original LCM techniques extending the physical blocking scheme. These LCM techniques allow for greater flexibility. They are also equivalent to the usual techniques for the restricted cases. The presentation of these general techniques stressed their systematic derivation from the properties of the underlying mapping. The importance of the LCM tables introduced in Chapter 2 has been discussed and shown to provide an acceptable and convenient framework to present algorithmic redistribution operations. The latter form the elementary building blocks to express more complex parallel operations such as a complete, efficient and flexible set of parallel linear algebra operations. Four categories of operations naturally emerge from the previous discussion:

- Statically blocked computational operations,
- Aggregation kernels,
- LCM blocking tools,
- One and two-dimensional redistribution.

These basic buildings blocks are well delimited. They can all be expressed within a single framework using LCM tables. Such a partitioning is suitable for software library design.

Chapter 4

Performance Analysis

4.1 Introduction

This chapter presents a framework for quantifying the scalability of the algorithmic variants of the matrix-matrix multiplication presented in the previous chapter. This framework is used to assess the theoretical performance impact of the logical blocking factor NB_{log}. It is shown that under certain restrictions algorithmic blocking allows for high performance tuning. In addition, the relationship of NB_{log} with other machine and distribution parameters is addressed.

A theoretical model of a distributed memory computer is presented early in this chapter. It is an abstraction of physical models, and provides a convenient framework for developing and analyzing parallel distributed dense linear algebra algorithms without worrying about the implementation details or physical constraints. The model can be applied to obtain theoretical performance bounds on DMCCs or to estimate the execution time before or after the algorithm has been implemented. This abstract model is used in the context of scalability and programmability analysis. The machine model described and used in this chapter is a very crude approximation of reality. Its purpose is not to precisely reflect all the phenomena that occur during a general computation, but merely to identify the dominant costs relevant to dense linear algebra computations. The target architectures used for the experiments, as well as the machine parameters measured during the experiments, are presented after the machine model. They justify the reasonable approximations one can make when using the machine model. The application of the model to each of the blocking strategies presented in Chapter 3 allows for a characterization, evaluation and comparison of these blocking techniques. It is shown in this chapter that none of these strategies is clearly superior to its challengers. Instead, they are complementary. Independent of the distribution parameters or alignment of the matrix operands for the operation of interest, it is theoretically possible to use the machine resources at their best and achieve asymptotically comparable efficiency. As will be explained in the next chapter, however, the previous statement needs to be slightly refined in practice to accommodate physical memory size constraints as well as other factors.

4.2 The Machine Model

The DMCCs introduced in Chapter 1 consist of processors that are connected using a message passing interconnection network. Each processor has its own memory called the local memory, which is accessible only to that processor. As the time to access a remote memory is longer than the time to access a local one, such computers are often referred to as Non-Uniform Memory Access (NUMA) machines. Strictly speaking, a NUMA architecture differs from a message passing architecture in the sense that it provides hardware support for direct access to other processor's memories, whereas in a message passing architecture, remote access must be explicitly emulated via message passing [67].

The interconnection network of our machine model is static, meaning that it consists of point-to-point communication links among processors. This type of network is also referred to as a direct network as opposed to dynamic networks. The latter are constructed from switches and communication links. These links are connected to one another dynamically by the switching elements to establish at run time the paths between processors' memories. Furthermore, the interconnection network of the machine model considered here is a static two-dimensional $P \times Q$ rectangular mesh with wraparound connections as illustrated in Figure 4.1. In addition, it is assumed that all processors can be treated equally in terms of local performance and the communication rate between two processors is independent from the processors considered. Each processor in the two-dimensional mesh has four communication ports. However, the model assumes that a processor can send or receive data on only one of its ports at a time. This assumption is also referred to as the one-port communication model [67].

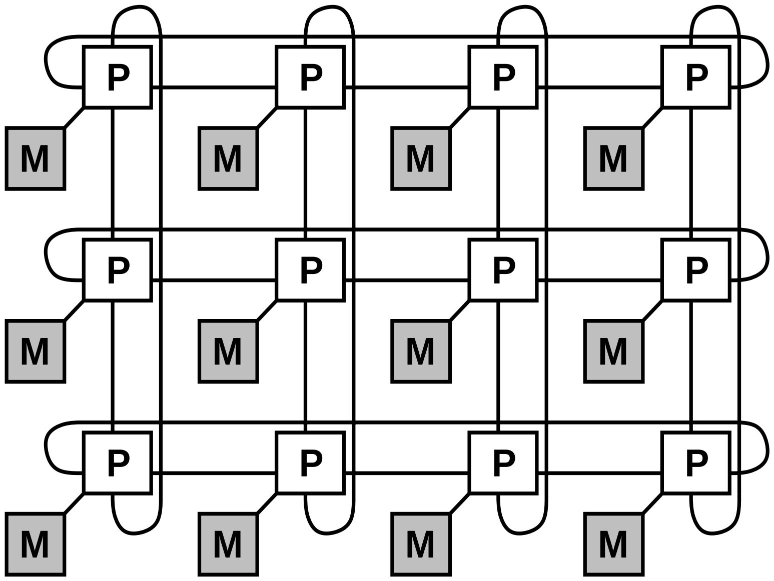


Figure 4.1: A 3×4 processor mesh with wraparound connections

The time spent to communicate a message between two processors is called the communication time T_c. In our machine model, T_c is approximated by a linear function of the number L of items communicated. T_c is the sum of the time to prepare the message for transmission α and the time βL taken by the message of length L to traverse the network to its destination, i.e.,

$$T_c = \alpha + \beta L.$$

This approximation of the communication time supposes that any two processors are equidistant from a communication point of view (cut-through or wormhole routing). For most current DMCCs, this approximation is reasonable. Finally, the model assumes that the communication links are bidirectional, that is, the time for two processors to send each other a message of length L is also T_c. A processor can send and/or receive a message on only one of its communication links at a time. In particular, a processor can send a message while receiving another message on the same or different link at the same time.

Since this dissertation is only concerned with a single regular local operation, namely the matrix-matrix multiplication, the time taken to perform one floating point operation is assumed to be a constant γ in our model. This very crude approximation summarizes in a single number all the steps performed by the processor to achieve such a computation. Obviously, such a model neglects all the phenomena occurring in the processor components, such as cache misses, pipeline startups, memory load or store, floating point arithmetic and so on, that may influence the value of γ as a function of the problem size for example. Similarly, the model does not make any assumption on the amount of physical memory per node.

This machine model is a very crude approximation that is designed specifically to illustrate the cost of the dominant factors to our particular case. More realistic models are described for example in [67] and the references therein.

4.3 Estimation of the Machine Parameters

Two DMCCs, namely the Intel XP/S Paragon and the IBM Scalable POWERparallel System, have been used in the experiments that have been performed for this dissertation. Both of these DMCCs differ in many aspects from the machine model. These differences are stressed to illustrate the crudeness of this model. The relevant performance characteristics of both computers are highlighted and commented. In addition to the information supplied by the manufacturers of these computers, the machine parameters have been measured as part of the experiments. The obtained results are presented below. It is convenient to present these technical features in order to assess the relative importance of each parameter.

As mentioned earlier, the rank-K update operation has been selected to illustrate the differences between all blocking strategies presented in Chapter 3. This operation globally produces an $M \times N$ matrix C by adding to itself the product of an $M \times K$ matrix A and a $K \times N$ matrix B

$$C \leftarrow C + A\,B.$$

The number of floating point operations required to perform this rank-K update operation is assumed to be equal to $2\,M\,N\,K$. It is important to notice that in this particular case, the global distributed operation is also the local operation performed by all processes. Locally in any given process, M (respectively N) is then the local number of rows (respectively columns) of the distributed matrix C contained in this process. Locally, K can be considered equal to NB_{log} as is almost always the case.

In our experimental implementation, the local rank-K operation is performed by calling the appropriate subprogram of the vendor-supplied BLAS. The communication operations are implemented by explicit calls to the Basic Linear Algebra Communications Subprograms (BLACS). The BLACS [37, 40] are a message passing library specifically designed for distributed linear algebra communication operations. The computational model consists of a one or two-dimensional grid of processes, where each process stores matrices and vectors. The BLACS include synchronous send/receive routines to send a matrix or submatrix from one process to another, to broadcast submatrices, or to compute global reductions (sums, maxima and minima). There are also routines to establish, change, or query the process grid. The BLACS provide an adequate interface level for linear algebra communication operations.

4.3.1 The Intel XP/S Paragon

The processing units of the Intel XP/S Paragon are nodes, based on the Intel's i860 XP RISC processors. Each processor is capable of a peak performance of 75 Mflops. Multiprocessor (MP) nodes have three i860 XP processors - two to execute application code and a third for use as either a message coprocessor or as an application processor. General-purpose (GP) nodes are also available. Those nodes have two XP application processors - one dedicated to applications and the other to message-passing [31]. Figure 4.2 shows the GP node performance of the vendor supplied matrix-matrix multiply library routine for distinct values of $M = N$ and K. In practice, the performance of such an operation was observed to be at most 45 Mflops for those GP nodes. The machine used for our experiments

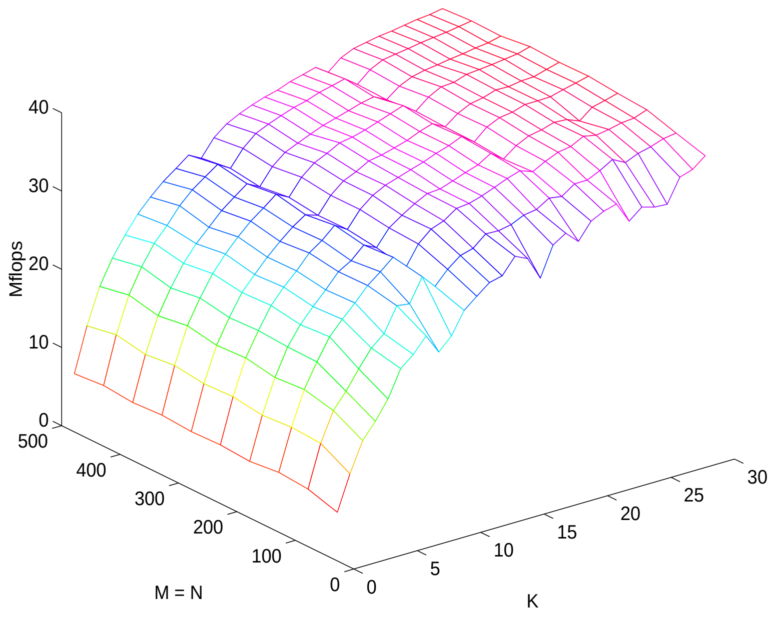


Figure 4.2: Performance of the rank-K update on one processor of the Intel Paragon

was primarily comprised of GP nodes having 16 MB of physical memory. Figure 4.2 shows that local performance is very sensitive to small values of $K = NB_{log}$. However, when the local value of $K = NB_{log}$ is large enough, the local performance of the rank-K update is almost constant. Figure 4.3 shows the performance degradation that occurs when the matrix operands do not fit in core, i.e., when the operating system begins swapping. This figure illustrates that the use of virtual memory may cause a large performance decrease. In other words, from the local computational point of view, very large values of $K = NB_{log}$ should also be avoided.

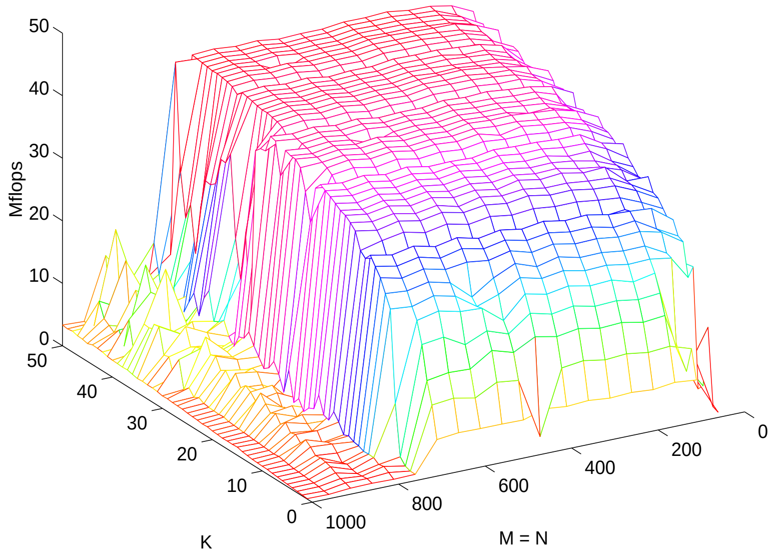


Figure 4.3: Performance of the rank-K update on one processor of the Intel Paragon

On the Intel XP/S, the actual transmission of messages is performed by an independent routing system of Mesh Router Components (MRCs), one for each node, arranged in a two-dimensional mesh. These fixed- function devices route messages between any two nodes in the system at hardware speeds of up to 175 MB/s. Hardware latency - the time to set up the transfer of the first byte of a message - is so low (40 ns per MRC traversed) that the physical location of nodes becomes unimportant for performance [31].

During our experiments, the performance of the BLACS communication primitives implemented on top of the native Intel XP/S message passing library was measured. For the message sizes relevant to our application, we found that the values of α and β that best approximate in the least-square sense the communication time $T_c = \alpha + \beta\, L$ are given by $\alpha \approx 60.0\ \mu$s and $\beta^{-1} \approx 70.0$ MB/s. The overhead induced by the BLACS primitives on this system compared to the native Intel message passing library is negligible as shown earlier in [87].

4.3.2 The IBM Scalable POWERparallel System

The IBM Scalable POWERparallel System, or SP, consists of nodes (processors with associated memory and disk) connected by ethernet and a high-performance switch. The processors are POWER2 architecture RS/6000 processors, which are superscalar pipelined chips capable of executing four floating point operations per cycle. The clock speed of this processor is 66.7 MHz, giving a peak performance per processor of 266 MFLOPS. There are two types of nodes, known as *thin* nodes and *wide* nodes. Thin nodes have a 64 KB data cache. Wide nodes have 256 KB data cache [4, 30, 80]. This data cache size difference results in slower performance on thin nodes for computationally intensive applications. The machine used for our experiments consisted of thin nodes exclusively having 128 MB of physical memory [4, 30, 80]. Figure 4.4 shows the thin node performance of the vendor supplied rank-K update library routine for distinct values of $M = N$ and K. In practice, the performance of such an operation was observed to be at most 200 Mflops for those thin nodes. Figure 4.4 also shows that the local performance of the rank-K operation is sensitive to small values of $K = NB_{log}$. However, when the local value of $K = NB_{log}$ is large enough, the local performance is almost constant.

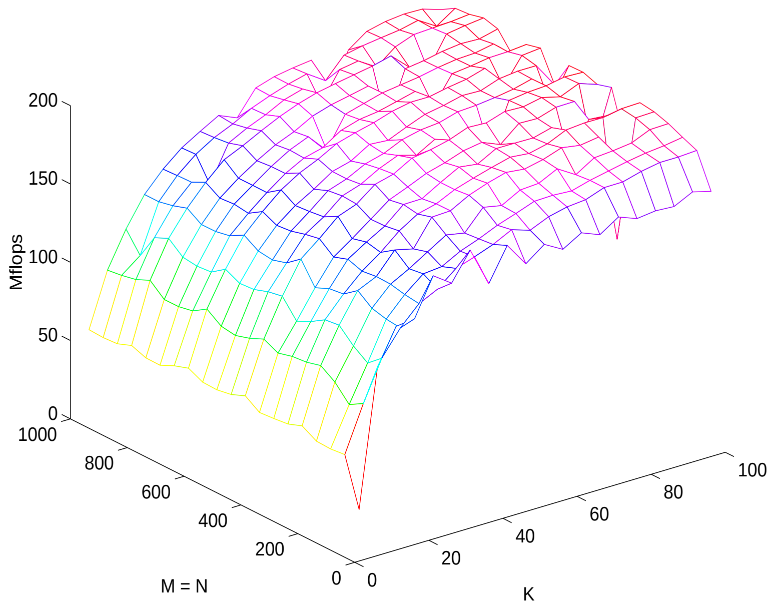


Figure 4.4: Performance of the rank-K update on one processor of the IBM SP2

The interconnection network of the IBM SP is a two-level crossbar switch. The TB2 switch adapter, which is the interface between the node and the switch, features a Direct Memory Access (DMA) engine. For message passing libraries optimized for the switch, the typical bandwidth is 35 MB/s with a latency of approximately 50 μs [4, 30, 80].

During our experiments, the performance of the BLACS communication primitives implemented on top of the native IBM message passing library (MPL) was measured. For the message sizes relevant to our application, we found that the values of α and β that best approximate in the least-square sense the communication time $T_c = \alpha + \beta\, L$ are given by $\alpha \approx 400.0\ \mu$s and $\beta^{-1} \approx 28.0$ MB/s. Comparing the BLACS performance versus the native IBM message passing library requires a more detailed explanation and can be found in [87].

4.4 Performance Analysis

In this section the machine model defined above is applied in turn to each blocking strategy presented in Chapter 3. The three matrix operands A, B and C are considered to be square of order $M = N = K$. The distributed matrix A (respectively B and C) is partitioned into $r_A \times s_A$ (respectively $r_B \times s_B$ and $r_C \times s_C$) blocks. All three matrices are distributed onto the same $P \times Q$ process grid. When the distributions of the rows of the matrix operands A and C are equal in the sense of Definition 3.2.11, and the distributions of the columns of the matrix operands B and C are equal, and the distributions of the columns of the matrix operand A and the rows of the matrix operand B are equal, we say that the matrix operands are "aligned" for the rank-K update operation. In this case, we say that the operation is aligned, meaning that the operation is performed on aligned data. Otherwise the operation is said to be “non-aligned”. The major difference between the aligned and non-aligned rank-K operations is the fact that the matrix operands A and B must be redistributed before the aligned operation can take place. Two strategies are possible. Either both A and B are redistributed at once and the physical blocking variant is then used to finish the computations. This strategy is called RED thereafter. Or, the redistribution of A and B is interleaved with partial rank-K updates. In this case, a panel of at most NB_{log} columns of A and at most NB_{log} rows of B are formed using either the physical blocking strategy (PHY), (dis)aggregation (AGG) or the LCM blocking strategy (LCM). These panels are then shifted if needed using the algorithm described in Section 3.3 and a rank-NB_{log} update is performed. Since the blocking strategy uniquely identifies the interleaving policy, we use these three identifiers PHY, AGG and LCM to refer to the corresponding algorithms. In summary, four algorithms are considered, denoted by PHY, AGG, LCM and RED. Depending on the initial distribution of the operands considered, the operation may or may not be aligned.

When the operation is not aligned, the matrix operands A and B are redistributed. The communication volume associated to a given operation is the total length of all messages performed by that operation. The additional communication volume associated with the non-aligned rank-K update is therefore equal to $2\,N^2$. When the operation is aligned and the panels of A and B have been constructed, it is necessary to replicate the panel of columns of A in all process columns, and the panel of rows of B in all process rows as shown in Figure 3.6. The volume of communication associated with this operation is given by $N^2\,(Q/P + P/Q)$. When the process grid is square, this volume of communication becomes $2\,N^2$. Consequently, the non-aligned operation roughly doubles the communication volume, whereas the amount of computation remains the same.

The number of floating point operations that have to be performed to update one entry of the matrix operand C is equal to $2\,N + 1$. The load imbalance of the rank-K update can then be bounded above by the difference of the largest number of entries of C owned by each process and the smallest number of entries of C owned by each process. This number is given by

$$r_C\, s_C\,(\lceil\frac{\lceil\frac{M}{r_C}\rceil}{P}\rceil\lceil\frac{\lceil\frac{N}{s_C}\rceil}{Q}\rceil \Leftrightarrow \lfloor\frac{\lceil\frac{M}{r_C}\rceil}{P}\rfloor\lfloor\frac{\lceil\frac{N}{s_C}\rceil}{Q}\rfloor)$$

It follows that the load imbalance for this operation is in general proportional to the product of r_C by s_C. This suggests that very large distribution blocking factors of C are likely to induce a large load imbalance of the computations.

The following sections estimate the execution time of the different redistribution and blocking variants on our machine model as a function of the local computational speed (γ), the communication time parameters (α and β), and finally the total number of processes $p = P \times Q$.

An important performance metric is *parallel efficiency*. Parallel efficiency,

$E(n,p)$, for a problem of size n on p processors is defined in the usual way [46] as

$$E(n,p) = \frac{1}{p}\frac{T_{\text{seq}}(n)}{T(n,p)} \tag{4.4.1}$$

where $T(n,p)$ is the runtime of the parallel algorithm, and $T_{\text{seq}}(n)$ is the runtime of the best sequential algorithm. An implementation is said to be *scalable* if the efficiency is an increasing function of n/p, the problem size per processor (in the case of dense matrix computations, $n = N^2$, the number of words in the input). We will also measure the *performance* of our algorithm in Mflops/s (or Gflops/s). This is appropriate for large dense linear algebra computations since floating point dominates communication. For a scalable algorithm with n/p held fixed, we expect the performance to be proportional to p.

4.4.1 Physical Blocking

In this section, the performance analysis of the physical blocking strategy for aligned operands is presented. A similar analysis can also be found in [2, 83]. The reason for reproducing it hereafter is that it considerably simplifies the presentation of the performance analysis for the aggregation and LCM blocking strategies. For the sake of simplicity, the underlying process grid is assumed to be a $\sqrt{p} \times \sqrt{p}$ square mesh of p processes. In addition, the partitioning unit of the matrix operands is considered to be a square of size $NB_{dis} \times NB_{dis} (r = s)$. The matrix operands are also considered to be $N \times N$ square matrices. These assumptions only simplifies the expression of the performance analysis without modifying its consequences. One could easily derive a more detailed analysis if needed. All of our experiments were performed in double precision arithmetic. On both of our testing platforms, a double precision real is 8 bytes long. Thus, the bandwidth of the machine model is more conveniently expressed in double precision real per second. In the following, β_d denotes $8\,\beta$.

The key-factor of this performance analysis is to model the cost of a sequence of b broadcasts of messages of length n among $\sqrt{p}$ processes. The cost of the sequence of b minimum spanning tree broadcasts is given by

$$b\,\log_2(\sqrt{p})\,(\alpha + n\,\beta_d).$$

During the last step of a minimum spanning tree broadcast, each of the $\sqrt{p}$ processes is sending or receiving a message of length n. There is thus no opportunity to pipeline the messages and overlap the communications. Moreover, the source process of each broadcast does not influence the total cost of the broadcast sequence since as mentioned above the minimum spanning tree algorithm synchronizes all processes involved in the operation.

A more cost effective algorithm to perform such a broadcast sequence is to use "ring" broadcasts. In this case, however, the source process of each broadcast has an impact on the overall estimated execution time. If the source process of each broadcast remains the same, the estimated execution time of this operation is given by

$$(\alpha + n\,\beta_d)\,(\sqrt{p} \Leftrightarrow 1) + (b \Leftrightarrow 1)\,(\alpha + n\,\beta_d).$$

The first term of this expression is referred to as the startup time of the communication pipeline. For sufficiently large values of b, the startup time becomes negligible. The execution time of such an operation can then be approximated by the second term of the above expression. It follows that the sequence of b ring broadcasts is more efficient than the sequence of b minimum spanning tree broadcasts.

When the source process of each ring broadcast is the process following the source process of the previous broadcast, the cost of the sequence of b ring broadcasts becomes

$$(\alpha + n\,\beta_d)\,(\sqrt{p} \Leftrightarrow 1) + (b \Leftrightarrow 1)\,2\,(\alpha + n\,\beta_d).$$

More generally, if the source process of each ring broadcast is the k$^{\text{th}}$ process on the ring following the source process of the previous broadcast, the cost of the sequence of b ring broadcasts is given by

$$(\alpha + n\,\beta_d)\,(\sqrt{p} \Leftrightarrow 1) + (b \Leftrightarrow 1)\,(k + 1)\,(\alpha + n\,\beta_d). \qquad (4.4.2)$$

In the physical blocking strategy, the source process of the broadcast sequence is incremented at each step. In addition, the physical distribution blocking factor $NB_{dis} = s_A = r_B$ is used to partition the communication and computation. Therefore the execution time of this operation on the machine model is given by

$$\begin{aligned} T_{PHY}(N,p) = \; & 2\,(\alpha + \frac{NB_{dis}\,N}{\sqrt{p}}\,\beta_d)\,(\sqrt{p} \Leftrightarrow 1) + \\ & 4\,(\frac{N}{NB_{dis}} \Leftrightarrow 1)\,(\alpha + \frac{NB_{dis}\,N}{\sqrt{p}}\,\beta_d) + 2\,\frac{N^3\,\gamma}{p} \\ \approx \; & \frac{2\,N^3\,\gamma}{p}\,(1 + \frac{2}{\gamma}(\frac{p\,\alpha}{NB_{dis}\,N^2} + \frac{\sqrt{p}\,\beta_d}{N})) \text{ when } \frac{N}{NB_{dis}} \gg \sqrt{p}. \end{aligned}$$

The parallel efficiency of the physical blocking variant is then given by

$$E_{PHY}(N,p) = (1 + \frac{2}{\gamma}(\frac{p\,\alpha}{NB_{dis}\,N^2} + \frac{\sqrt{p}\,\beta_d}{N}))^{-1} \text{ when } \frac{N}{NB_{dis}} \gg \sqrt{p}.$$

The physical blocking algorithm is thus *scalable* in the sense that if the memory use per process $(\frac{p}{N^2})$ is maintained constant, this algorithm maintains efficiency. The last equality shows that the physical block size NB_{dis} can be used to lower the importance of the latency α.

4.4.2 Aggregation

In this section, the performance analysis of the aggregation blocking strategy for aligned operands is presented. This technique essentially performs a sequence of accumulations followed by a ring broadcast. For the sake of simplicity, it is assumed that k blocks of the same size are aggregated. In practice, the blocks are only approximately of the same size. It is clear that k is bounded above by $\sqrt{p}$. In addition, the logical blocking factor NB_{log} is used to partition the communication and computation. It follows from Equation 4.4.2 that the estimated execution time on our machine model for the aggregation strategy is given by

$$T_{AGG}(N,p) \approx \frac{2\,N^3\,\gamma}{p}\,(1+\frac{k}{\gamma}\,(\frac{p\,\alpha}{NB_{log}\,N^2}+\frac{\sqrt{p}\,\beta_d}{N}))\text{ when }\frac{N}{NB_{log}} \gg \sqrt{p}.$$

This result generalizes the result obtained above for the physical blocking strategy. The parallel efficiency of the aggregation variant is thus given by

$$E_{AGG}(N,p) = (1+\frac{k}{\gamma}(\frac{p\,\alpha}{NB_{log}\,N^2}+\frac{\sqrt{p}\,\beta_d}{N}))^{-1}\text{ when }\frac{N}{NB_{log}} \gg \sqrt{p}.$$

Consequently, the aggregation algorithm is also *scalable*. The value of k is a constant that only depends on the ratio between the logical NB_{log} and physical NB_{dis} blocking factors. These formula show the communication overhead induced by the aggregation strategy in terms of the number of messages as well as the communication volume. When the physical blocking factor is larger than the logical blocking factor, the physical blocks are split into smaller logical blocks. Therefore, the estimated execution time of the disaggregation variant is bounded above by the result obtained for the aggregation strategy.

4.4.3 LCM Blocking

In the LCM blocking strategy, one looks at the diagonals of the virtual distributed matrix induced by the columns of A and rows of B residing in all process column and row pairs. It is assumed in this section that each process in the grid owns a number of diagonals that is proportional to NB_{log}. With these assumptions, the estimated execution time of the LCM blocking strategy is given by

$$T_{LCM}(N,p) = \frac{2\,N^3\,\gamma}{p}\,(1 + \frac{3}{2\,\gamma}(\frac{p\,\alpha}{NB_{log}\,N^2} + \frac{\sqrt{p}\,\beta_d}{N})) \text{ when } \frac{N}{NB_{log}} \gg \sqrt{p}.$$

The parallel efficiency is thus

$$E_{LCM}(N,p) = (1 + \frac{3}{2\,\gamma}(\frac{p\,\alpha}{NB_{log}\,N^2} + \frac{\sqrt{p}\,\beta_d}{N}))^{-1} \text{ when } \frac{N}{NB_{log}} \gg \sqrt{p}.$$

Our machine model assumes that the local data copy operation is free. In reality, such an assumption is reasonable. The cost associated to the local data copy performed by the LCM blocking variant is negligible when compared to the communication time. It follows from the two preceding formula that the LCM blocking variant is also scalable for aligned matrix operands. This variant is slightly more efficient than the physical and aggregation strategies.

4.4.4 One Dimensional Redistribution

When the matrix operands A and B are not aligned with the matrix C, it is necessary to redistribute the matrices A and B. In the physical blocking, aggregation and LCM blocking strategies, the matrix operands A and B are redistributed by panels of global size $N \times NB_{dis}$ or $N \times NB_{log}$. In the redistribution of a single panel, $\sqrt{p}\,(\sqrt{p} \Leftrightarrow 1)$ messages are exchanged. In our case, this redistribution phase is immediately followed by a broadcast. Thus, we choose to perform this operation on two process columns or rows in order to limit the link contention. The message scheduling policy used in our one-dimensional redistribution operation is the "caterpillar" algorithm, where the messages are exchanged by pairs [78]. Other scheduling policies exist [67, 86]. These methods, however, do not feature a contention-free message scheduling policy as well as an optimal communication volume. Due to the simplicity of our machine model, it is not possible to correctly model the link contention of this redistribution operations. Therefore, the estimated execution time given below for this operation should be regarded as an approximation. In the context of the panel redistribution, $\sqrt{p}$ processes are involved. Each of them owns $\frac{N\,nb}{\sqrt{p}}$ data items, where nb is either NB_{dis} for the physical blocking strategy, or NB_{log} for the aggregation or the LCM variants. Each process sends and receives $\sqrt{p} \Leftrightarrow 1 \approx \sqrt{p}$ messages of length $\frac{N\,nb}{p}$. The estimated execution time for the one-dimensional redistribution of a single panel is then given by

$$T_{1d-panel}(N,p) \approx \sqrt{p}\,(\alpha + \frac{N\,nb}{p}\beta_d).$$

In the physical blocking, aggregation and LCM blocking strategies, $\frac{N}{nb}$ panels per matrix operands are redistributed. Since each process row and column operates at best independently from each other, the total redistribution time is approximately given by:

$$T_{1d-all}(N,p) \approx \frac{N\,\sqrt{p}\,\alpha}{nb} + \frac{N^2\,\beta_d}{\sqrt{p}}.$$

This estimated execution time illustrates that the one-dimensional algorithm features an optimal volume of communication per panel. However, the number of messages exchanged can be much larger than the minimal number (p).

4.4.5 Two Dimensional Redistribution

The last redistribution strategy considered in this dissertation involves the complete redistribution of the matrix operands A and B beforehand. This operation has been implemented in the ScaLAPACK library [78]. This two-dimensional redistribution software was used for our experiments. The algorithm implemented features a minimal communication volume. The message scheduling policy is the “caterpillar” algorithm. This scheme is not contention free. Therefore, the estimated execution time given below should be regarded as a lower bound. In the context of the two-dimensional redistribution, all p processes are involved. Each of them owns $\dfrac{N^2}{p}$ data items. Each process sends and receives $p \Leftrightarrow 1 \approx p$ messages of length $\dfrac{N^2}{p^2}$. The estimated execution time for the two-dimensional redistribution of an entire distributed matrix operand is thus given by

$$T_{2d-all}(N,p) \approx p\,\alpha + \frac{N^2\,\beta_d}{p}.$$

This estimated execution time illustrates that this two-dimensional algorithm features an optimal volume of communication per matrix operand. The number of messages exchanged during the operation is also minimal.

4.5 Conclusions

Table 4.1 summarizes the estimated parallel efficiency for each variant studied in this chapter. For aligned experiments, these efficiencies show that all variants are scalable in the sense given above, i.e., efficiency is maintained if the memory-use per process is kept constant. The LCM blocking variant features a slightly higher efficiency than the physical blocking strategy. This theoretical analysis also explains why one expects to observe better performance for the physical strategy than the aggregation variant. For non-aligned experiments, all variants are scalable only if one neglects the latency factor. The complete redistribution (RED) is shown to be more efficient, because it is optimal in terms of communication volume for a matrix operand. The higher number of messages exchanged by the LCM strategy is, however, likely to make the difference on platforms featuring a high latency (α). Due to the crudeness of the machine model used for the performance analysis of these algorithms, these theoretical predictions must be confronted to

Table 4.1: Estimated parallel efficiencies for various blocking variants

	Aligned experiments	Non-aligned experiments
PHY	$(1+\frac{2}{\gamma}(\frac{p\,\alpha}{NB_{dis}N^2}+\frac{\sqrt{p}\,\beta_d}{N}))^{-1}$	$(1+\frac{1}{\gamma}(\frac{(\sqrt{p}+2)\,p\,\alpha}{NB_{dis}N^2}+\frac{3\sqrt{p}\,\beta_d}{N}))^{-1}$
AGG	$(1+\frac{k}{\gamma}(\frac{p\,\alpha}{NB_{log}N^2}+\frac{\sqrt{p}\,\beta_d}{N}))^{-1}$ $k \approx \lceil\frac{NB_{log}}{NB_{dis}}\rceil$	$(1+\frac{1}{\gamma}(\frac{(k+\sqrt{p})\,p\,\alpha}{NB_{log}N^2}+\frac{(k+1)\sqrt{p}\,\beta_d}{N}))^{-1}$ $k \approx \lceil\frac{NB_{log}}{NB_{dis}}\rceil$
LCM	$(1+\frac{3}{2\,\gamma}(\frac{p\,\alpha}{NB_{log}N^2}+\frac{\sqrt{p}\,\beta_d}{N}))^{-1}$	$(1+\frac{1}{\gamma}(\frac{(3/2+\sqrt{p})\,p\,\alpha}{NB_{log}N^2}+\frac{5\sqrt{p}\,\beta_d}{2\,N}))^{-1}$
RED	$(1+\frac{1}{\gamma}((2+\frac{p\,NB_{dis}}{N})\frac{p\,\alpha}{NB_{dis}N^2}+\frac{(2\sqrt{p}+1)\,\beta_d}{N}))^{-1}$	

practical experiments. One also expects these theoretical results to differ from the reality in larger proportions for the non-aligned experiments due to the neglection of the link contention by the machine model. It is worth noticing that for small grid sizes and all variants, the estimated efficiency of the aligned experiments is approximately equal to the efficiency of the non-aligned experiments. Similarly, the time complexity obtained above for the one- and two-dimensional redistribution does not take into account the fact that small physical blocking factors simplify considerably these operations as shown in Chapter 2. Furthermore, the aggregation and LCM blocking strategies should be regarded as complementary variants as explained in Chapter 3. Indeed, for some algorithms, it is not possible to reorder the operations as it is done by the LCM blocking strategy.

The estimated execution time can be used to predict the actual execution time of an implementation of these algorithms. Another use of these results is to compute the repartition of the total estimated execution time between communication and computation. Figures 4.5 and 4.6 illustrate this use of the estimation for the

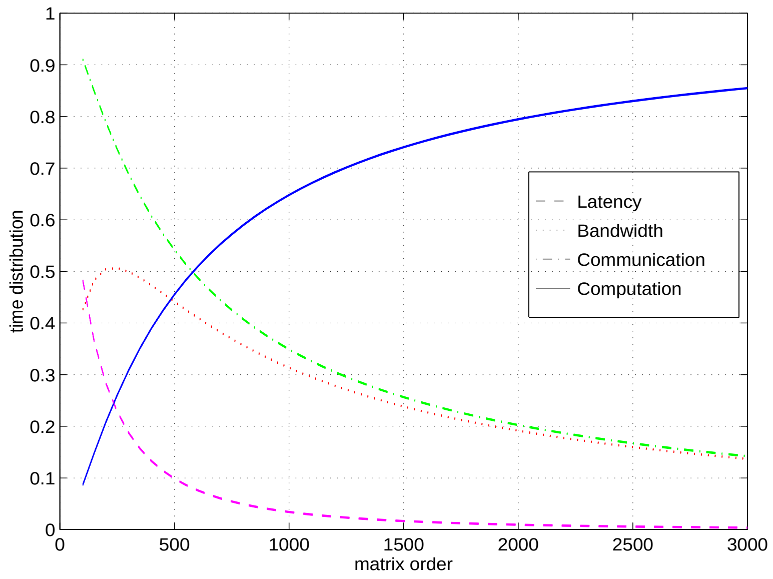


Figure 4.5: Time repartition of the aligned LCM blocking variant (LCM) on a 4×8 IBM SP

LCM blocking strategy applied to an aligned experiment and the complete redistribution (RED) applied to a non-aligned experiment. Both figures indicate that the model developed in this chapter is optimistic. Indeed, for even small-sized matrices, the time spent communicating is rapidly less than 50 % of the total execution time. This says that the model predicts very high performance for the aligned and non-aligned experiments. These figures also show the relative impor-

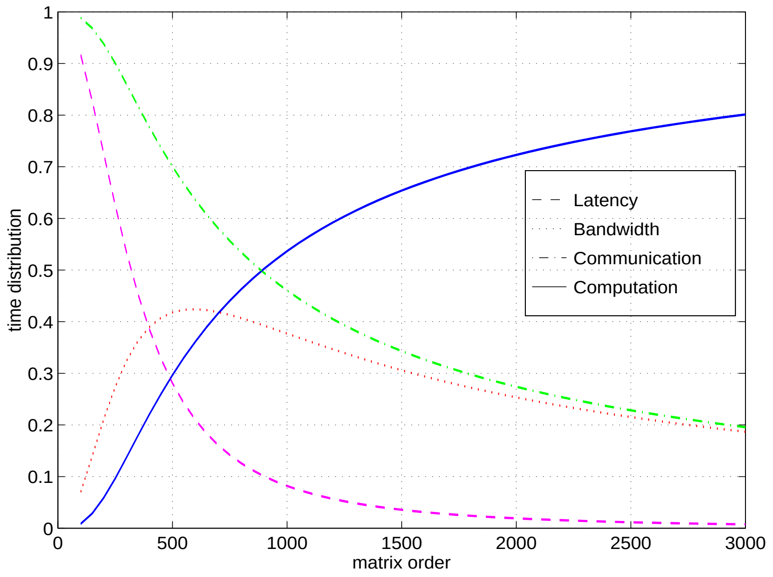


Figure 4.6: Time repartition of the non-aligned complete redistribution (RED) variant on a 4×8 IBM SP

tance of the performance of the interconnection network within the context of data redistribution. The relative performance impact of the communication performance is larger for the non-aligned experiment. It also slowly decreases with the problem size.

Chapter 5

Experimental Results

5.1 Introduction

In order to assess the performance of algorithmically redistributed operations, many experiments have been performed. Each of them is aimed at illustrating the efficiency of these operations. This chapter presents and discusses the results obtained. All experiments were performed in double precision arithmetic, and the matrix operands were randomly generated. The vendor-supplied BLAS library was used on the Intel XP/S Paragon and the IBM SP. The current native version 1.0 of the BLACS [17] was used on both systems. The experimental programs were compiled and executed unchanged on both platforms. A testing program was developed for debugging purposes as well as ensuring the validity of the results. This program was adapted from the more general testing software accompanying the PBLAS library [18]. The software passed statistically a large number of tests, that is, for a finite collection of random valid input arguments. Similarly, a timing program was developed and used to obtain the results presented below. Most of the experiments were performed twice or more. Only the best performance is reported.

The experimental results are presented and classified into two categories. First, the matrix operands A, B and C have been distributed such that the rows of the matrix operand A (respectively the columns of the matrix operand B) and the rows (respectively columns) of the matrix operand C were residing in the same process row (respectively column). These experiments are referred to as *aligned* experiments thereafter. The second category of experiments considers matrix operands that are not aligned, so that a complete redistribution of the matrix operands A and B has to occur. These experiments are referred to as *non-aligned* experiments. A single value of the logical value blocking factor NB_{log} has been a priori determined for each platform and used for all the experiments. Finally, for each category of experiments, different values of the physical block sizes have been timed.

Most of the experiments performed on the Intel XP/S Paragon have been performed twice on a dedicated machine. Only the best performance is reported. All experiments performed on the IBM SP were performed using a batch queueing system. The machine was never set up in a single user mode. However, previous relevant timing results obtained on another dedicated system indicate no meaningful differences with the results obtained by this batch queueing system.

For the sake of clarity, only limited results have been used for the plots presented in this chapter. Appendix B contains the tables of complete results. The purpose of this chapter is to illustrate the general behavior of algorithmically redistributed operations as opposed to presenting a collection of particular performance numbers. One can still precisely identify the relationship between a performance plot and the corresponding experiments. However, the presentation style aims at facilitating the comparison of the different blocking strategies for a set of illustrative and particular cases. For example, for a given blocking variant, one is interested in the performance variations as a function of the block sizes used for the distribution matrix operands. Ideally, one would like to minimize this dependence so that the performance of such an operation on a given machine configuration becomes a function of only the problem size.

5.2 Determining a "Good" Block Size

A "good" block size or blocking factor is one that maximizes the performance of a block algorithm. According to the results presented in Figures 4.2 and 4.4, the local performance is not very sensitive to the size of the matrix operands as soon as their sizes remain large enough. This allows one to determine a lower bound on the physical and/or logical block sizes under which the local performance would be the main factor for overall slow performance. Figures 4.2 and 4.4 indicate that the value of this lower bound is approximately 10 for the Intel XP/S Paragon and 20 for the IBM SP. A good value of the distribution block size can be empirically determined by trying a few candidates for a given problem size and a given grid size. Table 5.1 shows the performance in Mflops obtained for matrices of order 500 on a 2×2 process grid of the Intel XP/S Paragon, and matrices of order 1000 on a 2×4 process grid of the IBM SP for different values of the distribution blocking factor NB_{dis}. One should avoid selecting overly larges values for these

Table 5.1: Performance in Mflops for distinct distribution block sizes

2×2 Intel XP/S Paragon	
NB_{dis}	$M = N = K = 500$
8	163.59
12	164.41
14	168.85
16	164.48
18	169.12

2×4 IBM SP	
NB_{dis}	$M = N = K = 1000$
20	950.39
40	1063.99
60	1073.59
70	1103.12
80	1079.90

"good" block sizes in order to avoid load imbalance as well as limit the amount of workspace required by the parallel subprograms. Finally, a few simple experiments using the physical blocking variant allow one to determine somewhat arbitrarily an approximate value of this "good" blocking factor. Table 5.1 shows some partial results for a range of NB_{dis} values that were used in determining a "good" blocking factor for the Intel XP/S Paragon and the IBM SP. On the Intel XP/S Paragon, we found that a reasonable value for this blocking factor is 14. On the IBM SP, the value of 70 has been selected for the rest of our experiments. As mentioned above, the results presented in Table 5.1 illustrate on a particular case that the overall performance of this operation is not very sensitive to a range of values for the blocking factor. Moreover, an optimal value maximizing performance for all problem sizes does not exist. This value depends on the problem size, the target machine as well as the process grid considered. Therefore, one could have chosen other values within this acceptable range without greatly affecting the performance of the parallel operation. In all of the performance results presented hereafter, the value of the logical blocking factor has been chosen to be equal to the "good" values indicated above.

Good values of the physical or logical blocking factors are machine and algorithm dependent. From a software portability point of view, one can store these values in a table. At run-time, these values will be retrieved from this table. This is the option that has been selected by the LAPACK [5] designers. It is however conceivable to determine such values at run-time by performing a few quick experiments. On a distributed memory concurrent computer, such a method is particularly attractive because the overhead of such trials is in general negligible. However, the main problem with this approach is that all processes should agree on the value of the logical blocking factor to be used. Thus, on heterogeneous or unequally loaded homogeneous platforms, this requires a synchronization phase that lowers the advantages of this run-time approach. Consequently, the most appealing solution is to empirically determine those good blocking values before the installation of the software. One would then encode them in a static table and finish installing the software. Recompilation of the software is required when some hardware component of the system is changed; however, it is possible to determine slightly better blocking factors.

5.3 Specification of the Experiments

In this section the experiments are precisely specified. Each experiment has been given an encoded name of the form XX_T#. XX identifies on which target machine the experiment has been run, either XP for the Intel XP/S Paragon or SP for the IBM SP. T specifies the type of the operation and can be either A or N. If T is A, the operation is aligned as defined in Section 4.4. If T is N, the operation is not aligned. # is a number or a string distinguishing each experiment. For each experiment, the physical distribution parameters of the matrix operands A, B and C are specified, followed by an explanation of the purpose of the experiment. Table 5.2 contains the specifications of all of the experiments that have been performed. In all of the experiments, the matrix operands were square of order N. The values of N used for all experiments are 100, 250, 500, 1000, 1500, 2000 and 3000. Due to memory size constraints, it was not always possible to perform the experiments

Table 5.2: Specification of the experiments

Aligned Experiments	
Experiment #	**XP_A0, SP_A0**
Distribution	$r_A = s_A = r_B = s_B = r_C = s_C = NB_{log}$
Comments	Pure overhead of algorithmic blocking.
Experiment #	**XP_A1, SP_A1**
Distribution	$r_A = s_A = r_B = s_B = r_C = s_C = 1$
Comments	Impact of distribution blocking factors $\ll NB_{log}$.
Experiment #	**XP_A10**
Distribution	$r_A = s_A = r_B = s_B = r_C = s_C = 10$
Experiment #	**SP_A20**
Distribution	$r_A = s_A = r_B = s_B = r_C = s_C = 20$
Comments	Impact of distribution blocking factors $\leq NB_{log}$.
Experiment #	**XP_A40**
Distribution	$r_A = s_A = r_B = s_B = r_C = s_C = 40$
Comments	Impact of distribution blocking factors $\geq NB_{log}$.
Experiment #	**XP_A100**
Distribution	$r_A = s_A = r_B = s_B = r_C = s_C = 100$
Experiment #	**SP_A200**
Distribution	$r_A = s_A = r_B = s_B = r_C = s_C = 200$
Comments	Impact of distribution blocking factors $\gg NB_{log}$.
Non-aligned Experiments	
Experiment #	**XP_NA, SP_NA**
Distribution	$r_A = r_C = 40, s_B = s_C = 40, s_A = 5, r_B = 7$
Comments	Columns of A are not aligned with rows of B.
Experiment #	**XP_N1, SP_N1**
Distribution	$r_A = s_A = r_B = s_B = r_C = s_C = 1$
Comments	Impact of very small distribution blocking factors.
Experiment #	**XP_NN, SP_NN**
Distribution	$r_A = 3, s_A = 5, r_B = 7, s_B = 2, r_C = s_C = 40$
Comments	Non-aligned operation, small distribution blocking factors.

for all of these values. All of the experiments have been performed on five distinct process grids, namely 1×2, 2×2, 2×4, 4×4 and 4×8. As suggested in Section 4.4, the four blocking and redistribution strategies (PHY, AGG, LCM and RED) have been tried for almost all experiments. Unless otherwise specified, the value of the logical blocking factor NB_{log} has been chosen to be 14 on the Intel XP/S Paragon, and 70 on the IBM SP.

Experiments XP_A0 and SP_A0 use the value of NB_{log} as the physical distribution blocking factor for all of the matrix operands. These experiments aim at verifying that the algorithmically redistributed variants do not affect the reference performance obtained by the physical blocking strategy. Figures 5.1 and 5.2 show the performance of the physical blocking (PHY), aggregation (AGG) and

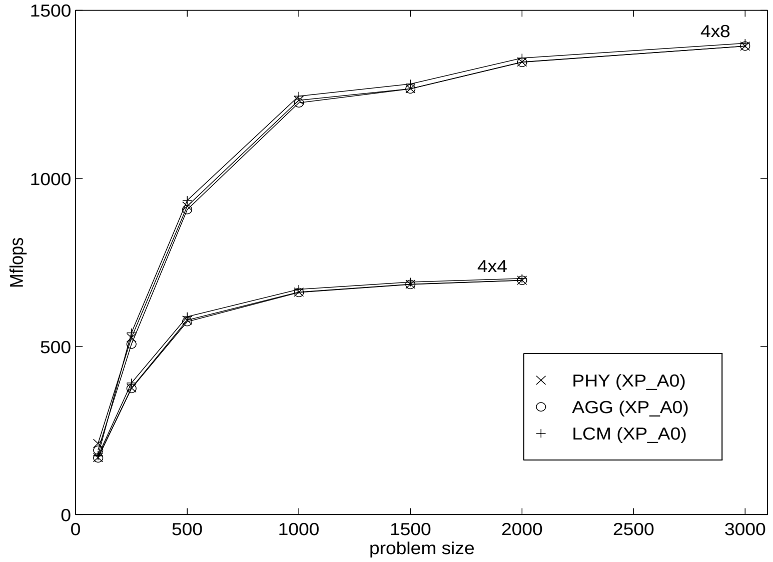


Figure 5.1: Performance in Mflops of algorithmic blocking variants for a “good” physical data layout case and various process grids on the Intel XP/S Paragon

the LCM blocking (LCM) strategies using the value of NB_{log} as the logical and distribution blocking factors for the three matrix operands. According to the conclusions of the previous chapter, the performance of the three variants is almost identical on each platform with a slight advantage to the LCM blocking variant.

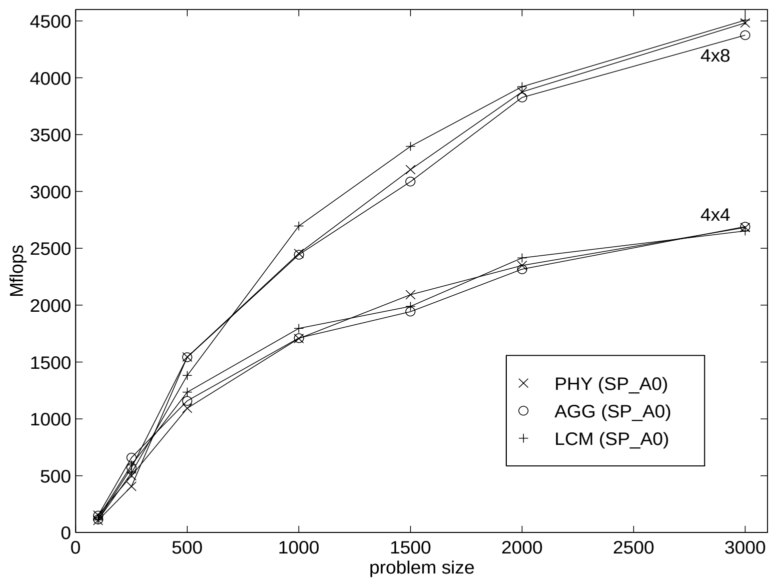


Figure 5.2: Performance in Mflops of algorithmic blocking variants for a "good" physical data layout case and various process grids on the IBM SP

In the rest of this chapter, the performance curves shown in Figures 5.1 and 5.2 are considered as a reference. The combined maximum of these curves has been replicated on all of the other plots presented. This maximal curve is thereafter always represented as a bold solid line. Ideally, one would like to observe no difference between the performance obtained for this "good" physical layout and the performance achieved by distributions induced by other physical blocking factors.

5.4 Aligned Experiments

In this section, the performance results obtained for the aligned experiments are presented for each blocking variant separately.

5.4.1 Physical Blocking

Figures 5.3 and 5.4 show the performance results obtained by the physical blocking strategy on aligned data. The physical blocking variant uses the distribution blocking factors as the computational unit. When the distribution parameters

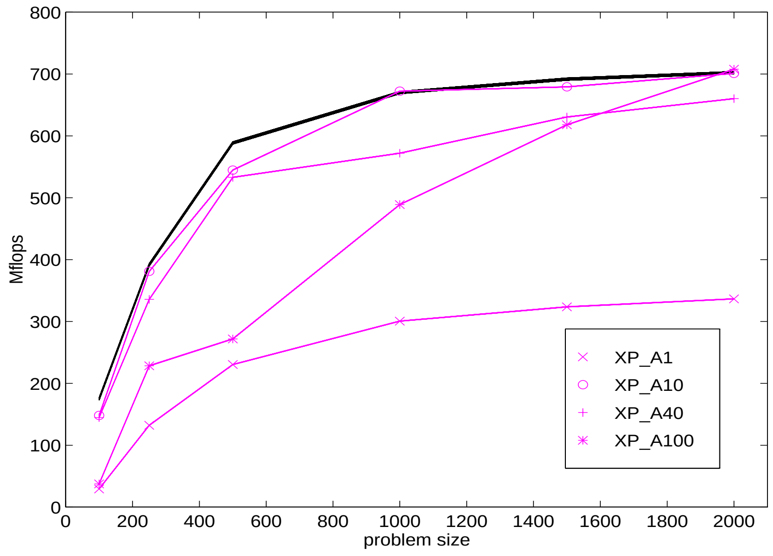


Figure 5.3: Performance of aligned physical blocking on a 4×4 Intel XP/S Paragon

are very small, one expects a large performance degradation because of the local performance of the rank-K update for small values of K as shown in Figures 4.2 and 4.4. Similarly, very large distribution block sizes increase the computation load imbalance as explained in Section 4.4. Figures 5.3 and 5.4 illustrate these two phenomena. The load imbalance is characterized by highly irregular performance results. For example in Figure 5.4, for $N = 1500$, each process has almost the same amount of data. However, for $N = 2000$, the matrix operands are made of 10×10 blocks of size 200. Since 10 is not divisible by 4 or 8, the most loaded processes have locally a 600×400 matrix on which to operate. The matrices residing in the

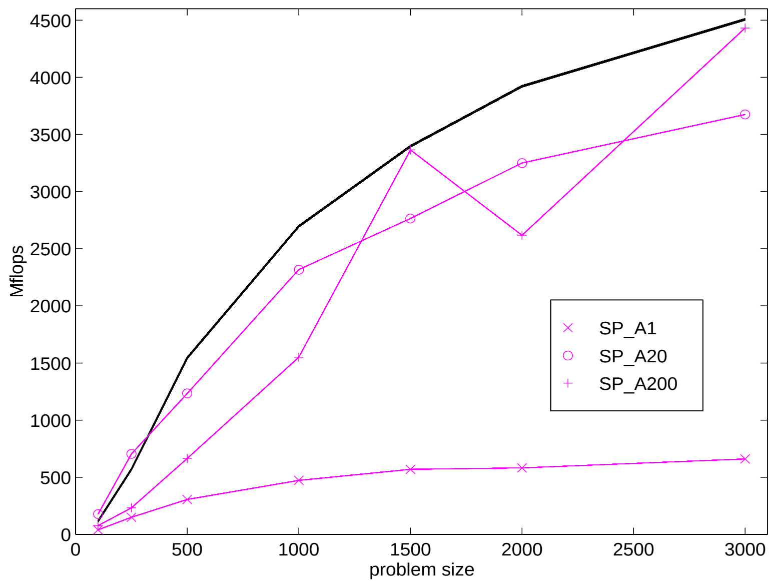


Figure 5.4: Performance of aligned physical blocking on a 4×8 IBM SP

least loaded processes are however of size 400×200. Therefore, some processes have three times as much work to perform than others. The ragged curves shown in Figures 5.3 and 5.4 are typical of load imbalance. When the distribution block size is very small, the performance is dramatically degraded. This is the difference that one should expect when using Level 1 or 2 BLAS based algorithms as opposed to Level 3 BLAS based algorithms.

5.4.2 Aggregation - Disaggregation

Figures 5.5 and 5.6 show the performance results obtained for the aggregation strategy on aligned data. These figures show that the aggregation variant decreases by a large amount the dependence of the performance from the physical distribution parameters, and thus smooths the performance results of the rank-K update towards the result of reference. The (dis)aggregation strategy builds

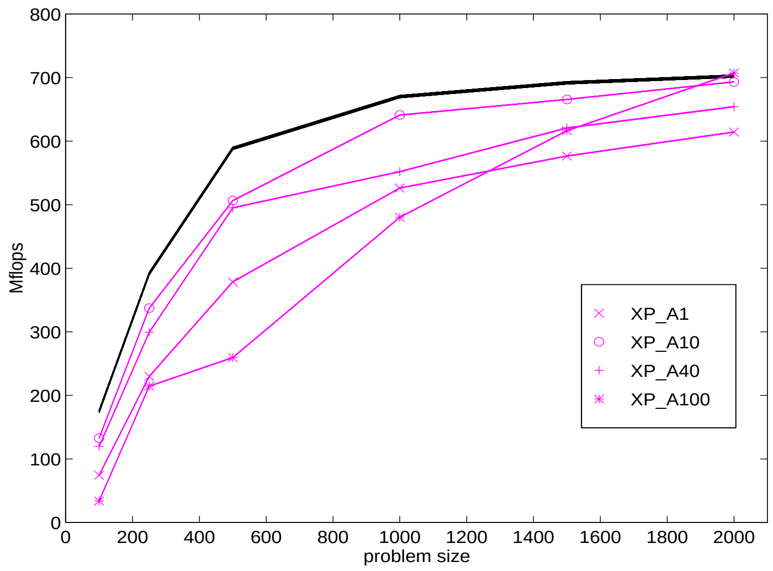


Figure 5.5: Performance of aligned aggregation on a 4×4 Intel XP/S Paragon

panels of NB_{log} globally continuous columns of A and rows of B. When the distribution parameters are very small, one expects a large performance improvement compared to the physical blocking strategy. This aspect is particularly evident for both target platforms as shown in Figures 5.5 and 5.6. The aggregation phase induces some communication overhead that somewhat limits the potential of this strategy. This phenomenon is not particularly well illustrated on the Intel XP/S Paragon due to the high speed of the interconnection network compared to the local computational performance. However, on the IBM SP, even if the per-

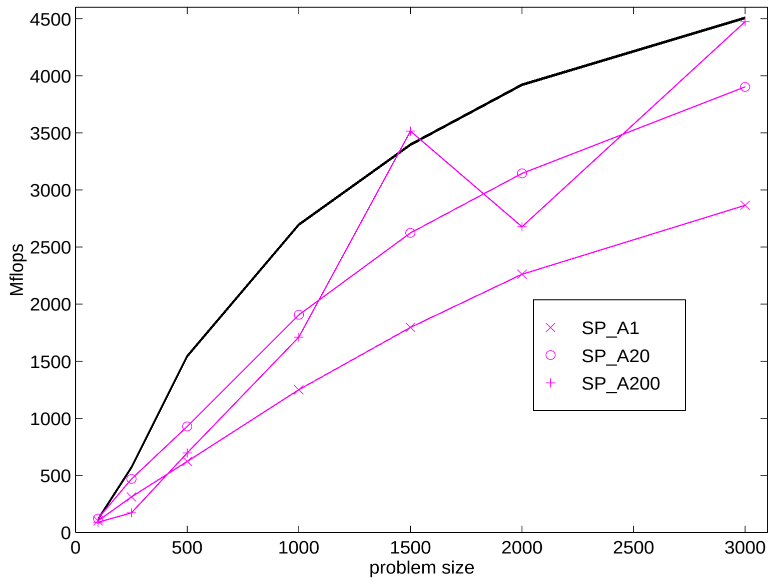


Figure 5.6: Performance of aligned aggregation on a 4×8 IBM SP

formance of Experiment SP_A1 has been considerably improved, it remains much lower than the reference performance because of the less favorable communication-computation performance ratio of this machine.

5.4.3 LCM Blocking

Figures 5.7 and 5.8 show the performance results obtained for the LCM blocking strategy on aligned data. These figures show that the LCM blocking variant produces the same effect as the aggregation strategy. It desensitizes the performance results from a poor choice of the blocking factor. The LCM results are however better than the ones shown above for the aggregation variant. In particular, the performance results observed for Experiments XP_A1 and SP_A1 have been considerably improved. On the Intel XP/S Paragon, the performance obtained for very small physical blocking factors is now superior to the performance observed

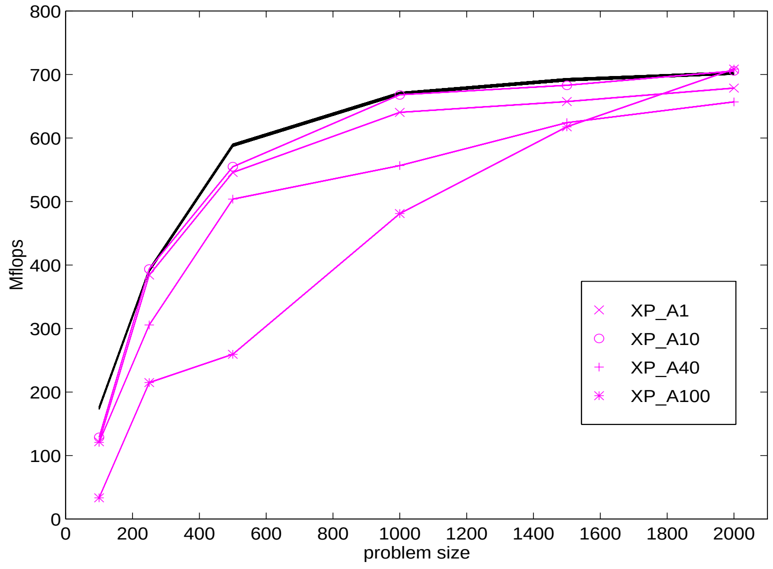


Figure 5.7: Performance of aligned LCM blocking on a 4×4 Intel XP/S Paragon

for physical blocking factors slightly larger than NB_{log} (XP_A40). On the IBM SP, there is virtually no performance difference between Experiments SP_A1 and SP_A20. The impact of the less favorable communication-computation performance ratio of this particular machine is somewhat hidden by the algorithmic blocking strategy. This relatively low ratio is however, the reason for the performance difference between the reference case and the Experiments SP_A1 and

SP_A20. The LCM blocking strategy builds panels of NB_{log} rows and columns with less communication overhead because it essentially determines and regroups the columns of A and rows of B that belong to a given process column and process row pair. This phase is communication free. On both platforms, the results

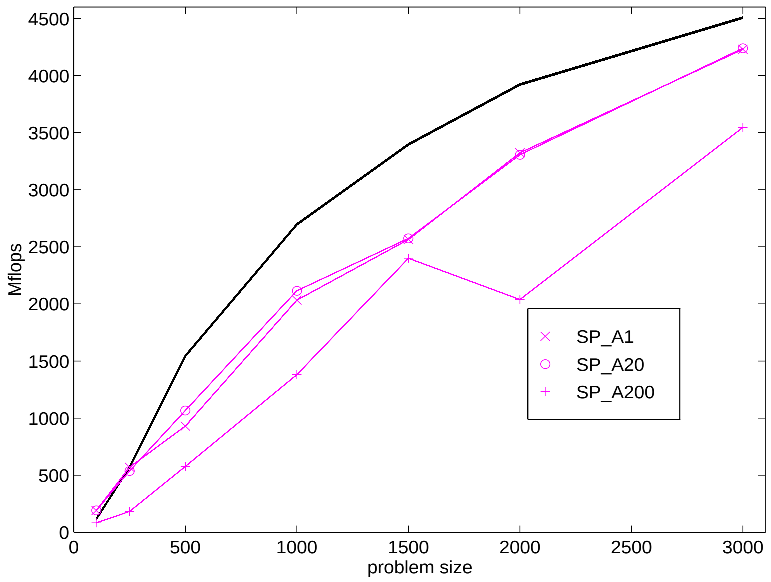


Figure 5.8: Performance of aligned LCM blocking on a 4×8 IBM SP

are spectacular. They show that for aligned data and uniform data distributions, the performance difference due to various distribution blocking factors is no more than a few percentage points from the reference as defined in Section 5.3.

5.4.4 Complete Redistribution

Figures 5.9 and 5.10 show the performance results when the matrix operands A and B are aligned but redistributed for efficiency reasons. Even if these plots show the performance obtained for the same experiments as the last three sections, one could argue that complete redistribution (RED) should only be used for the extreme cases. A major feature of redistributing the entire matrix operands A and B at once is the large memory cost required by this operation. This increases the chances of the possible use of virtual memory by a large factor. Figure 5.9 illustrates the dramatic performance consequences of using virtual memory on

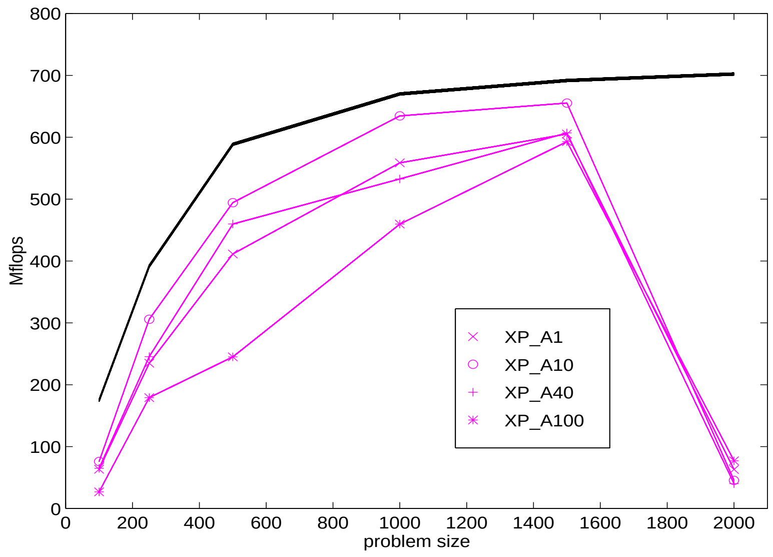


Figure 5.9: Performance of aligned redistribution on a 4×4 Intel XP/S Paragon

the Intel XP/S Paragon. On this particular machine the complete redistribution beforehand leads to lower performance than the one obtained by the LCM blocking variant. In other words, the cost of redistributing when needed beforehand is larger than the cost induced by the algorithmically redistributed LCM strategy. In both variants the amount of computation is performed at the same speed. On the IBM SP, the complete redistribution beforehand leads to slightly higher performance than the LCM blocking strategy. The lower total number of redistribution messages of the complete redistribution strategy takes better advantage of the low communication-computation performance ratio of this machine. It is clear

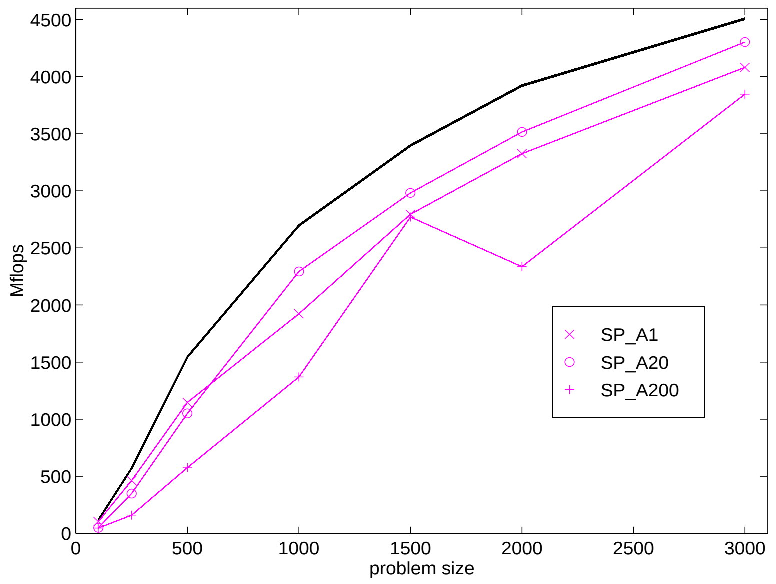


Figure 5.10: Performance of aligned redistribution on a 4×8 IBM SP

that the IBM SP may need to use virtual memory for sufficiently large problem sizes. However, the nodes of the machine we used for our experiments had each at least 128 MB of physical memory. It was not feasible to estimate the impact of the use of virtual memory in a reasonable amount of time.

5.5 Non-Aligned Experiments

Experiments XP_AN and SP_AN present the distinctive feature that only the columns of the matrix operand A are not aligned with the rows of the matrix operand B. Figures 5.11 and 5.12 show the performance results obtained for these experiments. In this particular case, it is not necessary to redistribute the operands A and B, but merely to perform a succession of rank-K update operations as in the aligned case. Thus, one expects to draw the same conclusions

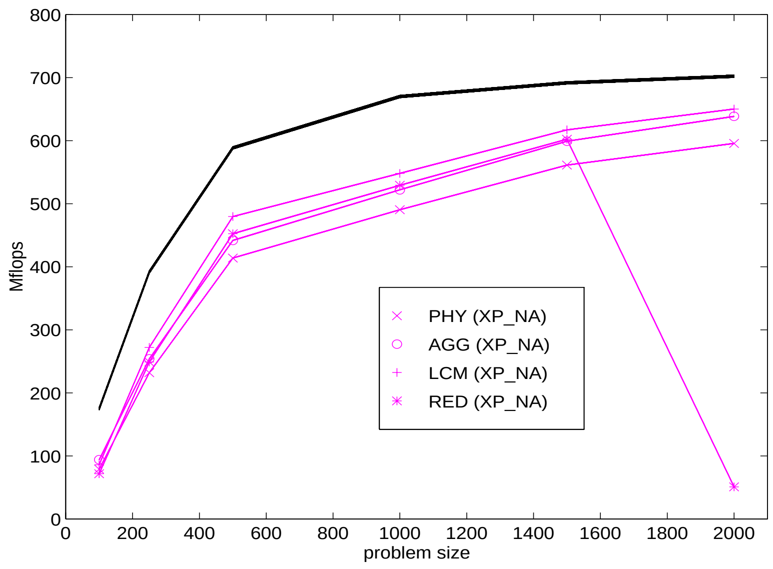


Figure 5.11: Performance in Mflops of algorithmic blocking variants for Experiment XP_NA on a 4×4 Intel XP/S Paragon

as the ones obtained in the previous section. These are the reasons justifying the presentation of these non-aligned experimental results separately. On the Intel XP/S Paragon, the performance results obtained by the four variants are similar with a slight advantage for the LCM blocking strategy (LCM). Just as it has been observed above, the complete redistribution beforehand (RED) allows for high performance as well, even if the physical memory constraints prevent from the collection of decent results for the largest problems. On the IBM SP, the

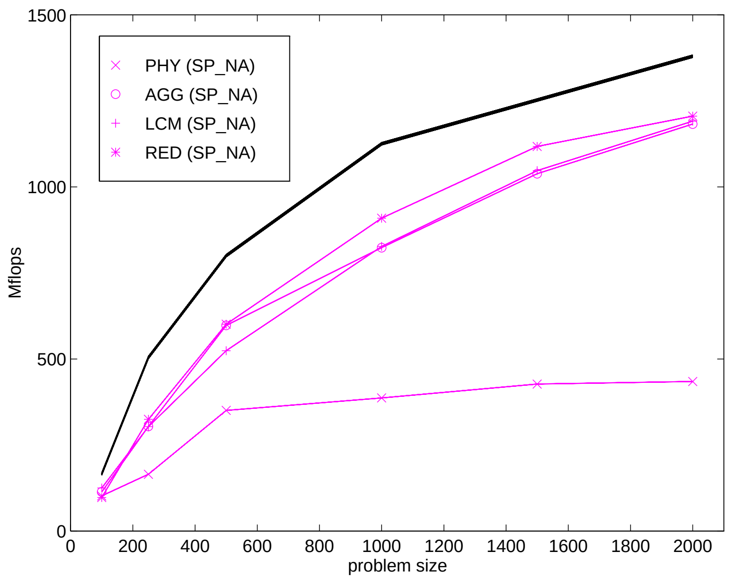


Figure 5.12: Performance in Mflops of algorithmic blocking variants for Experiment XP_NA on a 2×4 IBM SP

performance of the physical blocking strategy is considerably limited by the local performance of the rank-K update for small values of K. There is very little difference between the three variants AGG, LCM and RED.

5.5.1 Physical Blocking

Figures 5.13 and 5.14 present the performance results obtained for the physical blocking strategy (PHY). These results are mainly presented for completeness since this technique cannot achieve performance close to the reference for the selected non-aligned experiments. Indeed, Experiments XP_N1 and SP_N1 feature very small distribution blocking factors. Consequently, the local computational performance is low as shown in Figures 4.2 and 4.4. Experiments XP_NN and SP_NN feature also, in a slightly different sense however, too small physical distribution blocking factors. At each step of the physical blocking strategy, no

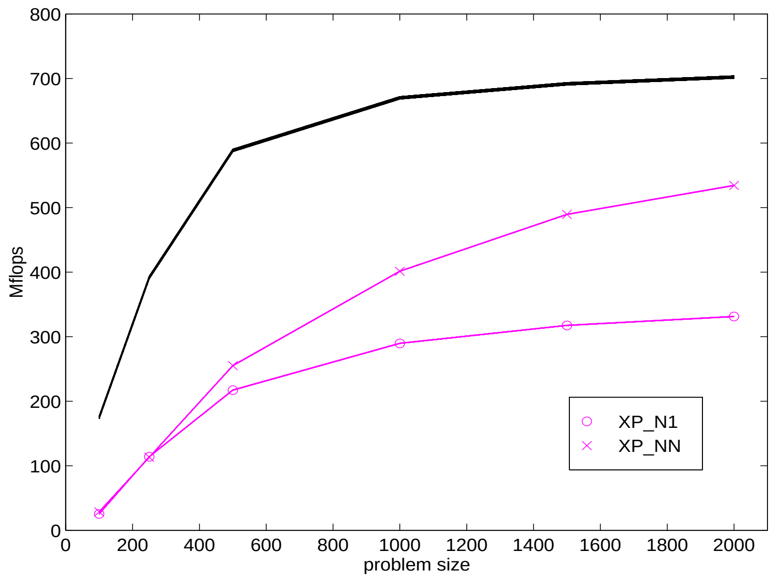


Figure 5.13: Performance of non-aligned physical blocking on a 4×4 Intel XP/S Paragon

attempts are made to regroup columns of A (respectively rows of B) that are not in the same block or the same process column (respectively process row). Consequently, the local rank-K update operation is performed on at best the minimum of s_A and r_B columns of A and rows of B. This also increases the number of panel redistributions since in this case the values of s_A and r_B are smaller than the value of NB_{log} in both target machines. The consequences of these remarks are clearly illustrated on both figures. The performance results obtained for the physical blocking strategy are considerably lower than the ones used for reference.

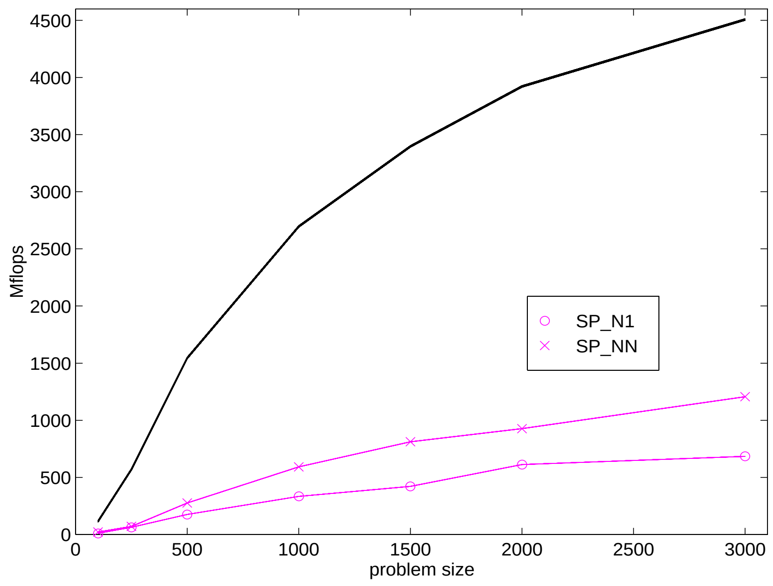


Figure 5.14: Performance of non-aligned physical blocking on a 4×8 IBM SP

5.5.2 Aggregation - Disaggregation

Figures 5.15 and 5.16 illustrate the performance achieved by the aggregation variant on non-aligned data. The performance results of the rank-K update are smoothed towards the reference results. As shown, this technique is much more efficient than the physical blocking variant. On the Intel XP/S Paragon, due to the high performance of the interconnection network, there is very little difference with the results obtained on aligned data. As one may expect, this small difference is the largest for small and medium problem sizes. In Figure 5.15 as well as almost all of the figures presented thereafter, the performance obtained for Experiments XP_N1 and SP_N1 is higher than that measured for Experiments XP_NN and SP_NN. Indeed, the redistribution phase for matrix operands distributed with very small physical block sizes is simpler and cheaper. For large problem size, the performance loss is estimated to be approximately to 15%. On

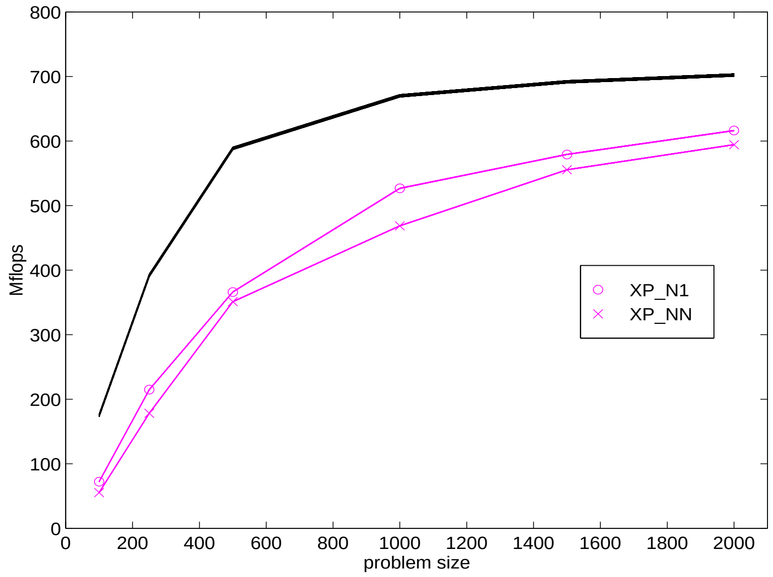


Figure 5.15: Performance of non-aligned aggregation on a 4×4 Intel XP/S Paragon

the IBM SP, the relative low performance of the interconnection network compared to the computational speed of the nodes prevents the aggregation strategy from obtaining much more than half of the reference performance. Still, small physical distribution blocking factors perform slightly better than more general distribution parameters. The gradual slope of the performance curve obtained for

Experiments SP_N1 and SP_NN is typical of lower communication performance. For example, most of the performance curves shown for the Intel XP/S Paragon feature a steep increase for small and medium size problems.

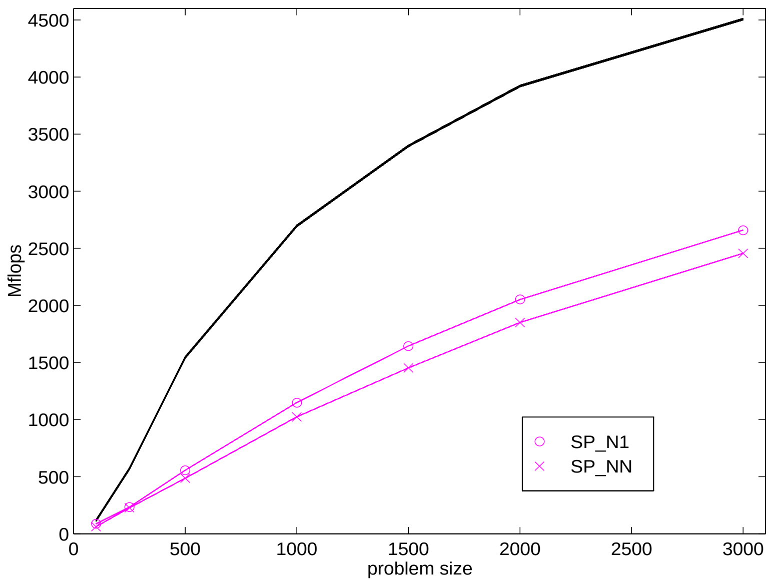


Figure 5.16: Performance of non-aligned aggregation on a 4×8 IBM SP

5.5.3 LCM Blocking

Figures 5.17 and 5.18 show the performance results obtained by the LCM blocking variant for the non-aligned Experiments XP_N1, XP_NN, SP_N1 and SP_NN. Assuming that one can always find at least NB_{log} columns of A and rows of B in every process column and row pair, one expects to observe slightly superior performance results than the ones presented in the last section. This is exactly what is shown in Figure 5.17 for the Intel XP/S Paragon. The performance obtained for non-aligned data on this machine is impressive. When combining this figure with the one presented earlier for aligned experiments, one can conclude that it is possible to achieve the highest (within 15 %) performance for aligned and non-aligned data independent from the physical distribution parameters and at a very low cost of physical memory.

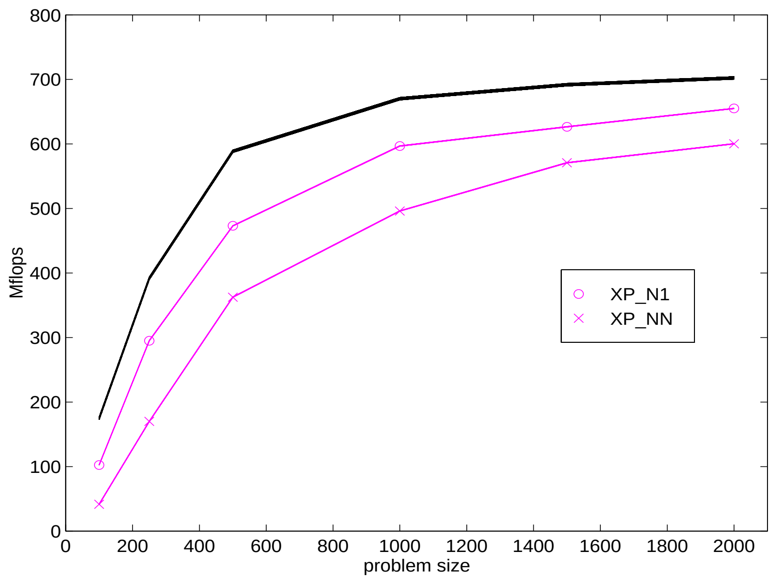


Figure 5.17: Performance of non-aligned LCM blocking on a 4×4 Intel XP/S Paragon

Figure 5.18 presents the performance results for the non-aligned experiments obtained on the IBM SP. The performance difference with the reference is much larger on this machine. Moreover, the performance obtained for Experiment SP_NN is much lower than that observed for Experiment SP_N1. This is surprising, especially when one considers the fact that so far the algorithmic blocking strategies have always smoothed the performance difference ascribed to the physical data layout. This argument has to be weighted, however, against the simpler redistribution operations performed in Experiment SP_N1. When one considers the differences in the experimentation methodologies used for the Intel XP/S Paragon and the IBM SP, there is in fact only one difference that comes to mind. The selected value of the logical blocking factor NB_{log} is much smaller (14) on the Intel XP/S Paragon than on the IBM SP (70). The distribution parameters for Experiment SP_NN give $\text{lcm}(P \times r_B, Q \times s_A) = \text{lcm}(40, 28) = 280$.

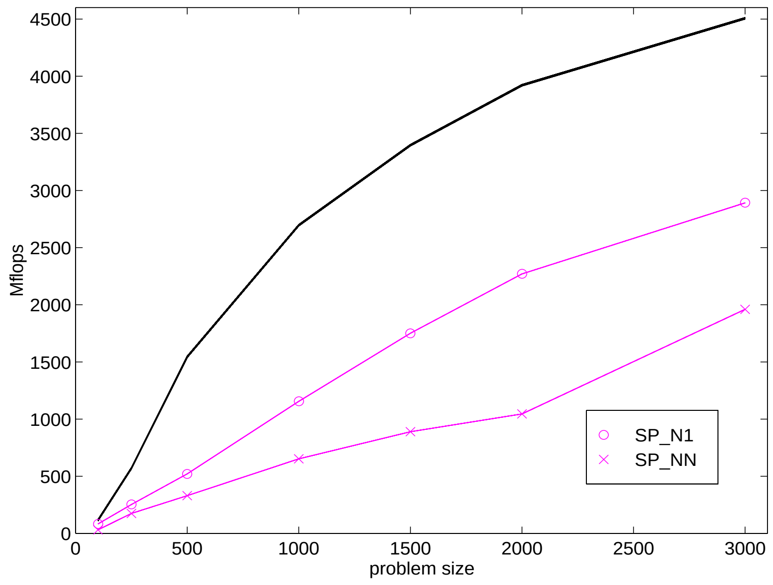


Figure 5.18: Performance of non-aligned LCM blocking on a 4×8 IBM SP

The corresponding LCM block is thus of order 280. For matrices of order 2000, there are approximately 7 diagonal LCM blocks. Each LCM block has 280 diagonals distributed over 32 processes, i.e., there are approximately, and almost exactly in this case, $63 = (280/32 \approx 9) \times 7$ diagonals per process. The LCM blocking strategy uses these virtual diagonals to pack columns of A and rows of B

by considering a process column and process row pair. These packed panels are then shifted along one dimension of the process grid. The algorithm performs thus 32 shift operations for each matrix operands. In Experiment SP_N1, however, 8 out of 32 processes own 250 diagonals each. In this case 32 shift operations are also performed for each matrix operands, but each of them consists of $12 = 4 + 8$ point-to-point communications instead of $4 \times 4 + 8 \times 8$ smaller messages. The performance difference observed for Experiments SP_N1 and SP_NN is due to these redistribution differences. On the Intel XP/S Paragon, this phenomenon is hidden by the the high performance of the interconnection network. Moreover, the small value of the logical blocking factor (14) on this platform also contribute to attenuate this effect. Indeed, in a given process column - process row pair, it is much easier to find 14 columns of A and rows of B rather than 70. These remarks justify the introduction in Chapter 3 of the hybrid blocking strategy (HYB) combining the advantages of the LCM variant and aggregation, i.e., minimizing the amount of communication while maintaining the computational granularity.

5.5.4 Complete Redistribution

Figures 5.19 and 5.20 show the performance results obtained for the non-aligned Experiments XP_N1, XP_NN, SP_N1 and SP_NN when the matrix operands A and B are redistributed at once. The physical blocking variant of the rank-K update operation is then performed to complete the computations. As it has been previously observed, the redistribution phase requires a large amount of workspace. Consequently, the use of virtual memory is inevitable for the largest matrix operands producing a non-negligible performance degradation. This phenomenon has been easier to observe on the Intel XP/S Paragon since the processors of the machine used for our experiments had 16 MB of physical memory from which

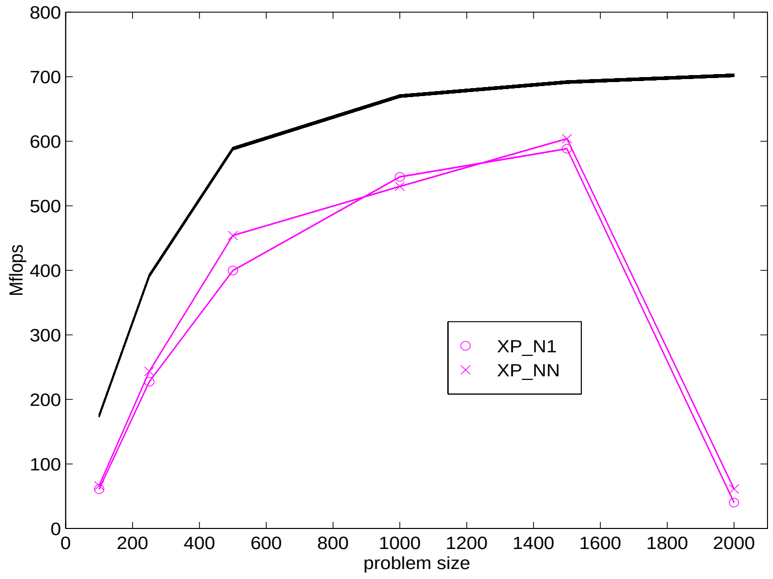


Figure 5.19: Performance of non-aligned redistribution on a 4×4 Intel XP/S Paragon

approximately 8 MB are usable by the user's program. The overall performance obtained by this technique is slightly better (a few percentage points) than the LCM blocking strategy. The precise performance numbers for these experiments are presented in Appendix B. On the IBM SP, the advantage of redistributing beforehand is greater and is shown in Figure 5.20. When applicable, this strategy appears to be the most efficient. The machine used for our experiments consisted of nodes having 128 MB or 256 MB of physical memory. Consequently, observing the effects of virtual memory on the IBM SP has not been attempted. Figure 5.20 also shows the performance results of the hybrid (HYB) strategy described above. The performance numbers have been reported in Appendix B. The hybrid technique provides better performance than the simpler LCM blocking variant as

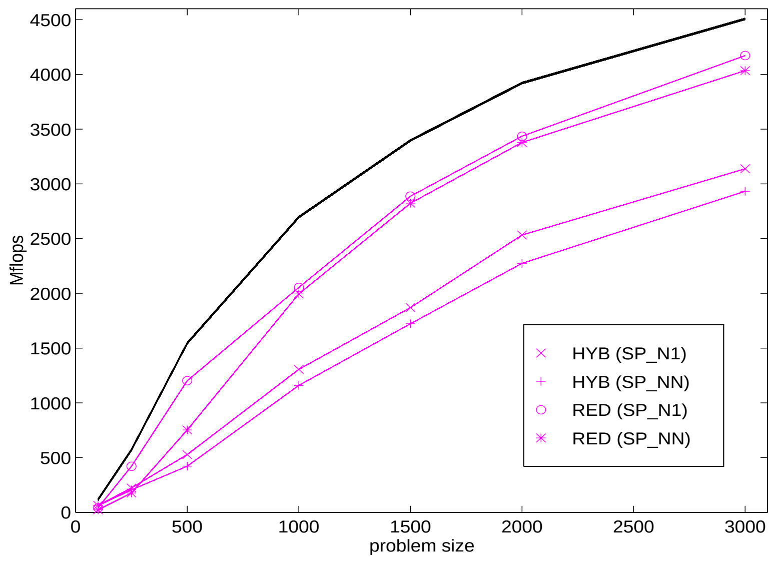


Figure 5.20: Performance of non-aligned hybrid (HYB) versus redistribution (RED) techniques on a 4×8 IBM SP

illustrated by the Figures 5.20 and 5.18. This strategy delays the computational phase until enough (NB_{log}) columns of the matrix operand A and rows of the matrix operand B have been algorithmically blocked. The communication overhead of this hybrid scheme is larger than the one induced by the LCM strategy. It allows, however, the achievement in certain cases of a much higher local performance rate. Obviously, the performance achieved by this hybrid scheme is always greater or equal to the one obtained by the LCM blocking variant.

5.6 Conclusions

A large number of experimental results were presented in this chapter. For our experiments, two platforms have been chosen with highly different communication-computation ratios. On one hand, the Intel XP/S Paragon allows for very efficient communications and relatively slow computations. On the other hand, the IBM SP features highly efficient processors for computational intensive applications and a comparatively slower interconnection network. This study restricted the scope of possible operations to a single one, namely the rank-K update operation. This operation is mathematically equivalent to a finite sequence of smaller rank-K updates. For the sake of the clarity of the presentation the experiments have been precisely specified and organized into two main categories. First, the "aligned" experiments feature simple and cheap redistribution phases due to restrictions on the data layout of the matrix operands. Second, the "non-aligned" experiments illustrate more general cases in terms of flexibility. Finally, four different blocking and redistribution strategies were studied. To perform the complete redistribution (RED) of a two-dimensional block cyclically distributed matrix into another matrix of the same kind, we used the appropriate component of the ScaLAPACK software library [78]. The rest of the software used in these experiments has been developed for this dissertation.

The results presented in this chapter clearly show that for the aligned experiments on both platforms, it is legitimate to use algorithmic redistribution variants. By doing so, one can obtain high performance and efficiency independent from the distribution parameters. Furthermore, the performance numbers obtained by the aggregation and LCM blocking techniques show a slight superiority for the latter. However, both techniques are complementary in the sense that it is not always possible to use the LCM blocking strategy as mentioned in Section 3.4.4. In order to address the problems induced by badly balanced computations, it is always possible to redistribute the matrix operand C, even if this somewhat contradicts the "owner's compute" rule. Another possibly more effective solution is to educate the users just like as has been done for the use of shared memory systems.

For non-aligned experiments, the results presented in this chapter not surprisingly illustrate the increasing importance of the communication-computation performance ratio. On the Intel XP/S Paragon, this ratio is quite high. In such a configuration, one can afford to redistribute the data "on the fly" without noticing much difference with the reference performance achieved in the aligned cases. In our context, redistributing the entire operands beforehand does not allow for any significant performance improvement on this platform. Interestingly, this strategy pointed out its own paradox. Indeed, in terms of performance, the larger the operands, the more benefits one should obtain from a complete redistribution. However, the amount of memory necessary to perform such an operation grows with the number of items redistributed, and thus inhibiting operation on the largest operands. This argumentation was at the beginning of our demarche for developing algorithmically redistributed operations that require an amount of memory proportional to the square root of the number of data items to be redistributed. Figure 5.21 shows the performance obtained by the LCM and hybrid

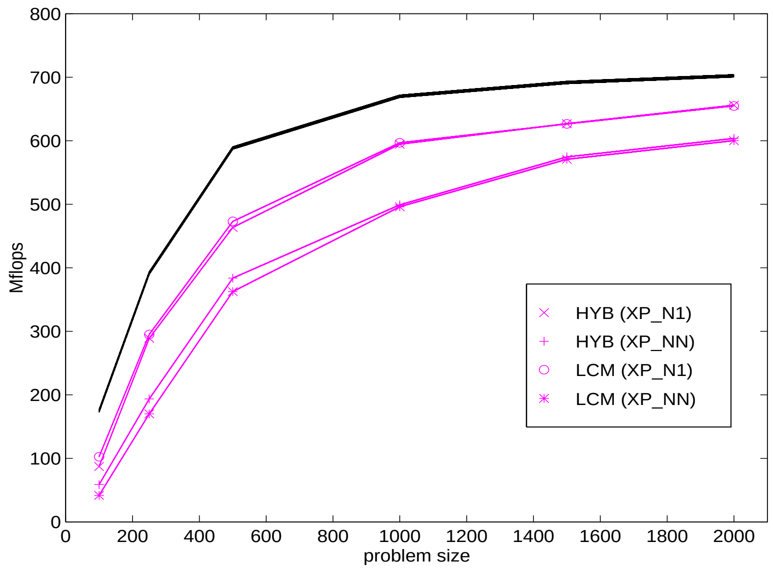


Figure 5.21: Performance of non-aligned hybrid (HYB) versus LCM blocking techniques on a 4×4 Intel XP/S Paragon

blocking variants for the non-aligned experiments on the Intel XP/S Paragon. This figure shows that almost no performance difference exists between these two variants on this platform, as opposed to the results shown in Figures 5.18 and 5.20 for the IBM SP. On this latter platform, the communication-computation performance ratio is less favorable to algorithmically redistributed operations. Consequently, it is in general beneficial to redistribute beforehand. This argument is particularly pertinent for small and medium sized matrices for two reasons. First, such redistribution operations require little workspace. Second, for large operands, the computational cost will dominate no matter which strategy is used to redistribute the operands.

Chapter 6

Conclusions

Il faut, autant qu'on peut, obliger tout le monde :

On a souvent besoin d'un plus petit que soi.

De cette vérité deux fables feront foi,

Tant la chose en preuves abonde.

Entre les pattes d'un Lion

Un Rat sortit de terre assez à l'étourdie ...

Jean de La Fontaine (1621-1695)

Performing a finite sequence of rank-k updates is the basic underlying operation of most modern dense linear algebra algorithms. Computing the numerical solution of linear systems and solving least squares matrix inequalities are traditionally performed on a computer in two steps. First, the linear operator is factorized into a product of two or more matrices featuring suitable properties for the resolution of the problem. Second, the solution of the problem is obtained by solving simpler matrix equalities typically involving triangular and/or orthogonal matrices [5, 49]. This same framework forms the basis of modern algorithms solving algebraic eigenvalue problems. The matrix representing the linear operator is first reduced to a condensed form. The numerical solution is then obtained by applying an iterative method to this condensed form [5, 49]. Block-partitioned algorithms have been developed for most of the matrix factorizations and reductions. These algorithms have been implemented in the LAPACK software library [5] for shared memory systems. The bulk of computation in these algorithms is performed on the matrix representation of the linear operator. When such a matrix is distributed onto a process grid according to the block cyclic scheme, the operands of the elementary rank-k updates feature natural alignment characteristics. A parallel implementation of these algorithms can take advantage of such distribution properties. Parallel basic linear algebra operations such as the matrix-matrix multiply or the triangular solve operations can also be expressed recursively as a succession of themselves and rank-k updates [33, 60]. The algorithms proposed in the literature thus far focus on the naturally aligned cases used in the factorization and reduction operations. This restricted interest prevents one from providing the necessary flexibility that a parallel software library requires to be truly usable. In addition, restricted operations considerably handicap the ease-of-use of such a library since one often needs to reformulate general operations to match obscure alignment restrictions that are difficult to document and to explain.

This dissertation demonstrates that it is possible to alleviate natural alignment restrictions for a low (sometimes negligible) performance cost for basic operations and various block cyclic distributions. Moreover, the techniques used for this purpose considerably reduce and often completely remove the complicated dependence between the performance of parallel basic linear algebra operations and the physical distribution parameters. We believe that the preceding statement is the major contribution of this dissertation. Indeed, it says that the algorithms presented in this document allow one to produce a general purpose and flexible parallel software library of basic linear algebra subprograms. These algorithms have been shown in this document to achieve high performance independently from the actual block cyclic distribution parameters. Efficiency and flexibility are not antagonistic objectives for basic dense linear algebra operations. This result is presented in greater detail in the following sections.

6.1 Application Domain of Algorithmic Operations

The performance results presented in Chapter 5 show that when the matrix operands are aligned, the algorithmically redistributed operations based on aggregation (AGG) and the LCM blocking (LCM) strategy are competitive in terms of performance with the complete redistribution variant (RED). This conclusion must be refined when the matrix operands must be redistributed before the aligned operation can take place. Figure 6.1 summarizes the application domain of algorithmically redistributed operations for the non-aligned cases. First, when the number of processors p is small, the redistribution operations are considerably simplified because the total number of messages to be exchanged during such an operation is proportional to p^2. Therefore, in these cases, the algorithmically re-

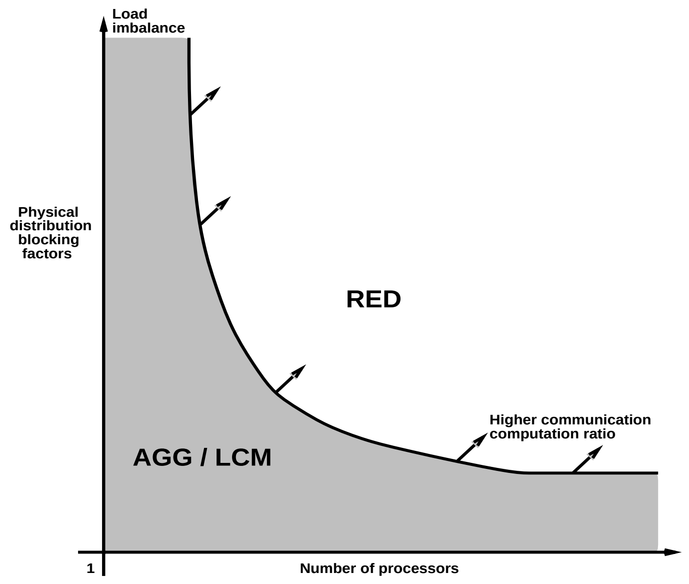

Load imbalance
Physical distribution blocking factors
RED
AGG / LCM
Higher communication computation ratio
1
Number of processors

Figure 6.1: Application domain of algorithmically redistributed operations

distributed operations are highly efficient. Their performance is very similar to the performance obtained by the corresponding aligned cases. Similarly, if one restricts oneself to very small values of the physical distribution parameters for all of the matrix operands, the redistribution operations are considerably simplified for the same reason as above. In all of the other cases, there is a tradeoff that depends on the communication-computation performance ratio of the target computer. This tradeoff is symbolized in Figure 6.1 by the curved border of the grey area. This border divides the plane quarter into two distincts areas. The upper right area is denoted by RED, and the other area is colored in grey and denoted by AGG/LCM. Suppose first that the distribution parameters, i.e., number of processors and physical blocking factors, are such that they identify a point in the grey area. In this case, algorithmically redistributed operations based on aggregation and the LCM blocking strategy are highly competitive. These methods are likely to deliver performance within 15 % of the performance achieved for the best aligned case. Second, if the distribution parameters identify a point in the white area denoted by RED, then complete redistribution of the matrix operands beforehand is more efficient than algorithmically redistributed operations. The position of the border separating both regions depends on the communication-computation performance ratio of the target computer. If this ratio increases, the curved border is shifted in the direction indicated by the arrows on the figure. The performance results presented in Chapter 5 show that the machine parameters can be such that this border is never encountered. For example, on the Intel XP/S Paragon, we never found a problem specification such that complete redistribution (RED) overperforms algorithmically redistributed operations such as AGG or LCM. On a machine featuring a less favorable ratio such as the IBM SP, it has been observed that non-aligned data redistribution beforehand (RED) allows for better performance as soon as the process grid and the physical blocking factors are sufficiently large.

6.2 Recommendations for a Software library

The results shown in this dissertation have a direct impact on the eventual production of a set of flexible parallel basic algebra subprograms. Indeed, we have shown that for a variety of distribution and machine parameters one can afford to redistribute the matrix operands "on the fly" without a significant performance degradation. However, for certain distributed memory concurrent computers featuring slow communication performance compared to their computational power, it is necessary to preserve the possibility of redistributing the data beforehand. It has been observed that such a redistribution phase has such a high memory cost that it is impractical for the largest problems fitting in the main physical memory. Such observations indicate that it is worthwhile to provide algorithmically redistributed operations that feature the flexibility that a library user may expect. This allows for fast prototyping and debugging of parallel algorithms. Moreover, the performance of such algorithmically redistributed operations is always higher or comparable to what currently exists. The user should be warned that slightly higher performance may be achieved on certain platforms by redistributing the data beforehand when it is feasible.

In order to address the high memory cost induced by redistributing beforehand, one may think about two distinct approaches. First, instead of redistributing the entire operands at once, it is possible to redistribute say only half of them in two steps. At each step the same workspace can be reused and only part of the computation performed. This approach is viable, even if it is problematic from a software point of view to precisely estimate at run-time the amount of usable memory on each process. Second, it is also possible to redistribute the operands in place. A non trivial algorithm as well as its memory cost could not be found in the literature. However, even if one assumes the availability of such an algorithm, the later remains impractical from a software point of view. Indeed, the size of the local arrays capable of storing the original and redistributed operands highly depends on the distribution parameters of the redistributed operand. These parameters may only be known at run time. Both of these preceding approaches may be attractive for a particular application, but their practical realization seems difficult in a modular software library fashion.

A message-passing program is naturally complex. The experimental programs written for this dissertation are by no means exceptions to this rule. These programs are complicated to write, debug and maintain. These facts have been considered when these experimental programs were designed. First, whenever possible, a "global interface" has been selected as used by the ScaLAPACK library and explained in [18]. If such an interface imposes some redundant index computations, it allows for the reuse of sequential data and control structures that are easier to write and debug. Second, the properties shown in Chapter 2 were used to verify and assert the correctness of the experimental programs. For example, a subprogram computing the number of diagonals owned by a particular process is not a trivial programming exercise. Such a task requires a good understanding of the data distribution properties. It is, however, easy to check the validity of the result. Sequential unit testing programs were thus developed for almost all of the subprograms computing indexes and local quantities. Writing a parallel program is often considered as an implementation detail when compared with the design of the algorithms. The complexity of a program is after all a subjective matter, as opposed to the complexity of an algorithm. There is undoubtedly some truth in such a statement, even if it is overlooking the software engineering aspects of distributed memory programming.

6.3 Contributions of this Dissertation

A number of properties of the block cyclic distribution were formally exhibited in Chapter 2. These properties form the theoretical basis of a characterization of the block cyclic distribution. They have been used to develop and ensure the correctness of algorithmically redistributed operations, as well as the robustness and reliability of their experimental implementation. This collection of properties naturally suggests an elegant and convenient data structure that encapsulates and reveals the essential features of the LCM block partitioning unit when used in the context of algorithmic redistributed operations. The LCM table definition was thus derived and shown to be a convenient framework for expressing algorithmically redistributed operations. It was noted that this approach can be generalized to the family of Cartesian mappings. The relationship between the distribution parameters and the complexity of the general one- and two-dimensional redistribution operations was determined and presented in Corollary 2.5.1. The intuitive result that the complexity of these operations increases with the perimeter of the $r \times s$ partitioning unit was proved for a finite range of possible and realistic values of the distribution parameters.

Most of the parallel algorithms proposed in the literature rely on the physical blocking strategy to efficiently use a distributed memory hierarchy. Within the restricted context of the rank-k update operation, algorithmically redistributed operations were thus introduced and presented as alternatives to the physical blocking strategy. The originality of the algorithms presented in Chapter 3 is their systematic derivation from the properties of the underlying mapping. These blocking strategies were expressed within a single framework using LCM tables. It was noted that the modular design of the resulting operations was appropriate for library software. Indeed, algorithmically redistributed operations feature a potential for high performance without the alignment restrictions of the physically blocked algorithms. Defining and studying algorithmically redistributed operations attempts to show that the antagonism between efficiency and flexibility is not a property of the block cyclic mapping, but merely a characteristic of the algorithms that have been so far proposed to deal with a distributed memory hierarchy.

The scalability of the algorithmically redistributed operations proposed in Chapter 3 is studied in Chapter 4. The performance analysis of these operations is presented for a simplified theoretical machine model. It is shown for this machine model that these operations are *scalable*, i.e., the parallel efficiency defined by Equation 4.4.1 is maintained if the memory use per process is kept constant. Experimental performance results are presented in Chapter 5. The experiments are conducted on two platforms featuring highly different communication-computation performance ratios. It is observed that when the matrix operands verify certain data alignment properties, algorithmically redistributed operations are competitive, in the sense that high performance can be achieved independently from the distribution parameters of the matrix operands. When these alignment restrictions are not met, the performance achieved by algorithmically redistributed operations is sensitive to the communication-computation performance ratio of the target architecture. For a distributed memory concurrent computer such as the Intel XP/S Paragon featuring a high communication-computation performance ratio, performance comparable to the one obtained in the aligned cases is observed. When the communication-computation performance ratio is lower as it is the case for the IBM SP, redistributing the data beforehand is in general more efficient than algorithmic redistribution.

6.4 Further Research Directions

There is undoubtly a need for a formal characterization of data decompositions. The properties of such mappings as well as their derivation suggest natural parallel algorithms. They also identify critical features of which one can take advantage. For example, periodicity is an essential property since it is the source of efficient blocking techniques. Ease-of-use and flexibility are also factors to be considered. These criteria may seem subjective and difficult to estimate; it is, however, usually the case that one can identify and characterize the differences of specific cases from the general definition. In other words, it is interesting to show that the LU factorization scales and performs well when the data is distributed according to a given mapping. It is, however, more interesting to show what the properties of the data decomposition should be able to produce an efficient implementation of the LU factorization.

The one- and two-dimensional redistribution algorithms presented and used in this dissertation feature an optimal volume of communication. The scheduling policies, however, are not contention free. Developing and/or characterizing such policies for static and dynamic networks is a research area on its own. References on the subject can be found in [32, 67]. More recent research work on this topic can be found in [78, 86]. These problems are complex and difficult to address. The experimental results presented in the literature are usually machine dependent. They are often based on empirical trials or heuristics. A comprehensive and detailed comparative summary of the known scheduling policies could not be found in the literature. Data redistribution can lead to a significant performance improvement. An inadequate scheduling policy can however ruin the performance of the redistribution operation and suggest an incorrect interpretation of experimental results.

Bibliography

Bibliography

[1] M. Aboelaze, N. Chrisochoides, and E. Houstis. The Parallelization of Level 2 and 3 BLAS Operations on Distributed Memory Machines. Technical Report CSD-TR-91-007, Purdue University, West Lafayette, IN, 1991.

[2] R. Agarwal, F. Gustavson, and M. Zubair. A High Performance Matrix Multiplication Algorithm on a Distributed-Memory Parallel Computer, Using Overlapped Communication. *IBM Journal of Research and Development*, 38(6):673–681, 1994.

[3] R. Agarwal, F. Gustavson, and M. Zubair. Improving Performance of Linear Algebra Algorithms for Dense Matrices Using Algorithmic Prefetching. *IBM Journal of Research and Development*, 38(3):265–275, 1994.

[4] T. Agerwala, J. Martin, J. Mirza, D. Sadler, D. Dias, and M. Snir. SP2 System Architecture. *IBM Systems Journal*, 34(2):153–184, 1995.

[5] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du

Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, Second Edition.* SIAM, Philadelphia, PA, 1995.

[6] I. Angus, G. Fox, J. Kim, and D. Walker. *Solving Problems on Concurrent Processors: Software for Concurrent Processors*, volume 2. Prentice Hall, Englewood Cliffs, N.J, 1990.

[7] C. Ashcraft. The Distributed Solution of Linear Systems Using the Torus-wrap Data mapping. Technical Report ECA-TR-147, Boeing Computer Services, Seattle, WA, 1990.

[8] M. Baber. Hypertasking Support for Dynamically Redistributable and Resizeable Arrays on the iPSC. In *Proceedings of the Sixth Distributed Memory Computing Conference*, pages 59–66, 1991.

[9] R. Bisseling and J. van der Vorst. Parallel LU Decomposition on a Transputer Network. In G. van Zee and J. van der Vorst, editors, *Lecture Notes in Computer Sciences*, volume 384, pages 61–77. Springer-Verlag, 1989.

[10] R. Bisseling and J. van der Vorst. Parallel Triangular System Solving on a mesh network of Transputers. *SIAM Journal on Scientific and Statistical Computing*, 12:787–799, 1991.

[11] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a Parallel C++ Runtime System for Scalable Parallel Systems. In *Proceedings of Supercomputing'93*, pages 588–597, 1993.

[12] R. Brent. The LINPACK Benchmark on the AP 1000. In *Frontiers, 1992*, pages 128–135, McLean, VA, 1992.

[13] R. Brent and P. Strazdins. Implementation of BLAS Level 3 and LINPACK Benchmark on the AP1000. *Fujitsu Scientific and Technical Journal*, 5(1):61–70, 1993.

[14] B. Chapman, P. Mehrotra, H. Moritsch, and H. Zima. Dynamic Data Redistribution in Vienna Fortran. In *Proceedings of Supercomputing'93*, pages 284–293, 1993.

[15] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. Tseng. Generating Local Adresses and Communication Sets for Data Parallel Programs. *Journal of Parallel and Distributed Computing*, 26:72–84, 1995.

[16] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance. *Computer Physics Communications*, 97:1–15, 1996. (also LAPACK Working Note #95).

[17] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. Installation Guide for ScaLAPACK. Technical Report UT CS-95-280, LAPACK Working Note #93, University of Tennessee, 1995.

[18] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. A Proposal for a Set of Parallel Basic Linear Algebra Subprograms. In J. Dongarra, K. Masden, and J. Waśniewski, editors, *Applied Parallel Computing*, pages 107–114. Springer Verlag, 1995. (also LAPACK Working Note #100).

[19] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. The Design and Implementation of the Reduction Routines in ScaLAPACK. In J. J. Dongarra, L. Grandinetti, G. R. Joubert, and J. Kowalik, editors, *High Performance Computing: Technology, Methods and Applications*, Advances in Parallel Computing, 10, pages 177–202. Elsevier, Amsterdam, The Netherlands, 1995.

[20] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. *Scientific Programming*, 5:173–184, 1996. (also LAPACK Working Note #80).

[21] J. Choi, J. Dongarra, R. Pozo, and D. Walker. ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers. In *Proceedings of Fourth Symposium on the Frontiers of Massively Parallel Computation (McLean, Virginia)*, pages 120–127. IEEE Computer Society Press, Los Alamitos, California, 1992. (also LAPACK Working Note #55).

[22] J. Choi, J. Dongarra, and D. Walker. Parallel Matrix Transpose Algorithms on Distributed Memory Concurrent Computers. In *Proceedings of Fourth Symposium on the Frontiers of Massively Parallel Computation (McLean, Virginia)*, pages 245–252. IEEE Computer Society Press, Los Alamitos, California, 1993. (also LAPACK Working Note #65).

[23] J. Choi, J. Dongarra, and D. Walker. PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers. *Concurrency: Practice and Experience*, 6(7):543–570, 1994. (also LAPACK Working Note #57).

[24] J. Choi, J. Dongarra, and D. Walker. The Design of a Parallel, Dense Linear Algebra Software Library: Reduction to Hessenberg, Tridiagonal and Bidigonal Form. *Numerical Algorithms*, 10:379–399, 1995.

[25] J. Choi, J. Dongarra, and D. Walker. PB-BLAS: A Set of Parallel Block Basic Linear Algebra Subroutines. *Concurrency: Practice and Experience*, 8(7):517–535, 1996.

[26] A. Chtchelkanova, J. Gunnels, G. Morrow, J. Overfelt, and R. van de Geijn. Parallel Implementation of BLAS: General Techniques for Level 3 BLAS. Technical Report TR95-49, Department of Computer Sciences, UT-Austin, 1995. Submitted to Concurrency: Practice and Experience.

[27] E. Chu and A. George. QR Factorization of a Dense Matrix on a Hypercube Multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 11:990–1028, 1990.

[28] Mathemetical Committee on Physical and Engineering Sciences, editors. *Grand Challenges: High Performance Computing and Communications*. NSF/CISE, 1800 G Street NW, Washington, DC, 20550, 1991.

[29] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT press, Cambridge, MA, 1990.

[30] IBM Corporation. IBM RS6000. (http://www.rs6000.ibm.com/), 1996.

[31] Intel Corporation. Intel Supercomputer Technical Publications Home Page. (http://www.ssd.intel.com/pubs.html), 1995.

[32] M. Cosnard, Y. Robert, P. Quinton, and M. Tchuente, editors. *Parallel Algorithms and Architectures*. North-Holland, 1986.

[33] M. Dayde, I. Duff, and A. Petitet. A Parallel Block Implementation of Level 3 BLAS for MIMD Vector Processors. *ACM Transactions on Mathematical Software*, 20(2):178–193, 1994.

[34] G. Lejeune Dirichlet. Abhandlungen Königlich Preuss. Akad. Wiss., 1849.

[35] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[36] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. Algorithm 656: An extended Set of Basic Linear Algebra Subprograms: Model Implementation and Test Programs. *ACM Transactions on Mathematical Software*, 14(1):18–32, 1988.

[37] J. Dongarra and R. van de Geijn. Two dimensional Basic Linear Algebra Communication Subprograms. Technical Report UT CS-91-138, LAPACK Working Note #37, University of Tennessee, 1991.

[38] J. Dongarra, R. van de Geijn, and D. Walker. Scalability Issues in the Design of a Library for Dense Linear Algebra. *Journal of Parallel and Distributed Computing*, 22(3):523–537, 1994. (also LAPACK Working Note #43).

[39] J. Dongarra and D. Walker. Software Libraries for Linear Algebra Computations on High Performance Computers. *SIAM Review*, 37(2):151–180,

1995.

[40] J. Dongarra and R. C. Whaley. A User's Guide to the BLACS v1.0. Technical Report UT CS-95-281, LAPACK Working Note #94, University of Tennessee, 1995. (http://www.netlib.org/blacs/).

[41] R. Falgout, A. Skjellum, S. Smith, and C. Still. The Multicomputer Toolbox Approach to Concurrent BLAS and LACS. In *Proceedings of the Scalable High Performance Computing Conference SHPCC-92*. IEEE Computer Society Press, 1992.

[42] M. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, 1972.

[43] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3–4), 1994.

[44] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface (Draft). (http://www.mcs.anl.gov/mpi), 1996.

[45] G. Fox, S. Hiranandani, K. Kennedy, C. Koebel, U. Kremer, C. Tseng, and M. Wu. Fortran D Language Specification. Technical Report TR90-141, Rice University, Department of Computer Science, 1990.

[46] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice Hall, Englewood Cliffs, N.J, 1988.

[47] G. Fox, S. Otto, and A. Hey. Matrix Algorithms on a Hypercube I: Matrix Multiplication. *Parallel Computing*, 3:17–31, 1987.

[48] G. Geist and C. Romine. LU Factorization Algorithms on Distributed Memory Multiprocessor Architectures. *SIAM Journal on Scientific and Statistical Computing*, 9:639–649, 1988.

[49] G. Golub and C. van Loan. *Matrix Computations*. Johns-Hopkins, Baltimore, second edition, 1989.

[50] M. Hall, S. Hiranandani, K. Kennedy, and C. Tseng. Interprocedural Compilation of Fortran D for MIMD Machines. In *Proceedings of Supercomputing'92*, pages 522–534, 1992.

[51] P. Hatcher and M. Quinn. *Data-Parallel Programming On MIMD Computers*. The MIT Press, Cambridge, Massachusetts, 1991.

[52] M. Heath and C. Romine. Parallel Solution Triangular Systems on Distributed Memory Multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, 9:558–588, 1988.

[53] B. Hendrickson, E. Jessup, and C. Smith. A Parallel Eigensolver for Dense Symmetric Matrices. Personal communication, 1996.

[54] B. Hendrickson and D. Womble. The Torus–wrap Mapping for Dense Matrix Calculations on Massively Parallel Computers. *SIAM Journal on Scientific and Statistical Computing*, 15(5):1201–1226, September 1994.

[55] G. Henry and R. van de Geijn. Parallelizing the QR Algorithm for the Unsymmetric Algebraic Eigenvalue problem: Myths and Reality. Technical Report UT CS-94-244, LAPACK Working Note #79, University of Tennessee, 1994.

[56] S. Hiranandani, K. Kennedy, J. Mellor-Crummey, and A. Sethi. Compilation Techniques for Block-Cyclic Distributions. Technical Report CRPC-TR95521-S, Center for Research on Parallel Computation, 1995.

[57] S. Huss-Lederman, E. Jacobson, A. Tsao, and G. Zhang. Matrix Multiplication on the Intel Touchstone DELTA. *Concurrency: Practice and Experience*, 6(7):571–594, 1994.

[58] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, 1993.

[59] S. L. Johnsson. Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures. *Journal of Parallel and Distributed Computing*, 2:133–172, 1987.

[60] B. Kågström, P. Ling, and C. van Loan. GEMM-Based Level 3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark. Technical Report UMINF 95-18, Department of Computing Science, Umeå University, 1995. Submitted to ACM TOMS.

[61] E. Kalns. *Scalable Data Redistribution Services for Distributed-Memory Machines*. PhD thesis, Michigan State University, 1995.

[62] W. Kaufmann and L. Smarr. *Supercomputing and the Transformation of Science*. Scientific American Library, 1993.

[63] K. Kennedy, N. Nedeljković, and A. Sethi. Efficient Address Generation For Block-Cyclic Distributions. Technical Report CRPC-TR94485-S, Center for Research on Parallel Computation, 1994.

[64] K. Kennedy, N. Nedeljković, and A. Sethi. A Linear-Time Algorithm for Computing the Memory Access Sequence in Data Parallel Programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, 1995.

[65] D. Knuth. *The Art of Computer Programming*. Addison-Wesley, second edition, 1973. Volume 1. Fundamental algorithms. Volume 2. Semi-numerical algorithms. Volume 3. Sorting and searching.

[66] C. Koebel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *The High Performance Fortran Handbook.* The MIT Press, Cambridge, Massachusetts, 1994.

[67] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[68] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

[69] G. Li and T. Coleman. A Parallel Triangular Solver for a Distributed-Memory Multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 9(3):485–502, 1988.

[70] G. Li and T. Coleman. A New Method for Solving Triangular Systems on Distributed-Memory Message-Passing Multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 10(2):382–396, 1989.

[71] J. Li and M. Chen. The Data Alignment Phase in Compiling Programs for Distributed-Memory Machines. *Journal of Parallel and Distributed Computing*, 13:213–221, 1991.

[72] W. Lichtenstein and S. L. Johnsson. Block-Cyclic Dense Linear Algebra. *SIAM Journal on Scientific and Statistical Computing*, 14(6):1259–1288, 1993.

[73] M. Mace. Memory Storage Patterns in Parallel Processing, 1987.

[74] K. Mathur and S. L. Johnsson. Multiplication of Matrices of Arbitrary Shapes on a Data Parallel Computer. *Parallel Computing*, 20:919–951, 1994.

[75] P. Mehrotra and J. Rosendale. *Programming Distributed Memory Architectures Using Kali*. The MIT Press, Cambridge, Massachusetts, 1991.

[76] Office of Science and Technology Policy, editors. *A Research and Development Strategy for High Performance Computing*. Executive Office of the President, 1987.

[77] Office of Science and Technology Policy, editors. *The Federal High Performance Computing Program*. Executive Office of the President, 1989.

[78] L. Prylli and B. Tourancheau. Efficient Block-Cyclic Data Redistribution. Technical Report 2766, INRIA, Rhône-Alpes, 1996.

[79] M. Rosing, R. Schnabel, and R. Weaver. The DINO Parallel Programming language. *Journal of Parallel and Distributed Computing*, 13:30–42, 1991.

[80] C. Stunkel, D. Shea, B. Abali, M. Atkins, C. Bender, D. Grice, P. Hochshild, D. Joseph, B. Nathanson, R. Swetz, R. Stucke, M. Tsao, and P. Varker. The SP2 High-Performance Switch. *IBM Systems Journal*, 34(2):185–204, 1995.

[81] Thinking Machines Corporation. *CMSSL for Fortran*, 1990.

[82] A. Thirumalai and J. Ramanujam. Fast Address Sequence Generation for Data Parallel Programs Using Integer Lattices. In P. Sadayappan and al., editors, *Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science*. Springer Verlag, 1996.

[83] R. van de Geijn and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. Technical Report UT CS-95-286, LAPACK Working Note #96, University of Tennessee, 1995.

[84] E. van de Velde. Data Redistribution and Concurrency. *Parallel Computing*, 16:125–138, 1990.

[85] E. van de Velde. Experiments with Multicomputer LU-Decomposition. *Concurrency: Practice and Experience*, 2:1–26, 1990.

[86] D. Walker and S. Otto. Redistribution of Block-Cyclic Data Distributions Using MPI. *Concurrency: Practice and Experience*, 8(9):707–728, 1996.

[87] R. C. Whaley. Basic Linear Algebra Communication Subprograms: Analysis and Implementation Across Multiple Parallel Architectures. Technical Report UT CS-94-234, LAPACK Working Note #73, University of Tennessee, 1994.

[88] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran: A Language Specification (Version 1.1), 1991.

Appendix

Appendix A

LCM Tables

A.1 LCM Table with First Partial Block

In order to take into account a first partial block of size (ir, is) into the data distribution parameters, the LCM table definition (2.5.2) needs to be modify as given below. For the sake of simplicity, the C conditional operator notation is used. The complementary information provided here may seem slightly more complicated than what is presented in the chapter 2. However, the differences depend only on the process coordinates and therefore do not impact significantly the computation of the LCM table in each process.

Definition A.1.1 Given a k-diagonal, the k-LCM table (LCMT) is a two-dimensional infinite array of integers local to each process (p,q) defined recursively by

$$\begin{cases} LCMT^{p,q}_{0,0} = (q > 0\,?\,is + (q \Leftrightarrow 1)s : 0) \Leftrightarrow (p > 0\,?\,ir + (p \Leftrightarrow 1)r : 0) + k, \\ LCMT^{p,q}_{1,*} = LCMT^{p,q}_{0,*} \Leftrightarrow P\ r \Leftrightarrow (p > 0\,?\,0 : ir \Leftrightarrow r), \\ LCMT^{p,q}_{l,*} = LCMT^{p,q}_{l-1,*} \Leftrightarrow P\ r, \text{ for } l \geq 2, \\ LCMT^{p,q}_{*,1} = LCMT^{p,q}_{*,0} + Q\ s + (q > 0\,?\,0 : is \Leftrightarrow s), \\ LCMT^{p,q}_{*,m} = LCMT^{p,q}_{*,m-1} + Q\ s \text{ for } m \geq 2. \end{cases}$$

Finally, the bounds (2.5.20) against which an LCM table entry are compared with in order to recognize a block that own k-diagonals depend on the process coordinates (p,q) and need to be modified as shown in table A.1.

Table A.1: The LCMT bounds characterizing block owning k-diagonals

$p = 0$ and $q = 0$.	$1 - is \leq LCMT^{p,q}_{0,0} \leq ir - 1,$ $1 - s \leq LCMT^{p,q}_{0,m} \leq ir - 1,$ for $m > 0,$ $1 - is \leq LCMT^{p,q}_{l,0} \leq r - 1,$ for $l > 0,$ $1 - s \leq LCMT^{p,q}_{l,m} \leq r - 1,$ for $l, m > 0.$
$p = 0$ and $q > 0$.	$1 - s \leq LCMT^{p,q}_{0,*} \leq ir - 1,$ $1 - s \leq LCMT^{p,q}_{l,*} \leq r - 1$ for $l > 0.$
$p > 0$ and $q = 0$.	$1 - is \leq LCMT^{p,q}_{*,0} \leq r - 1,$ $1 - s \leq LCMT^{p,q}_{l,m} \leq r - 1,$ for $m > 0.$
$p > 0$ and $q > 0$.	$1 - s \leq LCMT^{p,q}_{l,m} \leq r - 1.$

A.2 Examples of LCM tables

The example shown in figures A.1 and A.2 presents a distribution example where some processes have two blocks locally adjacent that own k-diagonals. For instance, the blocks of coordinates $(0,0)$ and $(1,0)$ in the process $(0,0)$ own exactly one entry of the k-diagonal.

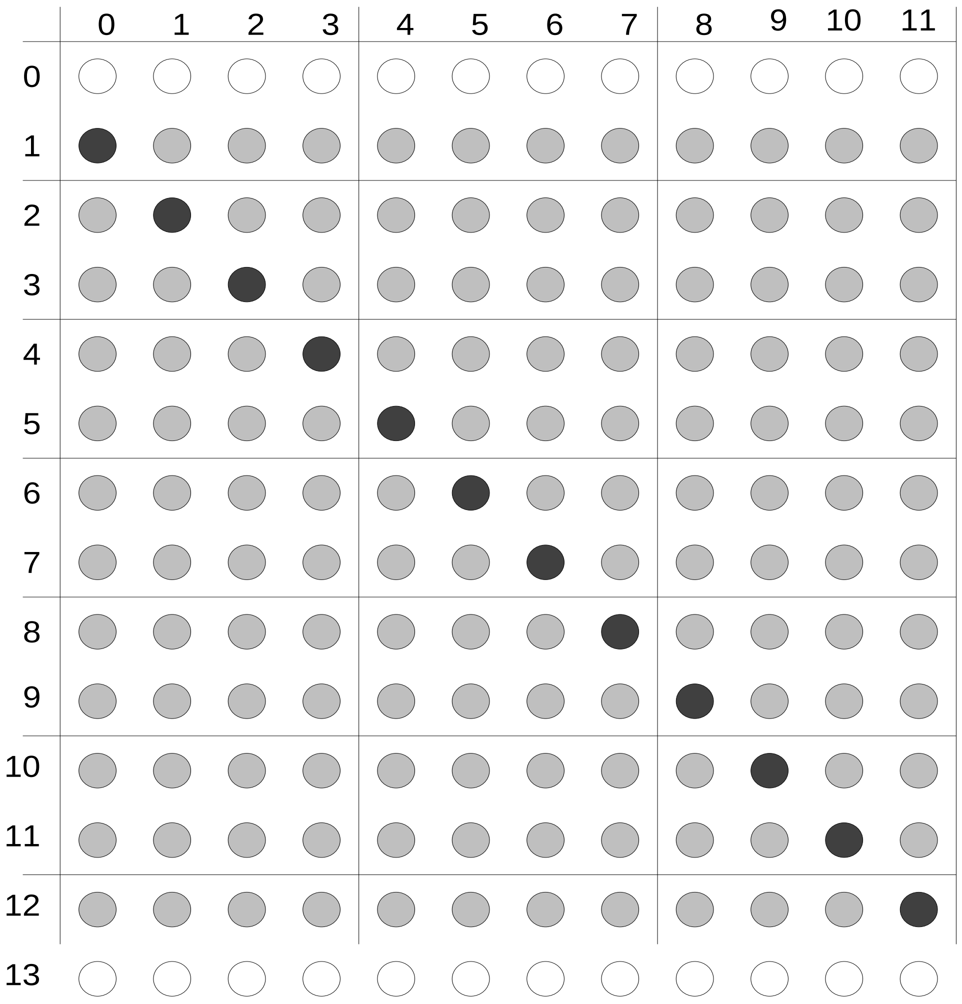


Figure A.1: The 1-LCM block obtained for $P = 2$, $r = 2$, $Q = 3$ and $s = 4$.

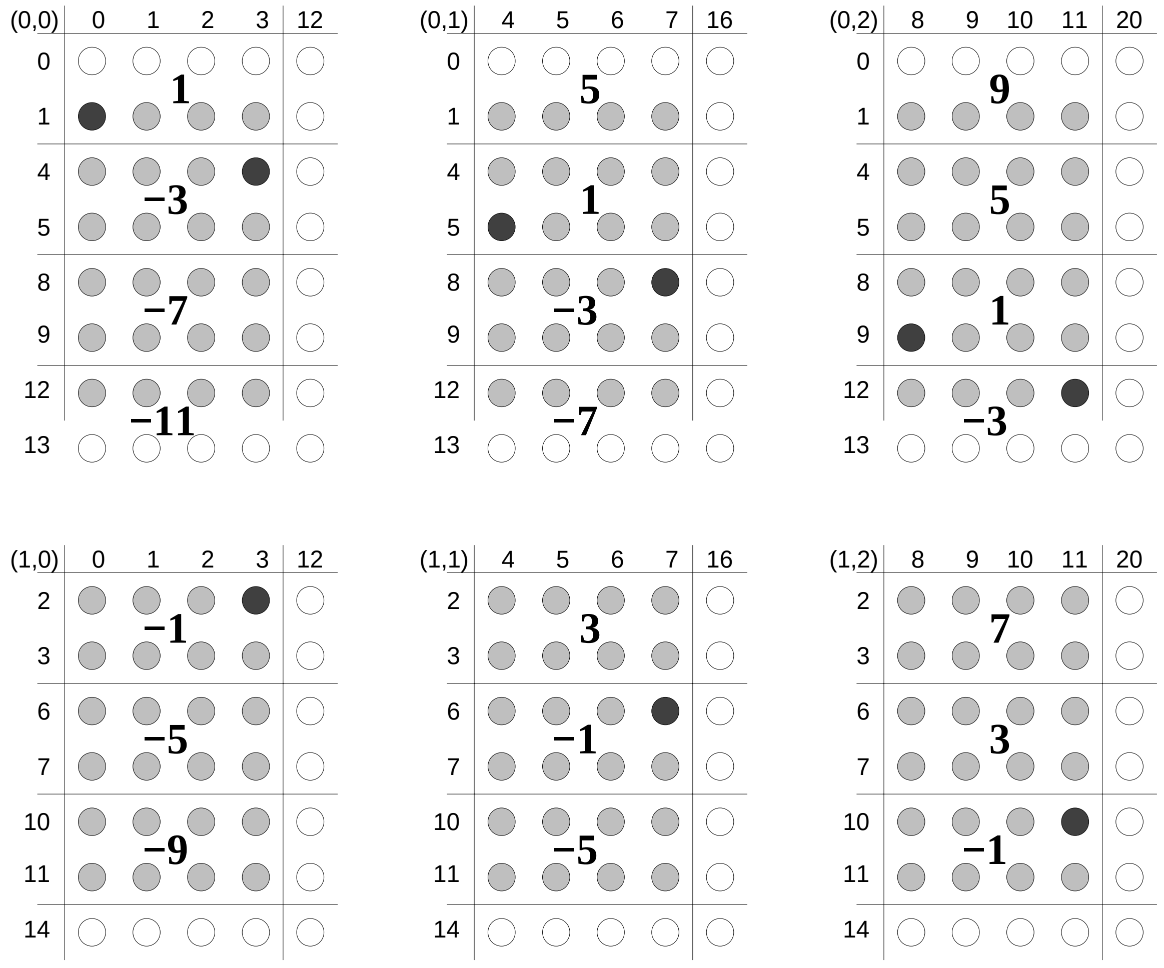


Figure A.2: The 1-LCM tables obtained for $P = 2$, $r = 2$, $Q = 3$ and $s = 4$.

The example shown in figures A.3 and A.4 presents a distribution example with a first partial block. The value of $LCMT_{0,0}^{2,0} = \Leftrightarrow 3$ is compared to $1 \Leftrightarrow is = \Leftrightarrow 2$ instead of $1 \Leftrightarrow s = \Leftrightarrow 3$. Therefore this block does not contain k-diagonals as one can easily check by looking at the figures.

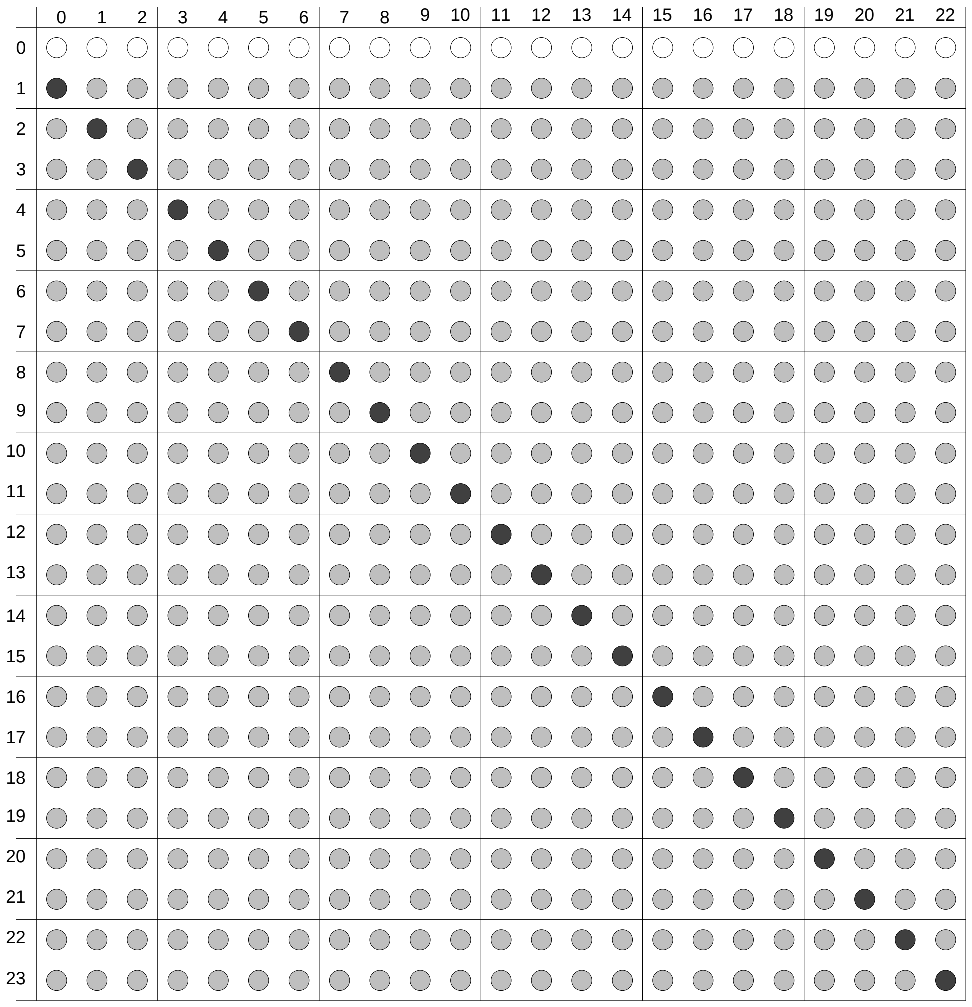


Figure A.3: The 1-LCM block obtained for $P = 3$, $ir = 2$, $r = 2$, $Q = 2$, $is = 3$ and $s = 4$.

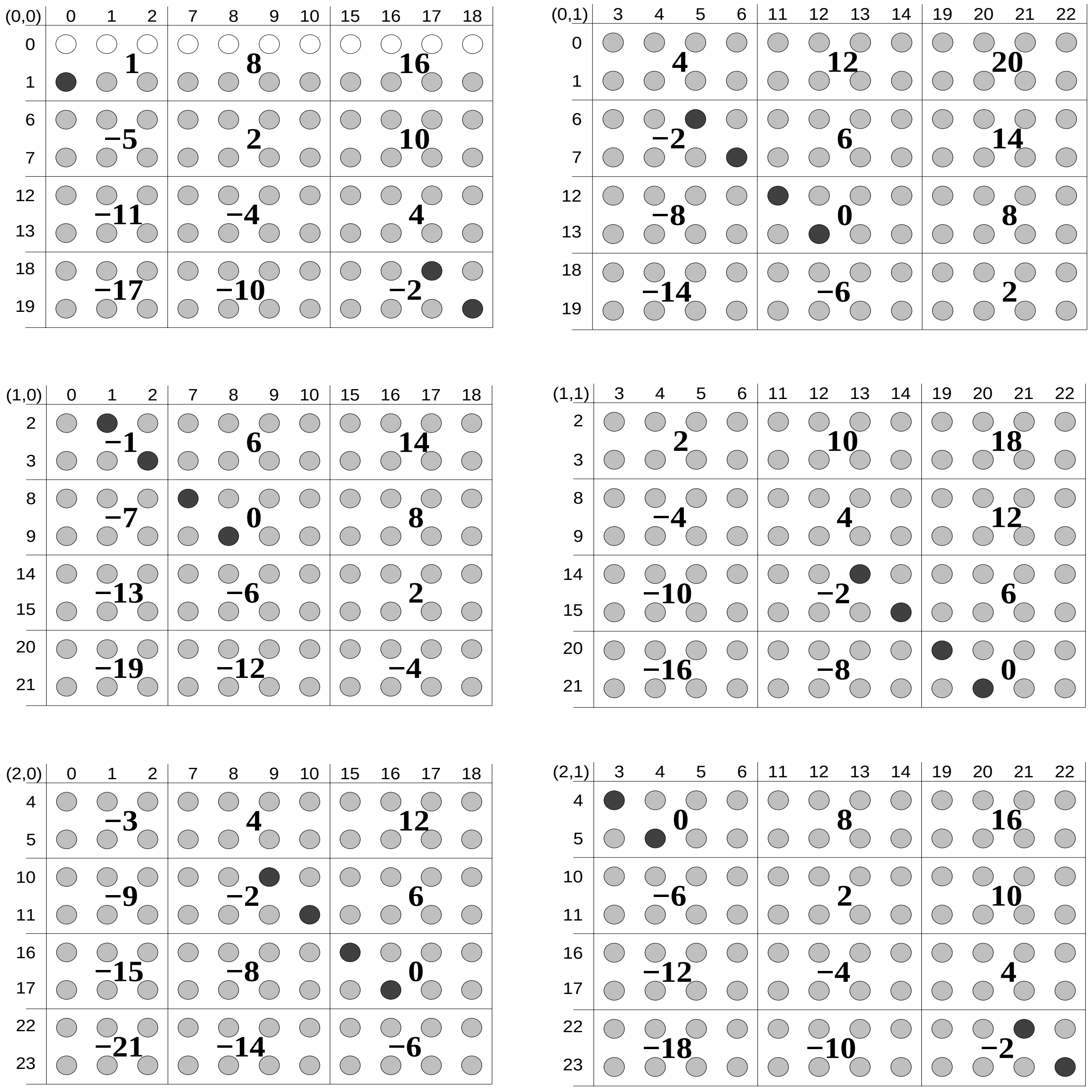


Figure A.4: The 1-LCM tables obtained for $P = 2$, $ir = 2$, $r = 2$, $Q = 2$, $is = 3$ and $s = 4$.

Appendix B

Performance Results

The Tables B.1 and B.2 presents most of the performance results obtained for this dissertation. These results have been used to draw all the plots shown in Chapter 5. The results are for both of the selected target machines, namely the Intel XP/S Paragon (see Section 4.3.1) and IBM SP (see Section 4.3.2), separately. Moreover, for each machine, the results are presented for the experiments specified in Table 5.2. All these experiments have been performed in double precision arithmetic. The matrix operands have been randomly generated. Finally, the three matrix operands were square of order N (See Section 5.3 for more details).

Table B.1: Performance results obtained on the Intel XP/S Paragon

Experiment # XP_A0		N						
Grid	Variant	100	250	500	1000	1500	2000	3000
1×2	PHY	56.4	79.3	85.8	-	-	-	-
	AGG	56.1	78.9	85.8	-	-	-	-
	LCM	55.7	78.8	85.7	-	-	-	-
2×2	PHY	89.0	150.7	168.6	175.9	-	-	-
	AGG	89.6	149.6	168.8	175.9	-	-	-
	LCM	90.3	147.7	168.3	176.0	-	-	-
2×4	PHY	124.9	241.9	309.7	337.6	348.6	-	-
	AGG	121.7	239.7	308.5	337.5	348.5	-	-
	LCM	129.5	239.3	312.1	339.8	350.4	-	-
4×4	PHY	169.8	376.9	578.4	662.0	685.3	697.2	-
	AGG	168.7	375.2	573.8	661.0	684.8	696.8	-
	LCM	174.7	391.8	588.6	670.1	691.7	702.2	-
4×8	PHY	211.7	526.1	917.5	1232.2	1266.9	1346.2	1393.4
	AGG	191.8	506.9	907.3	1224.7	1266.5	1345.4	1393.3
	LCM	176.5	540.7	934.6	1244.7	1281.0	1357.9	1401.9
Experiment # XP_A1								
1×2	PHY	27.8	37.6	41.8	-	-	-	-
	AGG	54.1	76.9	84.6	-	-	-	-
	LCM	54.8	75.6	81.9	-	-	-	-
	RED	40.6	70.1	80.3	-	-	-	-
2×2	PHY	36.3	64.3	78.2	85.2	-	-	-
	AGG	74.9	125.7	154.7	168.2	-	-	-
	LCM	84.9	137.0	159.6	169.3	-	-	-
	RED	44.4	92.6	128.6	21.3	-	-	-
2×4	PHY	35.2	99.3	139.5	163.7	171.3	-	-
	AGG	71.5	178.5	259.7	307.2	328.6	-	-
	LCM	102.2	226.6	300.7	327.2	341.8	-	-
	RED	52.4	150.4	230.9	286.7	27.3	-	-
4×4	PHY	29.4	132.3	230.4	300.6	323.7	336.6	-
	AGG	74.5	229.9	378.7	526.2	576.6	614.3	-
	LCM	121.7	384.2	545.8	640.4	657.4	678.6	-
	RED	63.4	234.9	411.4	558.7	605.3	63.1	-
4×8	PHY	24.5	150.8	352.0	534.4	608.5	645.9	681.9
	AGG	66.6	249.1	491.2	854.5	1006.2	1113.4	1229.4
	LCM	148.1	512.9	906.1	1201.6	1252.6	1309.7	1368.1
	RED	59.3	312.1	677.8	1022.8	1140.0	1223.1	43.2

Experiment # XP_A10		*N*						
Grid	Variant	100	250	500	1000	1500	2000	3000
1×2	PHY	62.1	77.0	85.3	-	-	-	-
	AGG	56.6	75.6	85.5	-	-	-	-
	LCM	58.4	76.4	85.2	-	-	-	-
	RED	46.7	68.1	80.6	-	-	-	-
2×2	PHY	102.5	141.7	170.8	176.8	-	-	-
	AGG	80.3	129.4	163.1	174.3	-	-	-
	LCM	87.3	141.1	166.3	175.9	-	-	-
	RED	62.9	114.6	155.8	12.9	-	-	-
2×4	PHY	122.8	233.6	304.0	339.3	348.7	-	-
	AGG	110.2	212.2	288.7	332.0	343.3	-	-
	LCM	116.8	234.3	300.3	339.9	348.1	-	-
	RED	75.8	194.9	277.4	322.8	13.5	-	-
4×4	PHY	148.1	381.0	544.8	672.3	679.2	701.0	-
	AGG	132.7	337.3	506.4	641.0	665.6	693.2	-
	LCM	128.9	394.0	554.7	668.1	683.1	705.6	-
	RED	76.0	305.9	494.4	634.5	655.4	45.5	-
4×8	PHY	143.7	553.6	900.2	1205.2	1290.4	1351.3	1389.7
	AGG	114.5	470.6	832.5	1139.1	1252.7	1327.5	1366.7
	LCM	147.6	549.8	916.0	1202.4	1300.1	1362.8	1394.4
	RED	73.3	389.2	795.7	1139.3	1246.4	1316.7	93.1
Experiment # XP_A40								
1×2	PHY	59.2	81.2	84.3	-	-	-	-
	AGG	53.1	76.7	83.0	-	-	-	-
	LCM	53.0	76.9	83.0	-	-	-	-
	RED	41.9	68.8	76.4	-	-	-	-
2×2	PHY	91.2	150.7	164.2	166.6	-	-	-
	AGG	75.7	138.3	157.9	164.0	-	-	-
	LCM	80.2	142.1	159.7	164.7	-	-	-
	RED	47.8	110.2	147.6	13.4	-	-	-
2×4	PHY	113.2	224.6	295.0	303.8	334.2	-	-
	AGG	97.9	202.7	277.0	296.6	329.5	-	-
	LCM	96.9	205.4	280.9	298.0	330.8	-	-
	RED	56.6	147.5	258.3	284.6	18.7	-	-
4×4	PHY	144.6	335.9	533.0	572.0	630.6	660.2	-
	AGG	119.7	299.4	495.1	552.0	620.7	654.2	-
	LCM	120.9	305.8	503.8	556.6	624.2	656.9	-
	RED	65.0	245.5	459.8	532.6	606.9	40.2	-
4×8	PHY	132.6	525.9	835.2	968.9	1203.2	1197.8	1318.8
	AGG	114.4	474.8	769.7	920.5	1182.1	1183.2	1312.5
	LCM	106.6	462.8	770.9	927.3	1190.3	1189.4	1317.6
	RED	59.6	344.7	678.2	889.6	1139.7	1151.0	48.7

Experiment # XP_A100		N						
Grid	Variant	100	250	500	1000	1500	2000	3000
1×2	PHY	38.5	71.4	73.5	-	-	-	-
	AGG	34.9	67.5	72.4	-	-	-	-
	LCM	34.8	67.5	72.4	-	-	-	-
	RED	27.3	60.8	62.2	-	-	-	-
2×2	PHY	37.4	116.2	124.2	178.5	-	-	-
	AGG	33.8	108.5	120.4	176.6	-	-	-
	LCM	33.7	109.3	120.8	176.8	-	-	-
	RED	27.6	96.3	113.4	20.0	-	-	-
2×4	PHY	37.4	162.0	183.5	290.9	216.9	-	-
	AGG	34.0	151.1	175.5	287.8	301.7	-	-
	LCM	33.8	150.9	175.5	288.0	310.8	-	-
	RED	27.0	132.6	165.5	270.8	25.1	-	-
4×4	PHY	37.5	228.4	271.9	489.0	617.9	707.8	-
	AGG	33.9	214.5	259.5	480.3	616.4	707.2	-
	LCM	33.5	214.9	259.5	481.0	617.7	708.5	-
	RED	26.9	179.2	245.1	459.6	592.9	77.1	-
4×8	PHY	37.4	220.9	490.4	713.8	1188.2	1155.8	696.1
	AGG	33.9	214.3	468.8	699.7	1180.0	1153.5	1109.0
	LCM	33.6	213.0	465.3	698.1	1181.1	1154.9	1247.7
	RED	25.2	167.0	429.3	667.8	1130.9	1116.6	90.6
Experiment # XP_NA								
1×2	PHY	44.0	67.6	74.9	-	-	-	-
	AGG	50.0	74.8	82.0	-	-	-	-
	LCM	50.2	75.7	81.7	-	-	-	-
	RED	38.5	68.6	74.9	-	-	-	-
2×2	PHY	59.5	119.0	140.5	150.3	-	-	-
	AGG	67.2	127.7	150.7	160.6	-	-	-
	LCM	73.8	137.0	157.0	163.4	-	-	-
	RED	47.9	121.5	147.0	21.0	-	-	-
2×4	PHY	53.8	165.1	240.7	269.2	300.0	-	-
	AGG	77.3	177.3	254.4	284.8	320.3	-	-
	LCM	68.6	195.3	274.5	294.9	327.7	-	-
	RED	66.0	176.0	257.8	284.9	24.1	-	-
4×4	PHY	78.9	231.9	413.9	490.5	561.3	595.7	-
	AGG	94.0	254.0	441.9	522.1	599.2	638.5	-
	LCM	86.0	272.1	479.7	548.2	617.2	650.2	-
	RED	71.7	249.1	452.4	529.3	602.3	51.0	-
4×8	PHY	79.9	328.2	604.1	809.4	1040.6	1063.7	1182.5
	AGG	89.5	380.4	669.1	857.1	1119.4	1141.3	1275.6
	LCM	64.3	381.5	712.3	911.0	1173.1	1172.9	1302.2
	RED	59.3	283.5	628.6	882.4	1133.2	1140.8	75.5

Experiment # XP_N1		*N*						
Grid	Variant	100	250	500	1000	1500	2000	3000
1×2	PHY	19.7	33.7	40.3	-	-	-	-
	AGG	41.2	71.0	81.6	-	-	-	-
	LCM	45.4	69.3	79.0	-	-	-	-
	RED	44.7	69.0	43.5	-	-	-	-
2×2	PHY	19.5	57.8	75.7	84.6	-	-	-
	AGG	57.4	116.3	148.6	164.7	-	-	-
	LCM	70.0	124.9	152.8	165.6	-	-	-
	RED	43.2	97.9	133.9	11.2	-	-	-
2×4	PHY	25.6	87.9	131.8	160.6	169.9	-	-
	AGG	70.0	161.7	245.5	297.8	321.5	-	-
	LCM	84.5	191.3	274.7	312.6	331.2	-	-
	RED	51.8	150.5	227.7	282.7	19.3	-	-
4×4	PHY	25.3	114.1	217.2	289.7	317.5	331.2	-
	AGG	72.3	214.8	366.2	526.7	579.3	616.2	-
	LCM	102.5	295.2	473.0	596.8	626.5	655.0	-
	HYB	87.2	289.0	463.5	594.6	626.9	656.0	-
	RED	61.1	227.5	399.7	544.9	588.5	40.2	-
4×8	PHY	17.7	109.8	284.8	470.0	562.6	610.0	658.3
	AGG	61.7	235.0	476.7	797.1	945.2	1062.6	1187.2
	LCM	91.4	332.5	652.7	1004.1	1112.1	1195.6	1285.1
	RED	60.4	315.4	690.9	1035.2	1150.4	1232.0	54.9
Experiment # XP_NN								
1×2	PHY	24.8	53.9	67.4	-	-	-	-
	AGG	39.9	67.0	77.9	-	-	-	-
	LCM	39.1	66.7	77.0	-	-	-	-
	RED	41.2	68.4	78.1	-	-	-	-
2×2	PHY	28.5	78.5	113.3	135.4	-	-	-
	AGG	43.5	101.2	131.3	148.8	-	-	-
	LCM	46.6	105.2	136.1	151.5	-	-	-
	RED	54.4	119.9	145.2	23.1	-	-	-
2×4	PHY	23.9	94.1	173.9	230.6	270.7	-	-
	AGG	46.4	132.2	211.8	257.1	300.0	-	-
	LCM	54.3	136.4	228.8	272.1	310.5	-	-
	RED	64.5	175.4	258.5	282.7	28.8	-	-
4×4	PHY	28.6	113.2	255.1	401.4	489.4	534.3	-
	AGG	55.3	178.2	351.1	468.6	555.5	594.3	-
	LCM	41.7	170.1	362.6	496.1	570.8	600.2	-
	HYB	58.7	193.6	383.8	499.1	574.5	603.7	-
	RED	66.5	243.6	454.0	530.0	604.0	61.3	-
4×8	PHY	26.3	110.7	283.8	557.7	763.8	870.8	1033.0
	AGG	46.9	210.8	456.8	723.3	946.2	1022.4	1166.3
	LCM	24.7	179.2	427.0	765.4	959.8	1053.3	1203.2
	RED	42.9	302.2	644.5	885.3	1126.6	1131.2	104.8

Table B.2: Performance results obtained on the IBM SP

Experiment # SP_A0		N						
Grid	Variant	100	250	500	1000	1500	2000	3000
1×2	PHY	146.9	274.2	321.1	379.8	-	-	-
	AGG	154.0	276.1	321.8	375.8	-	-	-
	LCM	154.3	275.2	323.7	376.0	-	-	-
2×2	PHY	171.3	419.2	526.6	711.9	744.2	-	-
	AGG	172.2	390.8	525.5	705.3	725.3	-	-
	LCM	174.8	384.4	529.1	690.0	729.0	-	-
2×4	PHY	150.0	498.6	772.9	1103.1	1252.4	1368.9	-
	AGG	162.9	504.6	750.7	1082.7	1249.0	1363.5	-
	LCM	166.1	465.6	800.2	1125.3	1241.3	1379.6	-
4×4	PHY	154.0	500.2	1092.9	1706.6	2091.5	2349.1	2683.9
	AGG	149.7	659.4	1158.7	1710.4	1942.6	2316.2	2691.2
	LCM	131.0	594.6	1234.8	1795.5	1990.0	2415.0	2653.1
4×8	PHY	106.8	406.2	1544.7	2454.2	3192.2	3876.2	4482.4
	AGG	115.4	570.6	1541.9	2443.3	3086.1	3826.9	4374.5
	LCM	111.0	525.3	1382.2	2696.3	3396.8	3921.3	4506.9
Experiment # SP_A1								
1×2	PHY	56.6	42.9	48.0	50.1	-	-	-
	AGG	147.9	253.0	324.4	363.0	-	-	-
	LCM	155.5	269.2	328.6	352.4	-	-	-
	RED	91.1	198.1	261.8	331.2	-	-	-
2×2	PHY	59.4	67.5	88.8	95.6	100.0	-	-
	AGG	153.7	348.3	513.7	597.8	653.2	-	-
	LCM	138.8	320.0	527.9	646.9	643.3	-	-
	RED	94.5	177.1	386.7	534.5	606.7	-	-
2×4	PHY	56.7	136.2	150.6	178.9	184.9	187.4	-
	AGG	123.2	325.8	599.9	893.5	1046.1	1155.5	-
	LCM	147.8	352.5	668.5	1040.3	1180.6	1256.3	-
	RED	106.5	170.1	488.2	854.1	997.1	1132.4	-
4×4	PHY	56.7	167.4	223.5	312.4	347.6	351.1	372.1
	AGG	131.1	399.3	780.6	1020.9	1585.2	1811.7	2107.0
	LCM	174.2	436.9	831.8	1553.4	2044.0	2334.7	2286.9
	RED	120.0	396.7	753.3	1363.0	1528.5	2042.4	2387.9
4×8	PHY	40.5	150.6	306.4	473.2	570.1	582.7	660.9
	AGG	102.1	311.7	622.6	1250.4	1796.4	2261.6	2865.5
	LCM	190.0	569.7	930.5	2032.7	2564.2	3323.4	4229.4
	RED	103.5	462.9	1147.0	1923.1	2794.9	3325.6	4081.0

Experiment # SP_A20		N						
Grid	Variant	100	250	500	1000	1500	2000	3000
1×2	PHY	145.4	237.2	290.6	311.7	-	-	-
	AGG	134.2	258.4	326.3	372.7	-	-	-
	LCM	155.0	279.1	337.6	375.5	-	-	-
	RED	94.6	201.9	282.6	343.3	-	-	-
2×2	PHY	189.5	374.6	480.6	595.6	591.7	-	-
	AGG	160.6	352.9	512.7	683.2	709.5	-	-
	LCM	140.6	340.2	518.7	699.9	710.1	-	-
	RED	118.8	298.4	438.2	612.2	653.4	-	-
2×4	PHY	173.7	469.2	727.1	986.1	1044.2	1108.1	-
	AGG	130.7	227.4	648.9	940.6	1130.7	1230.8	-
	LCM	144.6	391.4	721.9	1061.6	1253.8	1347.6	-
	RED	108.9	350.6	604.4	969.9	1132.7	1248.4	-
4×4	PHY	189.6	634.1	971.9	1544.0	1800.8	2020.5	2161.3
	AGG	125.7	423.1	828.9	1313.1	1783.6	2030.7	2314.4
	LCM	155.8	495.7	927.9	1481.6	2029.1	2446.6	2400.0
	RED	105.8	460.5	951.2	1538.1	1906.2	2200.3	2471.4
4×8	PHY	177.3	704.9	1233.5	2317.5	2765.0	3250.3	3674.7
	AGG	122.3	468.6	929.1	1905.7	2623.8	3144.8	3902.8
	LCM	190.7	537.7	1066.5	2115.1	2574.1	3306.3	4238.8
	RED	48.8	349.6	1050.4	2293.1	2981.7	3515.2	4302.7
Experiment # SP_A200								
1×2	PHY	117.6	202.8	308.2	329.6	-	-	-
	AGG	116.7	197.0	285.7	324.8	-	-	-
	LCM	114.2	203.4	303.6	323.5	-	-	-
	RED	70.0	155.8	251.3	296.5	-	-	-
2×2	PHY	108.8	168.0	476.7	527.8	689.2	-	-
	AGG	99.9	212.3	441.6	510.2	660.1	-	-
	LCM	99.4	204.6	415.8	501.6	666.6	-	-
	RED	67.2	133.8	374.1	451.2	603.7	-	-
2×4	PHY	107.6	231.9	599.0	752.0	1216.0	1221.0	-
	AGG	98.7	201.0	597.9	728.9	1205.4	1214.1	-
	LCM	98.9	203.0	536.3	678.4	1095.0	1134.1	-
	RED	65.6	146.0	459.8	651.3	1042.1	1097.8	-
4×4	PHY	35.9	233.1	729.1	1093.8	2151.3	1896.9	2494.3
	AGG	101.0	191.8	718.7	1018.2	2137.6	1891.7	2420.7
	LCM	84.4	182.0	593.7	900.0	1762.6	1713.0	2303.5
	RED	29.2	158.8	583.4	932.8	1767.0	1689.2	2250.1
4×8	PHY	77.0	233.4	665.2	1550.6	3364.8	2618.2	4431.6
	AGG	89.6	172.7	697.3	1710.7	3515.6	2677.8	4473.4
	LCM	82.3	182.9	576.9	1380.4	2399.7	2038.3	3545.6
	RED	46.7	159.6	575.5	1369.5	2770.6	2335.7	3846.4

Experiment # SP_NA		N						
Grid	Variant	100	250	500	1000	1500	2000	3000
1×2	PHY	82.3	103.0	124.3	117.3	-	-	-
	AGG	128.6	249.4	322.4	356.5	-	-	-
	LCM	160.0	280.8	342.8	363.4	-	-	-
	RED	91.3	199.1	285.5	335.0	-	-	-
2×2	PHY	89.6	136.2	219.3	222.4	235.2	-	-
	AGG	136.5	313.7	509.0	613.6	699.1	-	-
	LCM	123.8	257.4	421.9	574.7	677.4	-	-
	RED	113.4	167.6	448.9	576.8	661.0	-	-
2×4	PHY	102.4	165.3	350.7	387.0	427.4	434.7	-
	AGG	113.9	304.3	597.6	823.6	1038.3	1182.7	-
	LCM	125.2	303.5	524.3	826.9	1047.3	1191.1	-
	RED	97.8	324.8	601.1	909.1	1117.6	1205.7	-
4×4	PHY	102.9	324.2	536.9	687.2	757.0	798.0	850.6
	AGG	104.1	344.8	733.2	1162.4	1600.6	1890.1	2207.1
	LCM	128.5	266.1	624.4	891.9	1545.6	1392.2	1637.4
	RED	98.0	422.1	891.0	1471.5	1865.2	2063.0	2489.3
4×8	PHY	97.7	306.6	775.6	1053.6	1274.3	1341.7	1501.9
	AGG	78.4	218.4	597.9	1210.4	1763.8	2228.6	2895.5
	LCM	89.3	469.8	856.9	1398.7	1986.1	2107.3	3131.0
	RED	80.5	478.9	1179.1	1989.1	2998.1	3422.6	4195.6
Experiment # SP_N1								
1×2	PHY	28.5	38.5	46.1	49.1	-	-	-
	AGG	95.4	190.2	267.9	334.5	-	-	-
	LCM	89.1	190.2	256.8	252.5	-	-	-
	RED	71.5	175.7	250.9	317.8	-	-	-
2×2	PHY	31.3	53.2	77.8	92.1	96.8	-	-
	AGG	107.0	232.6	423.3	541.3	588.8	-	-
	LCM	95.5	245.4	414.8	569.3	589.3	-	-
	RED	75.6	210.9	360.9	517.1	595.3	-	-
2×4	PHY	27.1	76.6	126.5	166.4	175.8	188.8	-
	AGG	104.1	267.7	483.7	764.2	933.5	1056.6	-
	LCM	93.9	246.3	492.5	677.0	734.5	756.1	-
	RED	81.5	289.3	519.0	867.3	1024.2	1162.2	-
4×4	PHY	16.7	99.4	172.4	268.6	317.2	330.7	351.5
	AGG	91.0	342.4	580.5	1091.0	1380.8	1640.7	1950.9
	LCM	90.8	230.7	562.8	1163.2	1571.2	1849.0	1973.4
	RED	101.3	292.5	778.3	1367.5	1741.0	2034.0	2349.3
4×8	PHY	11.1	64.0	176.3	334.3	421.9	612.9	684.5
	AGG	87.6	233.6	556.4	1149.1	1644.9	2051.3	2659.6
	LCM	82.6	252.7	520.7	1155.1	1751.2	2270.4	2892.0
	HYB	62.9	222.5	527.4	1305.4	1870.0	2532.8	3138.3
	RED	41.6	420.3	1202.2	2053.8	2887.3	3435.7	4171.9

Experiment # SP_NN		N						
Grid	Variant	100	250	500	1000	1500	2000	3000
1×2	PHY	39.6	83.9	104.6	108.4	-	-	-
	AGG	84.4	183.7	256.8	320.0	-	-	-
	LCM	72.1	128.9	164.7	184.2	-	-	-
	RED	66.7	179.6	252.8	321.0	-	-	-
2×2	PHY	45.2	116.4	183.1	217.5	217.1	-	-
	AGG	93.3	227.1	369.4	505.3	598.7	-	-
	LCM	80.0	175.2	323.4	480.9	579.3	-	-
	RED	27.7	180.4	404.5	545.0	621.5	-	-
2×4	PHY	38.9	122.3	239.3	348.3	379.1	419.0	-
	AGG	85.7	234.4	438.0	693.5	885.4	1004.8	-
	LCM	70.5	206.3	367.2	647.9	838.1	982.7	-
	RED	24.5	155.8	599.1	868.7	1090.3	1168.4	-
4×4	PHY	38.5	147.4	319.5	525.2	644.9	750.8	776.2
	AGG	86.1	281.5	552.9	949.4	1308.8	1532.8	1912.9
	LCM	54.6	185.7	331.7	515.9	987.2	1009.1	1388.0
	RED	29.1	220.4	785.0	1352.4	1804.7	2068.4	2418.2
4×8	PHY	22.4	71.5	276.5	592.7	812.0	927.2	1206.4
	AGG	62.2	227.2	486.4	1024.1	1452.7	1850.6	2456.2
	LCM	31.0	174.4	329.8	652.0	889.5	1045.1	1960.5
	HYB	61.4	205.5	421.3	1159.1	1722.7	2274.2	2932.7
	RED	23.1	176.9	752.2	1994.9	2823.5	3378.4	4034.7

Vita

Antoine Petitet was born in Neuilly sur Seine, France on April 22, 1966. He received his high school education from the Collège Saint Louis de Gonzague in Paris, France. He graduated in 1984. From 1984 until 1987 he attended the classes préparatoires at the Lycée Janson de Sailly in Paris, France. In June 1987 he entered the Ecole Nationale Supérieure d'Electrotechnique, d'Electronique, d'Informatique et d'Hydraulique de Toulouse (ENSEEIHT), Toulouse, France. In 1990 he received the Engineer of Computer Science degree from the ENSEEIHT and the Diplôme d'Etudes Approfondies in Parallel Architectures and Applied Mathematics (1990) from the Université Paul Sabatier, Toulouse, France. In January 1993 he entered the PhD program in Computer Science at the University of Tennessee, Knoxville.

From November 1990 until March 1992 he did his military service working for the French Nuclear Commission (Commisariat à l'Energie Atomique) as Adviser of the Counselor for nuclear questions at the French Permanent Mission in Vienna, Austria. Working experience had also included several visiting positions in Computer Science laboratories. In 1989-1990 he was a trainee at the European Center for Research and Advanced Education in Scientific Computing (CERFACS) in Toulouse, France. In 1992 he worked as an engineer at the Etablissement Technique Central de l'Armement (ETCA) in Paris, France. During the summer of 1994 he visited the IBM T.J. Watson Research Center, Yorktown Heights, New-

York. During the summer of 1995 he visited the Danish Computing Centre for Research and Education (UNI•C) and the Institute for Mathematical Modeling of the Technical University of Denmark (IMM).

In 1995 Antoine Petitet was awarded a citation for extraordinary professional promise by the Chancellor of the University of Tennessee, Knoxville. He was awarded the Doctor of Philosophy degree in Computer Science from the University of Tennessee in December of 1996.