Algorithmic Redistribution Methods for Block Cyclic Decompositions*

Antoine P. Petitet[†], Jack J. Dongarra[‡]

7th February 1998

Abstract. In a serial computational environment, transportable efficiency is the essential motivation for developing blocking strategies and block-partitioned algorithms. An algorithmic blocking factor adjusts the granularity of the subtasks to maximize the efficiency of the hardware resources. In a distributed-memory environment, load balance is the essential motivation for distributing array entries over a collection of processes according to the block cyclic decomposition scheme. A distribution blocking factor is used to partition an array into blocks that are then mapped onto the processes. Optimal values of the algorithmic and distribution blocking factors often differ for a given algorithm and target architecture. Despite this fact, most of the parallel algorithms proposed in the literature assume the values of these blocking factors to be identical. This feature limits the flexibility and ease of use of such algorithms. When these blocking factors differ, methods are necessary to redistribute some data into the appropriate algorithmic form. This paper presents and discusses such algorithmic redistribution methods for the block cyclic decomposition scheme.

Algorithmic redistribution methods attempt to reorganize logically the computations and communications within an algorithmic context. In order to derive such methods, some properties of the block cyclic data distribution are first exhibited. Various algorithmic redistribution methods are then presented and applied to the representative outer product matrix-matrix multiply algorithm. Performance results are finally discussed and analyzed. The general block cyclic decomposition scheme is shown to allow for the expression of flexible and efficient algorithmically blocked basic linear algebra operations. Moreover, 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 high 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.

Key words. algorithmic blocking, redistribution, block cyclic decomposition

*This work was supported by the Defense Advanced Research Projects Agency under contract DAAH04-95-1-0077, administered by the Army Research Office.

[†]Department of Computer Science, University of Tennessee, Knoxville, TN 37996.

[‡]Department of Computer Science, University of Tennessee, Knoxville, TN 37996 and Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831.

1 Introduction

In a serial computational environment, *transportable efficiency* is the essential motivation for developing blocking strategies and block-partitioned algorithms [3, 6, 26, 41]. An algorithmic blocking factor adjusts the granularity of the subtasks to maximize the efficiency of the hardware resources. In a distributed-memory environment, load balance is the essential motivation for distributing array entries over a collection of processes according to the block cyclic decomposition scheme [1, 14, 37, 29, 17, 22, 30]. A distribution blocking factor is used to partition an array into blocks that are then mapped onto the processes. Optimal values of the algorithmic and distribution blocking factors often differ for a given algorithm and target architecture. Despite this fact, most of the parallel algorithms proposed in the literature assume the values of these blocking factors to be identical [21, 22, 51, 58]. This feature limits the flexibility and ease of use of such algorithms. The expression and implementation of these algorithms is simplified thanks to alignment restrictions on the operands that are naturally suggested by the unique blocking factor. Consequently, the scope of application of these algorithms is limited in a way that does not satisfy general purpose library requirements. High performance is however achievable on a wide range of distributed-memory concurrent computers, but usually depends on the chosen value of the distribution blocking factors. High performance can be maintained across platforms by parameterizing the user's data distribution or across library function calls by using general redistribution packages [42, 53]. When the algorithmic and distribution blocking factors differ, methods are necessary to redistribute some data into the appropriate algorithmic form. This paper presents and discusses such algorithmic redistribution methods for the block cyclic decomposition scheme.

Algorithmic redistribution methods attempt to reorganize logically the computations and communications within an algorithmic context. In order to derive such methods, some properties of the block cyclic data distribution are exhibited in Section 2. These properties are the basis of efficient algorithms for address generation, fast indexing techniques, and communication scheduling. Some of these algorithms are described in detail along with the properties from which they are deduced. Various algorithmic redistribution methods are then presented and applied to the representative outer product matrix-matrix multiply algorithm in Section 3. 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, little practical experiments have been so far reported in the literature. A scalability analysis of the presented algorithmic redistribution methods is finally given in Section 4. A number of experimental performance results are also presented and commented.

The general block cyclic decomposition scheme is shown to allow for the expression of flexible and efficient algorithmically blocked basic linear algebra operations. Moreover, 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 high 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.

2 Properties of the Block Cyclic Data Distribution

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 scheme is to maximize system performance by balancing the computational load among the 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 [18, 32, 34, 38] has aimed at studying different data decompositions [7, 13, 39]. As a result the two-dimensional block cyclic distribution [46] has been suggested as a possible general purpose basic decomposition for parallel dense linear algebra software libraries [23, 37, 49] because of its scalability [29], load balance and communication [37] properties.

The purpose of this section is to present important properties of the two-dimensional block cyclic data distribution. These properties are the basis of efficient algorithms for address generation, fast indexing techniques, and communication scheduling. Some of these algorithms are described in detail along with the properties from which they are deduced.

2.1 Analytical Definition of the Block Cyclic Data Distribution

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, where Γ denotes the set of all process coordinates (p, q) in this grid:

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

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:

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

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

Definition 2.1 The block cyclic distribution is 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

$$\left\{ \begin{array}{c} \Delta \longrightarrow \Gamma \\ (i,j) = ((l\ P + p)\ r + x, (m\ Q + q)\ s + y) \longmapsto (p,q) \end{array} \right. \tag{2.1}$$

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

$$\left\{ \begin{array}{c} \Delta \longrightarrow \Lambda \\ (i,j) = ((l\ P + p)\ r + x, (m\ Q + q)\ s + y) \longmapsto (l, m) \end{array} \right. \tag{2.2}$$

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

$$\left\{ \begin{array}{c} \Delta \longrightarrow \Theta \\ (i,j) = ((l\ P + p)\ r + x, (m\ Q + q)\ s + y) \longmapsto (x, y) \end{array} \right. \tag{2.3}$$

A few particular occurrences of the above definition are worth mentioning. First, **the blocked distribution** is determined by Definition 2.1 with $r = \lceil \frac{M}{P} \rceil$ and $s = \lceil \frac{N}{Q} \rceil$, i.e., $\Lambda = \{(0,0)\}$. Second, **the square block cyclic distribution** is a special case of Definition 2.1 with $r = s$. Finally, **the purely scattered or cyclic decomposition** is a particular instance of the square block cyclic distribution with $r = s = 1$, i.e., $\Theta = \{(0,0)\}$. The expression of the properties presented in the following section can often be simplified for those specific cases, however, these properties will be stated for the above general definition of the block cyclic distribution.

2.2 Properties of the Block Cyclic Data Distribution and LCM Tables

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 this data distribution scheme in order to derive algorithmic redistributed operations.

The k-diagonal of a matrix is defined to be the set of entries a_{ij} such that $i - j = k$. With this definition the 0-diagonal is the "main" diagonal of a matrix. The first sub-diagonal and super-diagonal are respectively the 1-diagonal and the -1-diagonal.

Definition 2.2 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

$$\left\{ \begin{array}{lll} LCMT^{p,q}_{0,0} & = q\ s - p\ r + k, & \\ \forall l \in I\!N, & LCMT^{p,q}_{l,*} = LCMT^{p,q}_{l-1,*} - P\ r, & \\ \forall m \in I\!N, & LCMT^{p,q}_{*,m} = LCMT^{p,q}_{*,m-1} + Q\ s. & \end{array} \right.$$

An equivalent direct definition is

$$\forall (l, m) \in I\!N^2 \quad LCMT^{p,q}_{l,m} = (m\ Q + q)\ s - (l\ P + p)\ r + k.$$

The above definition of an LCM table could be generalized in order to encompass the entire family of Cartesian mappings [10]. Indeed, 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 document. In particular, Definition 2.2 can easily be adapted to a block cyclic distribution with a partial first block [52]. In other words, the first block of rows (respectively columns) is of size ir (respectively is) instead of r (respectively s). Such a generalization is convenient to allow for the specification of sub-matrix operands which upper left corner is not aligned on block boundaries [45].

The equation for the k-diagonal is given by

$$LCMT_{l,m}^{p,q} = x - y, \tag{2.4}$$

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

$$1 - s \leq LCMT_{l,m}^{p,q} \leq r - 1. \tag{2.5}$$

In addition the value of $LCMT_{l,m}^{p,q}$ specifies where the diagonal starts within a block owning diagonals as illustrated in Figure 1.

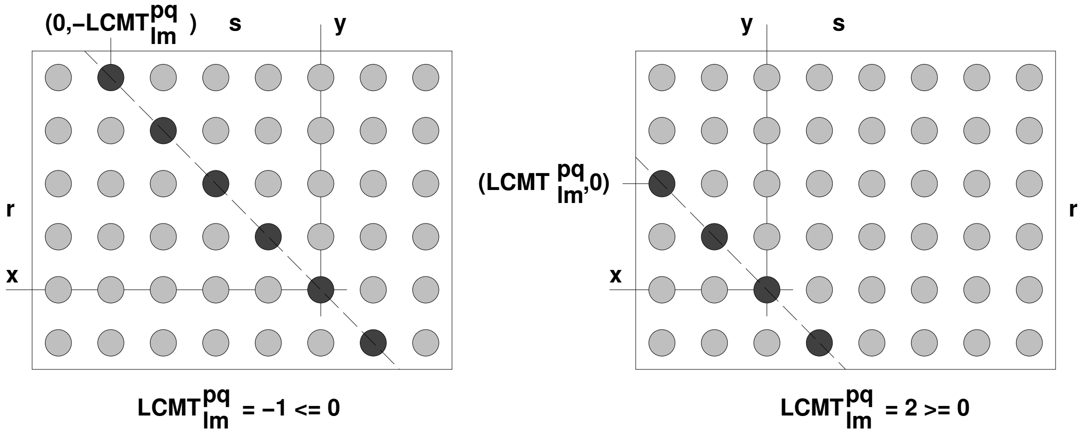

Figure 1: Meaning of different values of $LCMT_{l,m}^{p,q}$ with $r = 6$, $s = 8$

It follows from Definition 2.2 that the local blocks in process (p, q) such that $LCMT_{l,m}^{p,q} \leq 0$ (respectively $LCMT_{l,m}^{p,q} \geq 0$) own matrix entries a_{ij} that are globally below (respectively above) the k-diagonal. Similarly, the local blocks in process (p, q) such that $LCMT_{l,m}^{p,q} \leq -s$ (respectively $LCMT_{l,m}^{p,q} \geq r$) correspond globally to strictly lower (respectively upper) blocks of the matrix. Moreover, 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. Similarly, 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.

Let $L = \operatorname{lcm}(P\ r, Q\ s)$ and $g = \gcd(P\ r, Q\ s)$ be respectively the least common multiple and greatest common divisor of the quantities Pr and Qs. The index pairs (i, j), $(i + L, j)$, $(i, j + L)$ and $(i + L, j + L)$ refer to array entries that are residing in the same process (p, q). Indeed, L is a multiple of Pr and Qs. In other words, the distribution pattern repeats after each square block of size L. This square matrix is called an *LCM block*. Each process owns exactly $L/P \times L/Q$ entries of this LCM block. This larger partitioning unit has been originally introduced in the restricted context of square block cyclic mappings in [19, 20, 21]. The meaningful part of the LCM tables to be considered in each process is of size $L/(Pr) \times L/(Qs)$.

Figure 2 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 direct implications of Definition 2.2 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 L-periodicity of the distribution mapping, 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 in each process to locate the k-diagonals. This information is cheap to compute and one can afford to recompute it when needed.

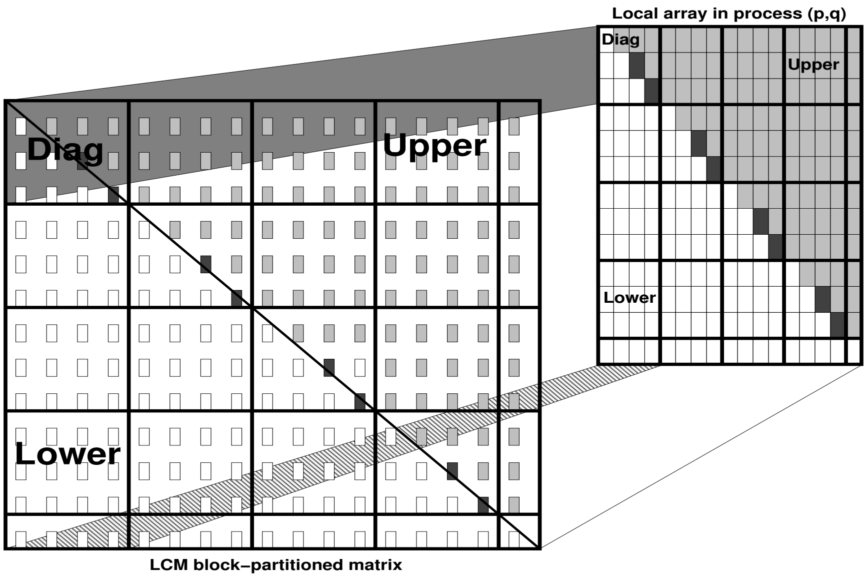

Figure 2: LCM template ($P = 2$, $Q = 3$, $r = 4$, $s = 2$ and $(p, q) = (1, 1)$).

Figure 3 shows a 1-LCM block for a given set of distribution parameters P, r, Q and s as well as the associated 1-LCM tables. 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 process $(0, 0)$. The value of the LCM table entry $(0, 0)$ is 1. Since this value is greater than $-s = -3$ and less than $r = 2$, it follows that this block $(0, 0)$ owns diagonals. Moreover, locally within this block the diagonal starts in position $(LCMT_{00}^{00}, 0) = (1, 0)$. The periodicity in this table is shown by the block of coordinates $(3, 2)$ which is such that $LCMT_{00}^{00} = LCMT_{32}^{00} = 1$. 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 if and only if $LCMT_{lm}^{00} \leq -s$ (respectively $LCMT_{lm}^{00} \geq r$). These same remarks apply to all of the other LCM tables shown in Figure 3.

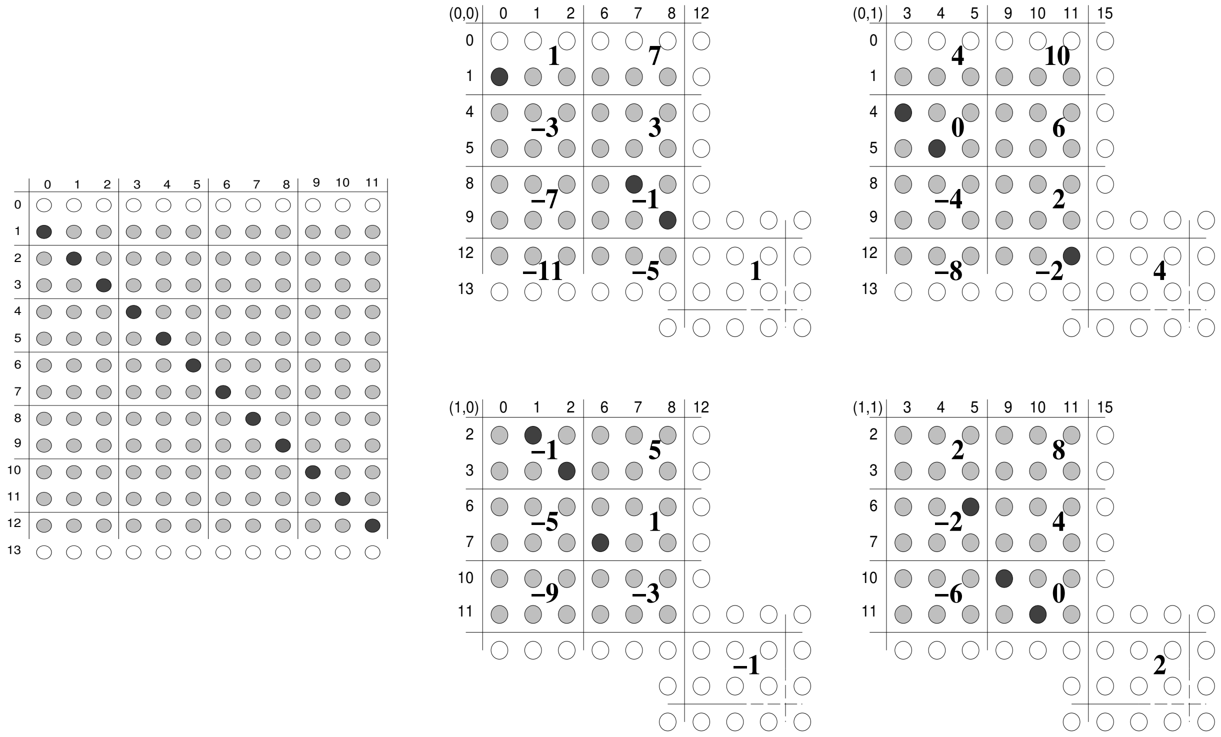

Figure 3: 1-LCM block and 1-LCM tables for $P = 2$, $r = 2$, $Q = 2$ and $s = 3$.

Property 2.1 The number of $r \times s$ blocks owning k-diagonal entries is given by

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

Proof. (sketch) First note that one can assume without loss of generality that $-s < k < r$ 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 $(r-1, s-1)$ 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 $L/\mathrm{lcm}(r,s)$ is indeed an integer, one may observe that this quantity can be rewritten as $((u\ Q)\ P\ r + (t\ P)\ Q\ s)/g$ with u and t in $\mathbb{Z}$. Finally, there are exactly $L/\mathrm{lcm}(r,s)$ such blocks in an LCM block. ∎

Property 2.2 A necessary and sufficient condition for every process to own k-diagonal entries is given by $r + s - \gcd(r,s) \geq g$ if $\gcd(r,s)$ divides k and $r + s \geq g$ otherwise.

Proof. The condition is *sufficient*: remark that $\gcd(r,s)$ divides g. If $\gcd(r,s)$ divides k (note that this will always be the case if $\gcd(r,s) = 1$), then $\dfrac{r+s}{\gcd(r,s)} - 1$ is the number of multiples of

$\gcd(r, s)$ in the interval $I_{p,q} = (p\,r - (q-1)\,s \ldots (p+1)\,r - q\,s)$. The number of multiples of g in the interval $I_{p,q}$ is $\frac{g}{\gcd(r,s)}$. Thus, the inequality $\frac{r+s}{\gcd(r,s)} - 1 \geq \frac{g}{\gcd(r,s)}$ is a sufficient condition for a multiple of g to be in this interval $I_{p,q}$. Otherwise, when $\gcd(r, s)$ does not divide k, Equation 2.5 can be rewritten as

$$p\ r - (q+1)\ s < m\ Q\ s - l\ P\ r + k < (p+1)\ r - q\ s. \tag{2.6}$$

For any given process of coordinates (p, q), there must exist a $t \in \mathbb{Z}$ such that $m\ Q\ s - l\ P\ r = t\ g$ verifying Inequality 2.6. Moreover, the interval of interest $I_{p,q}$ is of length $r + s - 1$. A sufficient condition for all processes to have k-diagonals is given by $r + s - 1 \geq g$. Since $\gcd(r, s) \neq 1$ and $\gcd(r, s)$ divides g, this sufficient condition can be equivalently written as $r + s \geq g$.

The condition is *necessary*: suppose there exists a process (p, q) having two distinct blocks owning k-diagonals. Then, $r + s - \gcd(r, s) \geq g$ if $\gcd(r, s)$ divides k, and $r + s \geq g$ otherwise. There are two multiples of g in some interval $I_{p,q}$. 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 2.1. ∎

Property 2.3 The number of processes owning k-diagonal entries is given by

$$\begin{cases} \max(PQ\,(\frac{r+s-\gcd(r,s)}{g}), P\ Q) & \text{if } \gcd(r,s) \text{ divides } k, \\ \max(PQ\,(\frac{r+s}{g}), P\ Q) & \text{otherwise.} \end{cases}$$

PROOF. The result follows from the fact that $Lg = (P\ r)(Q\ s)$ and Properties 2.1 and 2.2. ∎

These last properties are summarized in Table 1. The end of this section aims at determining the probability that the quantities $r + s - \gcd(r, s)$ or $r + s$ are greater or equal to g, that is, the probability that every process owns k-diagonals entries. The result 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 $\gcd(r, s)$ and $g = \gcd(P\,r, Q\,s)$.

Table 1: Properties of the k-diagonal for the block cyclic distribution

Blocks owning k-diagonals	$-s < m\ Q\ s - l\ P\ r + q\ s - p\ r + k < r$
Processes owning k-diagonals	$\exists t \in \mathbb{Z}$, such that $p\ r - (q+1)\ s < t\ g + k < (p+1)\ r - q\ s$
Number of such processes	$\min((PQ\,(r+s))/g, PQ)$ if $\gcd(r,s)$ divides k, $\min((PQ\,(r+s-\gcd(r,s)))/g, PQ)$ otherwise.

It is difficult to compute analytically the probability that all processes will own k-diagonal entries. However, it is likely that this probability, if it exists, converges rapidly [52]. It is possible to rely on a computer to enumerate all 4-tuples in a finite and practical range such that the quantities $r+s-\gcd(r,s)$ or $r+s$ are greater or equal to g. The results are presented in Figure 4. It is important 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 $[1..n]^4$. Figure 4 does not prove the existence of the limit and therefore of the probability. However, if it exists, its value is very close to one. In other words, if one picks *random* distribution parameters, it is very likely that all processes in the grid will own k-diagonals. Figure 4 not surprisingly shows that the ratios of distribution parameters such that k-diagonals are evenly distributed tends

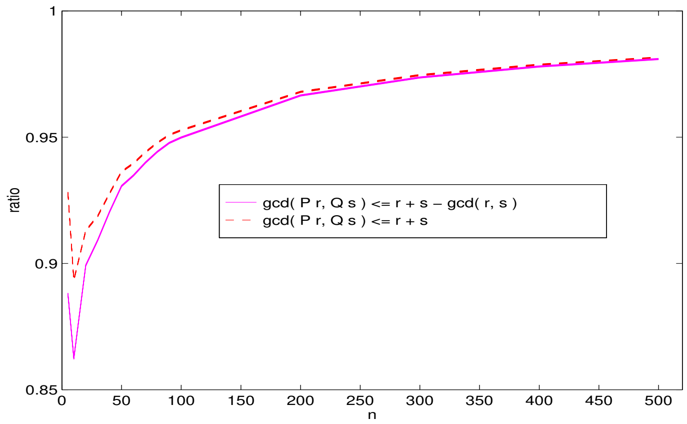

Figure 4: Ratios of tuples (P, r, Q, s) in $[1..n]^4$ such that $r+s-\gcd(r,s) \geq g$ or $r+s \geq g$.

towards one. More interesting is the fact that this function R 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. Properties 2.2 and 2.3 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. In other words, 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. To reduce this complexity, it is necessary to choose small values of the distribution parameters r and s. Furthermore, 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 sections will confirm the validity of this assumption. It follows that small distribution blocking factors are favorable to 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 distribution blocking factors, as far as the performance of redistribution operations is concerned. Transportable efficiency for redistribution operations requires thus the support of the parameterized family of block cyclic mappings.

The algorithmic redistributed operations described can be expressed in terms of locating diagonals of a distributed matrix. The next section 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 above.

3 Algorithmic Redistribution Methods

This section presents different kinds of blocking strategies for distributed-memory hierarchies. Most of them can be formulated in terms of "LCM-operations", i.e., operations relying on LCM tables for their derivation, expression and implementation. 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, little practical experiments have been so far reported in the literature.

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

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

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 parameters. 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. Ideally, this choice has been influenced by its performance implications on the physical blocking strategy. Most of the parallel algorithms presented in the literature [2, 7, 10, 11, 22, 23, 30, 33, 34, 35, 39, 47, 48, 59] rely on this strategy. The algorithm performing the rank-K update operation using a physical blocking strategy is relatively easy to express. Strong alignment and distribution assumptions are made on the matrix operands. In particular, the distribution blocking factors used to decompose the columns of A and the rows of B must be equal. Moreover, the rows of A (respectively the columns of B) must be aligned to the rows (respectively columns) of C. The pseudo code for this algorithm is given below.

$$\left\{\begin{array}{l}
\text{for } kk = 1,\, K,\, NB_{dis} \\
\quad kb = \min(K - kk + 1,\, NB_{dis}) \\
\quad \text{Broadcast } A(:, kk : kk + kb - 1) \text{ within process rows;} \\
\quad \text{Broadcast } B(kk : kk + kb - 1, :) \text{ within process columns;} \\
\quad C \leftarrow C + A(:, kk : kk + kb - 1) * B(kk : kk + kb - 1, :); \\
\text{end for}
\end{array}\right.$$

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 distribution blocking factor NB_{dis}. The smaller this value is, the larger the number of messages and the lower the possible data reuse during each computational phase. In other words, the performance degrades as the value of the distribution blocking factor is decreased. If the value of this factor is very large, the communication computation overlap decreases causing a performance degradation. High performance and efficiency can still be achieved for a wide range of blocking factors. This has been reported in [2, 29, 51, 58, 16].

Three alternatives to the physical blocking strategy are first presented in this section. Then, a few other related applications of those methods are outlined. 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.

3.1 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 an algorithmic blocking factor. Its optimal value depends on the target machine characteristics. If this logical value is equal to the distribution blocking factors, then aggregation and physical blocking are the same. Otherwise, a few rows or columns which are globally contiguous and residing in distinct processes, are gathered into a single process row or column and this panel becomes the matrix operand. This strategy is particularly efficient when the distribution 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 an algorithmic blocking factor NB_{alg} 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 eigenvalue problem in [13, 14, 8, 36, 54, 55] for the purely scattered distribution. 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.

$$\left\{\begin{array}{l}
\text{for } kk = 1,\, K,\, NB_{alg} \\
\quad kb = \min(K - kk + 1,\, NB_{alg}); \\
\quad \text{Aggregate } A(:, kk : kk + kb - 1) \text{ in one process column;} \\
\quad \text{Broadcast } A(:, kk : kk + kb - 1) \text{ within process rows;} \\
\quad \text{Aggregate } B(kk : kk + kb - 1, :) \text{ in one process row;} \\
\quad \text{Broadcast } B(kk : kk + kb - 1, :) \text{ within process columns;} \\
\quad C \leftarrow C + A(:, kk : kk + kb - 1) * B(kk : kk + kb - 1, :); \\
\text{end for}
\end{array}\right.$$

The aggregation and disaggregation techniques attempt 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 this 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 [40].

3.2 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 an algorithmic blocking factor. Its optimal value depends on the target machine characteristics. However, rows or columns that may not be locally contiguous are packed, but according to an external criterion, typically the distribution parameters of another operand.

Consider the rank-K update operation illustrated in Figure 5. 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 5, 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 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 5. 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 can be regarded as a generalization of the physically blocked version. It presents,

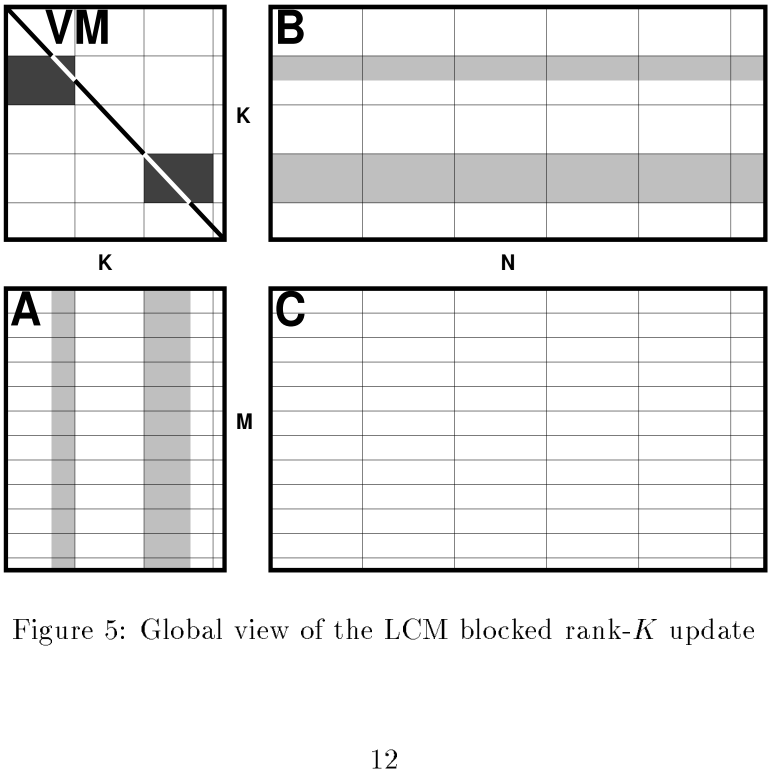

Figure 5: Global view of the LCM blocked rank-K update

however, some advantages. First, 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 by having the process column emitting the broadcasts remaining fixed as long as possible. Furthermore, 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 at each step. The pseudo code for the LCM blocking strategy is given below.

```
for q = 0, Q - 1
  for p = 0, P - 1
    npq = number of diagonals process (p, q) owns;
    Process column q packs and broadcasts those npq columns of A within process rows;
    Process row p packs and broadcasts those npq rows of B within process columns;
    Perform local matrix - matrix multiply;
  end for
end for
```

This approach presents the advantage that the cost of the gathering 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, the LCM blocking strategy is well-suited for multiplying two matrices, where each contribution to the resulting matrix entries can be added in any order. 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.

3.3 Aggregated LCM Blocking

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. This more elaborate algorithmic blocking method maintains the local computational granularity even if the number of diagonals residing in a process is or becomes too small. The algorithm goes as follows. A process of coordinates (p, q) is considered and the number of diagonals npq that process owns are handled by chunks of size NB_{alg}. If npq is a multiple of NB_{alg}, then the algorithm proceeds to the next process in the grid, either $(p+1, q)$ or $(0, q+1)$. Otherwise, if $(p+1, q)$ is the next process, the remaining rows of B in process (p, q) are sent to the process $(p+1, q)$, and the LCM blocking method proceeds to this process taking into account the remainder of the previous step. If $(0, q+1)$ is the next process, this last procedure is applied to the remaining rows of B and columns of A. This algorithm therefore maintains the local computational granularity at a low communication overhead.

3.4 Redistribution and Static Blocking

The above framework can be used to tackle the run-time array redistribution problem when those arrays are distributed in a block cyclic fashion over a multidimensional process grid. Solving this redistribution problem requires first to generate the messages to be exchanged, and second to schedule these messages so that communication overhead is minimized. A comparative survey of the available literature can be found in [61]. Most of the attention has been so far paid to the message generation phase [15, 44, 44, 5, 57] and only a few papers deal with the communication scheduling phase [43, 53, 60, 50]. It turns out that the properties of the block cyclic distribution presented in this paper can be used to study further this scheduling problem as it is shown in [27]. Moreover, the message generation phase can also be addressed with the help of the LCM tables. Figure 6 illustrates this fact in the one-dimensional case. X (respectively Y) is a $M \times N$ one-dimensional array distributed over P (respectively Q) processes with a distribution blocking factor of r (respectively s). The figure shows the global and local view of the redistribution mapping as well as the $M \times M$ distributed matrix induced by X and Y. Locating the diagonals of this matrix using LCM tables naturally provides a possible message generation algorithm. This figure also shows that the general complexity of the redistribution problem is related to the number of processes in the $P \times Q$ grid owning diagonals. Furthermore, the transpose and shift operations can be handled similarly within this framework. Finally, this approach can be generalized to handle the multidimensional case and the problem of accessing array entries with a non-unit stride [52]. It follows that efficient algorithms for the re-alignment of operands block cyclically distributed can also be expressed using the LCM tables and the above properties. In other words, flexible and efficient basic linear algebra kernels for distributed-memory concurrent computers can be expressed within the same framework.

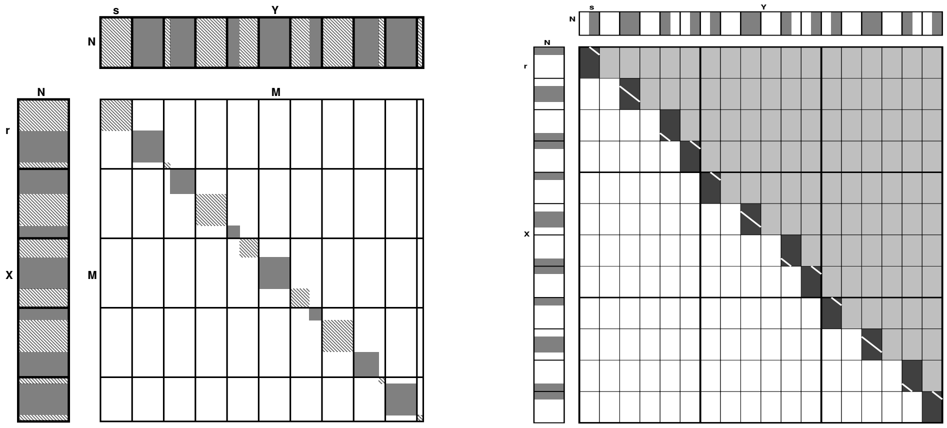

Figure 6: Global and local view of one-dimensional redistribution

LCM tables can be also used to derive another algorithmic blocking method, called the *static blocking* strategy thereafter, which 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 symmetric rank-K update operation $C \leftarrow C + A\,A^T$ is easy to describe. C is an $N \times N$ symmetric matrix for which only the upper or lower triangle should be referenced, and A is a matrix of dimension $N \times K$. The matrix A has been replicated in every process column and the matrix A^T replicated in every process row. The distributed matrix C is partitioned into diagonal and strictly upper or lower LCM blocks as shown in Figure 7. 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 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.

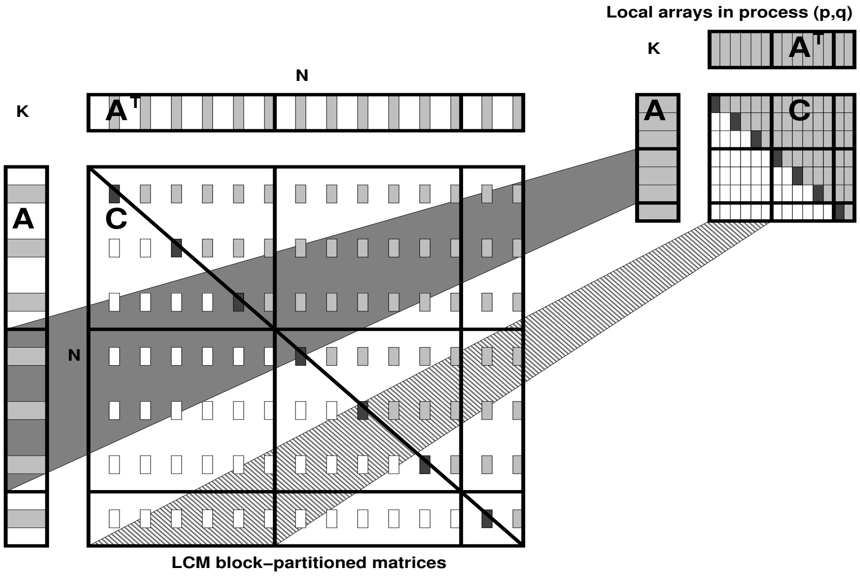

Figure 7: Static symmetric rank-K update

Figure 7 shows that the local update can be expressed in terms of symmetric rank-K updates and matrix-matrix multiplies. The LCM tables provide the necessary information to organize the local computation in such a way that one can take advantage of the high efficiency of the matrix-matrix multiply kernel [9, 62]. A similar approach has been proven highly efficient for *GEMM-based* BLAS implementations such as [26, 41]. 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 logically rearrange the computations. This suggests that an efficient implementation of the symmetric rank-K update when $N \gg K$ would use the LCM blocking strategy to replicate A over C, and the static blocking technique to perform the local update. The algorithmic blocking factors for both phases can be chosen independently.

3.5 Rationale

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 [6] can be reused. The ScaLAPACK [12] 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 nevertheless 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.

This section 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 equivalent to the usual techniques for the restricted cases. The presentation of these general strategies stressed their systematic derivation from the properties of the underlying mapping. The importance of the LCM tables introduced in Section 2.2 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.

4 Performance Analysis and Experimental Results

A theoretical model of a distributed-memory computer is presented early in this section. It is an abstraction of physical models that 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 is then applied to the algorithmic blocking strategies presented in Section 3 in order to analyze their scalability. Finally, a number of experimental results are presented and commented.

4.1 The Machine Model

Distributed-memory computers 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 [46]. 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 dynamically connected to one another by the switching elements to establish at run time the paths between processors' memories. The interconnection network of the machine model considered here is a static two-dimensional $P \times Q$ rectangular mesh with wraparound connections. 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 [46].

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 $\beta\, L$ 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 worm-hole routing). This approximation is reasonable for most current distributed-memory concurrent computers. 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 paper 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 a 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 [46] and the references therein.

4.2 Scalability Analysis

The rank-K update 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

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

In the following we assume for simplicity that $M = N = K$. The number of floating point operations is assumed to be equal to $2\,N^3$. All three matrices are distributed onto the same square process grid. Moreover, we also assume that the matrix operands are distributed according to the square block cyclic data distribution (see Definition 2.1), and that the distribution blocking factors are the same for all operands. Therefore, no re-alignment phase is necessary to be performed. This distribution blocking factor is denoted by NB_{dis} in the following.

Four algorithms are considered, denoted by PHY, AGG, LCM and RED. PHY denotes the physically blocked variant, AGG identifies the aggregation algorithm and the LCM blocking algorithm is denoted by LCM. Finally, a fourth variant RED is considered where the matrices A and B are completely redistributed before hand. For the algorithmic blocking variants AGG and LCM, the algorithmic blocking factor is denoted by NB_{alg}. Parallel efficiency, $E(n,p)$, for a problem of size n on p processors is defined in the usual way [32] by

$$E(n,p) = \frac{1}{p}\frac{T_{\mathrm{seq}}(n)}{T(n,p)} \tag{4.7}$$

where $T(n,p)$ is the runtime of the parallel algorithm, and $T_{\mathrm{seq}}(n)$ 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). The parallel runtime and efficiency on our machine model for the four algorithms PHY, AGG, LCM and RED are computed below as a function of the local computational speed γ, the communication parameters α and β_d (the time for a floating point number to traverse the network), and the total number of processors p.

The key-factor of this performance analysis is to model the cost of a sequence of broadcasts on a ring [2, 58] where the source either remains the same or is incremented by one after each broadcast. In the physical blocking strategy, the source process of the broadcast sequence is incremented at each step. The parallel runtime of the physically blocked variant algorithm is given by

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

A similar analysis for the physical blocking variant can also be found in [2, 58]. The aggregation blocking strategy 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 ($k \geq 2$). In practice, the blocks are only approximately of the same size. k is clearly bounded above by $\sqrt{p}$. In addition, the algorithmic blocking factor NB_{alg} is used to partition the communication and computation. It follows that the estimated execution time on our machine model for the aggregation strategy is given by

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

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_{alg}. With these assumptions, the estimated execution time of the LCM blocking strategy is given by

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

Finally, the parallel run time of the RED variant is obtained by adding to the quantity $T_{PHY}(N,p)$ computed above the approximated time to redistribute two square matrices of order N, that is $2\,(\,p\,\alpha + \frac{N^2\,\beta_d}{p}\,)$.

Table 2 summarizes the estimated parallel efficiency for each variant studied in this Section. 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 when a "good" value of the distribution blocking factor NB_{dis} is selected. 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 physical block size NB_{dis} can be used to lower the importance of the latency α. The aggregation algorithm is also

Table 2: Estimated parallel efficiencies for various blocking variants

$E_{PHY}(N,p)$	$(1+\frac{2}{\gamma}(\frac{p\,\alpha}{NB_{dis}N^2}+\frac{\sqrt{p}\,\beta_d}{N}))^{-1}$
$E_{AGG}(N,p)$	$(1+\frac{k}{\gamma}(\frac{p\,\alpha}{NB_{alg}N^2}+\frac{\sqrt{p}\,\beta_d}{N}))^{-1}$ with $k \approx \lceil\frac{NB_{alg}}{NB_{dis}}\rceil$
$E_{LCM}(N,p)$	$(1+\frac{3}{2\,\gamma}(\frac{p\,\alpha}{NB_{alg}N^2}+\frac{\sqrt{p}\,\beta_d}{N}))^{-1}$
$E_{RED}(N,p)$	$(1+\frac{1}{\gamma}((2+\frac{p\,NB_{dis}}{N})\frac{p\,\alpha}{NB_{dis}N^2}+\frac{(2\,\sqrt{p}+1)\,\beta_d}{N}))^{-1}$

scalable. The value of k is a constant that only depends on the ratio between the algorithmic NB_{alg} and distribution NB_{dis} blocking factors. This 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 distribution blocking factor is larger than the algorithmic 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. The LCM blocking variant is also scalable for aligned matrix operands. This variant is slightly more efficient than the physical and aggregation strategies. It should be noted however that our machine model assumes that the local data copy operation is free. In reality, such an assumption depends on the target machine and may affect the results presented in Table 2. The RED algorithm, however, is not scalable because of the latency term.

4.3 Experimental Performance Results

The purpose of this section is to illustrate the general behavior of algorithmically redistributed operations as opposed to presenting a collection of particular performance numbers. The presentation style aims at facilitating the comparison of the different blocking strategies for a set of illustrative and particular cases. Experimental performance results are presented below for two distinct distributed-memory concurrent computers, namely the Intel XP/S Paragon [25] and the IBM Scalable POWERparallel System [4, 24, 56]. All of our experiments were performed in double precision arithmetic. The local rank update operation was performed by calling the appropriate subprogram of the vendor-supplied BLAS. The communication operations were implemented by explicit calls to the Basic Linear Algebra Communications Subprograms (BLACS). The BLACS [28, 31] 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 sub-matrix from one process to another, to broadcast sub-matrices, 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. The performance of our algorithms is measured in Mflops/s. This is appropriate for large dense linear algebra computations since floating point dominates communication.

The matrix operands used for our experiments were distributed in such a way that no re-alignment phase was necessary as explained in Section 4.2. Experimental performance results for non-aligned operands have been reported in [52]. Different values of the distribution blocking factor have been used. A machine dependent value of the algorithmic blocking factor NB_{alg} used by the AGG and LCM variants has first been determined for each platform and used for all of the experiments. On our Intel XP/S Paragon, we found that a reasonable value for this algorithmic blocking factor was 14. On the IBM SP, the value of 70 has been selected. The first experiment denoted XP_A0 for the Intel XP/S paragon and SP_A0 for the IBM SP uses the value of NB_{alg} as the distribution blocking factor NB_{dis} 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. Figure 8 shows the performance of the physical blocking (PHY), aggregation (AGG) and the LCM blocking (LCM) strategies using the value of NB_{alg} as the distribution blocking factors for the three matrix operands. According to the conclusions of the previous section, the performance of the three variants is almost identical on each platform with a slight advantage

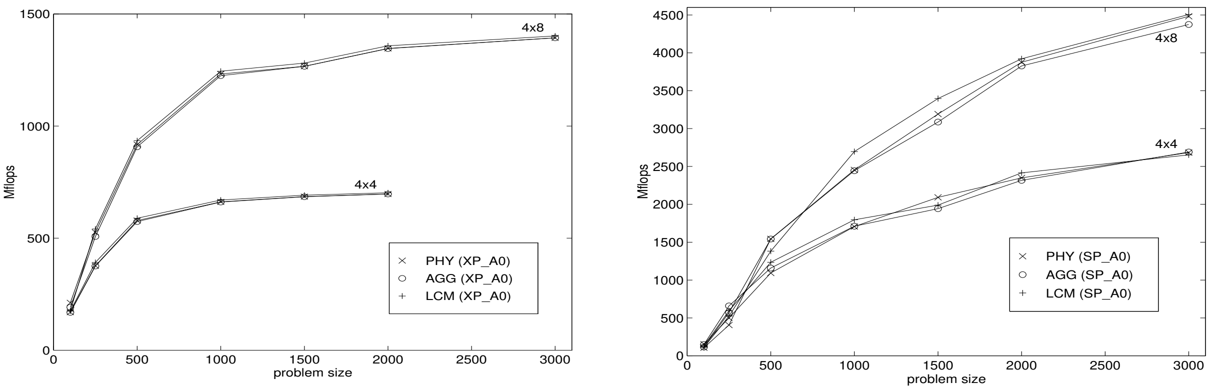

Figure 8: Performance in Mflops/s of algorithmic blocking variants for a "good" physical data layout case and various process grids on the Intel XP/S Paragon and the IBM SP

to the LCM blocking variant. In the rest of this section, the performance curves shown in Figure 8 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 distribution blocking factors.

Table 3: Specification of the experiments

Experiment #	Distribution parameters	
XP_A0, SP_A0	$NB_{dis} = NB_{alg}$	for all operands.
XP_A1, SP_A1	$NB_{dis} = 1$	for all operands.
XP_A10	$NB_{dis} = 10$	for all operands.
SP_A20	$NB_{dis} = 20$	for all operands.
XP_A40	$NB_{dis} = 40$	for all operands.
XP_A100	$NB_{dis} = 100$	for all operands.
SP_A200	$NB_{dis} = 200$	for all operands.

A few other experiments have been specified as follows. Each experiment has been given an encoded name of the form XX_A#. 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. # is a number or a string distinguishing each experiment. For each experiment, the distribution parameters of the matrix operands A, B and C are the same. Table 3 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 for all of these values. Results are reported on a 4×4 Intel XP/S Paragon and a 4×8 IBM SP.

Figure 9 shows the performance results obtained by the physical blocking strategy on aligned data. The physical blocking variant uses the distribution blocking factors as the computational

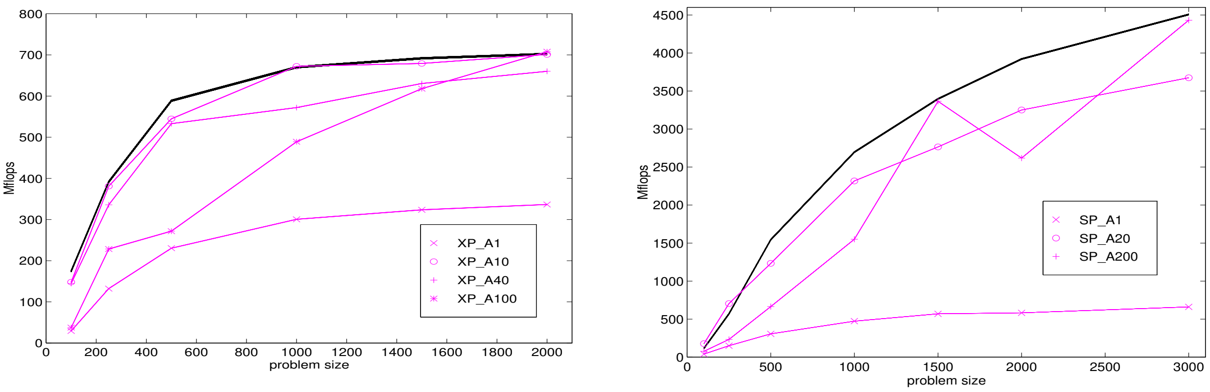

Figure 9: Performance of the PHY variant on a 4×4 Intel XP/S Paragon and on a 4×8 IBM SP

unit. When the distribution parameters are very small, the performance is dramatically degraded, because of the local performance of the rank-k update for small values of k. 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 on such computers. Very large distribution parameters increase the computation load imbalance, which is characterized by highly irregular performance results. For the experiment SP_A200, for $N = 1500$, each processor has almost the same amount of data. However, for $N = 2000$, the most loaded processes have locally a 600×400 matrix on which to operate. The matrices residing in the 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 Figure 9 are typical of this phenomenon.

Figure 10 shows the performance results obtained for the aggregation strategy on aligned data. The dependence of the performance from the physical distribution parameters is largely decreased. The performance results are pushed towards the result of reference. For very small values of the distribution parameters, one expects a large performance improvement compared to the physical blocking strategy. This aspect is particularly evident for both target platforms as shown in Figure 10. 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 performance 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.

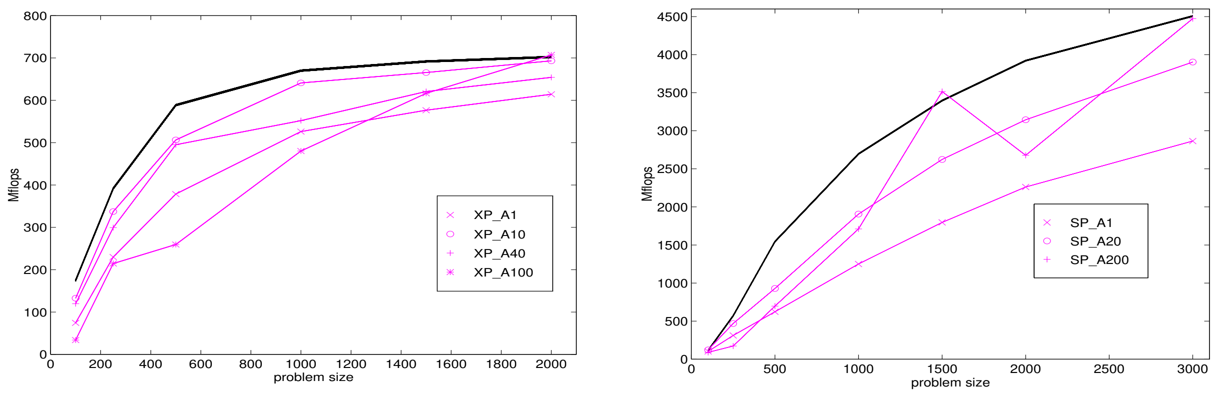

Figure 10: Performance of the AGG variant on a 4×4 Intel XP/S Paragon and on a 4×8 IBM SP

Figure 11 shows 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 decouples the performance results from a poor choice of the distribution 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 distribution blocking factors is now superior to the performance observed for distribution blocking factors slightly larger than NB_{alg} (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_{alg} 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. These results 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 above.

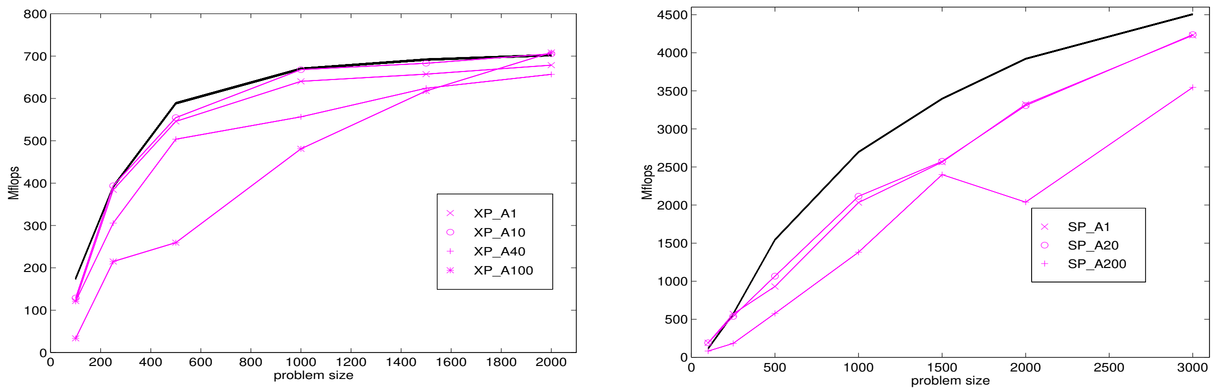

Figure 11: Performance of the LCM variant on a 4×4 Intel XP/S Paragon and on a 4×8 IBM SP

Figure 12 shows the performance results when the matrix operands A and B are aligned but redistributed (RED) for efficiency reasons. To perform the complete redistribution of a two-dimensional block cyclically distributed matrix, the appropriate component of the ScaLAPACK [12] software library [53] has been used. Even if these plots show the performance obtained for the same experiments as above, 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 12 illustrates the dramatic performance consequences

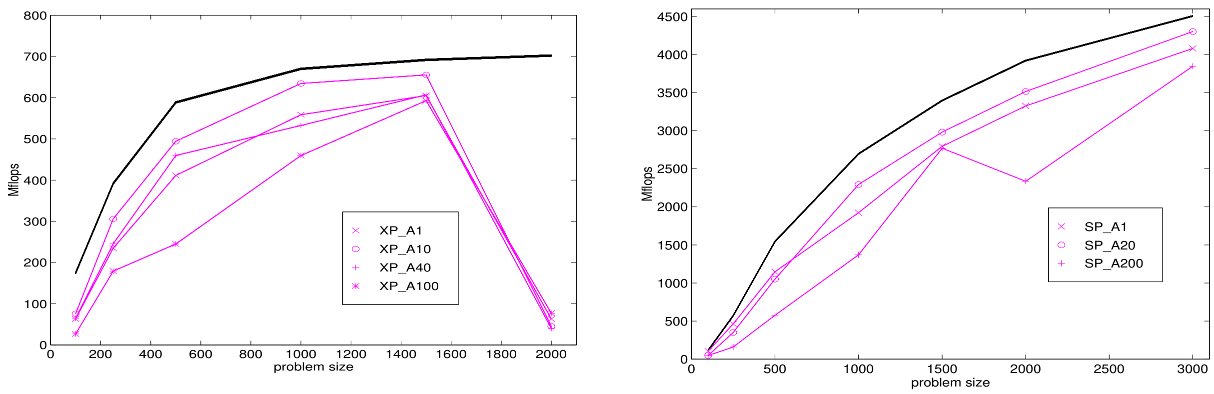

Figure 12: Performance of the RED variant on a 4×4 Intel XP/S Paragon and on a 4×8 IBM SP

of using virtual memory on 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 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 Megabytes of physical memory. It was not feasible to estimate the impact of the use of virtual memory in a reasonable amount of time.

The results presented in this section 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 independently from the distribution parameters. Moreover, 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 3.1. 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.

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 to be redistributed. This prevents from redistributing the largest operands. This argumentation was at the beginning of our motivation for developing algorithmically redistributed operations that require a much smaller amount of memory.

5 Conclusions

Most of the parallel algorithms for basic linear algebra operations proposed in the literature thus far focus on the naturally aligned cases and rely on the physical blocking strategy to efficiently use a distributed memory hierarchy. This restricted interest prevents one from providing the necessary flexibility that a parallel software library requires to be truly usable. These restrictions 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.

A number of properties of the block cyclic distribution were formally exhibited. The relationship between the distribution parameters and the complexity of the array redistribution was determined. The intuitive result that the complexity of these operations increases with the perimeter of the distribution partitioning unit was proved for a finite range of possible and realistic values of the distribution parameters. Moreover, these properties form the theoretical basis for a characterization of the block cyclic decomposition. They naturally suggested an elegant and convenient data structure that encapsulates and reveals the essential features of the LCM block partitioning unit. LCM tables were thus introduced and shown to be a convenient tool for the derivation of alternatives techniques to the physical blocking strategy. The originality of the algorithmic redistribution methods then presented resides in their systematic derivation from these properties of the underlying mapping. Such a feature is particularly attractive from the software library design point of view. Furthermore, this approach can be generalized to the more general family of Cartesian mappings.

The performance results presented in this paper show that when the matrix operands are aligned, the algorithmically redistributed operations based on the aggregation and the LCM blocking strategies are competitive in terms of performance with the beforehand complete redistribution variant (RED). For a variety of distribution and machine parameters one can thus afford to redistribute the matrix operands "on the fly" without a significant performance degradation. This conclusion must be refined when the matrix operands have to be redistributed before the aligned operation can take place [52]. Nevertheless, 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 despite the high memory cost. This problem can be tackle in two ways. First, it is conceivable to redistribute the operands 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 estimate at run-time the amount of usable memory on each process. Second, redistribution in place is also possible assuming a large enough amount of memory has been initially allocated.

Algorithmic redistribution methods can alleviate natural alignment restrictions at a low, sometimes negligible, performance cost for basic operations and various block cyclic distributions. In addition, these techniques 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 paper. Indeed, it says that the algorithms presented in this document allows to produce a general purpose and flexible parallel software library of basic linear algebra subprograms. These algorithms have been shown 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, but merely a characteristic of the algorithms that have been so far proposed to deal with a distributed memory hierarchy.

Software Availability

A complete set of parallel basic linear algebra subprograms (PBLAS) for distributed-memory computers heavily relying on the algorithmic redistribution methods presented in this paper, namely the aggregation and aggregated LCM blocking strategies, is available at the following address `http://www.netlib.org/scalapack/prototype`. This version (V2.0α) of the software is upward compatible with the version 1.5 currently used by the ScaLAPACK library [12]. In this prototype version, all the alignment restrictions have been removed. Data re-alignment is performed on the fly and only when necessary. All operands should be distributed according to the general block cyclic scheme as before, or to the general block cyclic scheme with a partial first block (see Section 2.2). In addition, operands can be replicated in process rows, columns or both. The algorithmic blocking techniques described in this paper are used throughout the software. Testing and timing programs have been upgraded and are also provided to test the above new functionalities. Preliminary performance results are highly satisfactory. Nevertheless, fine performance tuning, profiling and precise timing analysis of each component for various distributed-memory concurrent computers are ongoing tasks. Finally, a proper documentation as well as a precise software design description will be made available with the final release of PBLAS V2.0 in 1998.

Acknowledgments

The authors acknowledge 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.

References

[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] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear Algebra Framework for Static HPF Code Distribution. Technical Report A-278-CRI, CRI-Ecole des Mines, Fontainebleau, France, 1995. (Available at http://www.cri.ensmp.fr).

[6] 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.

[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] P. Bangalore. The Data-Distribution-Independent Approach to Scalable Parallel Libraries. Master's thesis, Mississippi State University, 1995.

[9] J. Bilmes, K. Asanovic, J. Demmel, D. Lam, and C. Chin. Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology. Technical Report UT CS-96-326, LAPACK Working Note No.111, University of Tennessee, 1996.

[10] 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.

[11] 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.

[12] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, 1997.

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

[14] 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.

[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. A New Parallel Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers. Technical Report UT CS-97-369, LAPACK Working Note No.129, University of Tennessee, 1997.

[17] 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 No.95).

[18] 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 No.55).

[19] 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 No.65).

[20] 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 No.57).

[21] 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.

[22] 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.

[23] 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.

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

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

[26] 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.

[27] F. Desprez, J. Dongarra, A. Petitet, C. Randriamaro, and Y. Robert. Scheduling Block-Cyclic Array Redistribution. Technical Report UT CS-97-349, LAPACK Working Note No.120, University of Tennessee, 1997. (To appear in IEEE Transactions on Parallel and Distributed Systems).

[28] J. Dongarra and R. van de Geijn. Two dimensional Basic Linear Algebra Communication Subprograms. Technical Report UT CS-91-138, LAPACK Working Note No.37, University of Tennessee, 1991.

[29] 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 No.43).

[30] J. Dongarra and D. Walker. Software Libraries for Linear Algebra Computations on High Performance Computers. *SIAM Review*, 37(2):151–180, 1995.

[31] J. Dongarra and R. C. Whaley. A User's Guide to the BLACS v1.0. Technical Report UT CS-95-281, LAPACK Working Note No.94, University of Tennessee, 1995. (http://www.netlib.org/blacs/).

[32] 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.

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

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

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

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

[37] 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.

[38] 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 No.79, University of Tennessee, 1994.

[39] 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.

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

[41] 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.

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

[43] E. Kalns and L. Ni. Processor Mapping Techniques towards Efficient Data Redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):1234–1247, 1995.

[44] 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.

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

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

[47] 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.

[48] 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.

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

[50] Y. Lim, P. Bhat, and V. Prasanna. Efficient Algorithms for Block-Cyclic Redistribution of Arrays. Technical Report CENG 97-10, Department of Electrical Engineering - Systems, University of Southern California, Los Angeles, CA, 1997.

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

[52] A. Petitet. *Algorithmic Redistribution Methods for Block Cyclic Decompositions*. PhD thesis, University of Tennessee, Knoxville, 1996. (also LAPACK Working Note No.128).

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

[54] P. Strazdins. Matrix Factorization using Distributed Panels on the Fujitsu AP1000. In *Proceedings of the IEEE First International Conference on Algorithms And Architectures for Parallel Processing (ICA3PP-95)*, Brisbane, 1995.

[55] P. Strazdins and H. Koesmarno. A High Performance Version of Parallel LAPACK: Preliminary Report. In *Proceedings of the Sixth Parallel Computing Workshop*, Fujitsu Parallel Computing Center, 1996.

[56] 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.

[57] 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.

[58] R. van de Geijn and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. Technical Report UT CS-95-286, LAPACK Working Note No.96, University of Tennessee, 1995.

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

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

[61] L. Wang, J. Stichnoth, and S. Chatterjee. Runtime performance of parallel array assignment: an empirical study. In *Proceedings of Supercomputing '96*. Sponsored by ACM SIGARCH and IEEE Computer Society, 1996. (ACM Order Number: 415962, IEEE Computer Society Press Order Number: RS00126. http://www.supercomp.org/sc96/proceedings/).

[62] R. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. Technical Report UT CS-97-366, LAPACK Working Note No.131, University of Tennessee, 1997.