# Impact of Mixed–Parallelism on Parallel Implementations of Strassen and Winograd Matrix Multiplication Algorithms

Frédéric Desprez, Frédéric Suter

## No 4482

June 2002

THÈME 1

*Rapport de recherche*

# Impact of Mixed–Parallelism on Parallel Implementations of Strassen and Winograd Matrix Multiplication Algorithms

Frédéric Desprez, Frédéric Suter

Thème 1 — Réseaux et systèmes
Projet ReMaP

**Abstract:** In this paper we study the impact of the simultaneous exploitation of data– and task–parallelism on Strassen and Winograd matrix multiplication algorithms. We present two mixed–parallel implementations. The former follows the phases of the original algorithms while the latter has been designed as the result of a list scheduling algorithm. We give a theoretical comparison, in terms of memory usage and execution time, between our algorithms and classical data–parallel implementations. This analysis is corroborated by experiments. Finally we give some hints about an heterogeneous version of our algorithms.

**Key-words:** Mixed–parallelism, matrix product, Strassen, Winograd.

*(Résumé : tsvp)*

# Impact du parallélisme mixte sur des implémentations parallèles des algorithmes de produits de matrices de Strassen et Winograd

**Résumé :** Dans cet article nous étudions l'impact de l'utilisation simultanée du parallélisme de tâches et du parallélisme de données sur les algorithmes de produit de matrices de Strassen et Winograd. Nous présentons deux implémentations à parallélisme mixte. La première suit les phases originales des algorithmes tandis que la seconde a été conçue comme le résultat d'un algorithme d'ordonnancement par liste. Nous donnons une comparaison théorique à la fois en termes de consommation mémoire qu'en coût de calcul, entre nos algorithmes et des implémentations data-parallèles classiques. Cette analyse est corroborée par des expérimentations. Enfin, nous donnons quelques suggestions pour une version hétérogène de nos algorithmes.

**Mots-clé :** Parallélisme mixte, produit de matrices, Strassen, Winograd.

# 1   Introduction

Parallel scientific applications can be divided in two major classes: *data–* and *task–parallel* applications. The former consists in applying the same operation in parallel on different elements of a data set, while the latter is defined to be concurrent computations on different data sets. These two classes can be combined to get a simultaneous exploitation of data– and task–parallelism, so called *mixed–parallelism*. In mixed–parallel applications, several data–parallel computations can be executed concurrently in a task–parallel way. Mixed–parallelism programming employs a M–SPMD (Multiple SPMD) style which is the combination of both task–parallelism (MPMD) and data-parallelism (SPMD). Such an exploitation of mixed–parallelism has many advantages. One of them is the ability to increase scalability because it allows the use of more parallelism when the maximal amount of data– or task–parallelism that can be exploited is reached. A good overview of this topic is given in [2]. Most of the researches about the simultaneous exploitation of data– and task–parallelism have been done in the area of programming languages to give simple high level accesses to more parallelism and in the area of compilers, where problems such as scheduling and allocation of concurrent data–parallel tasks are studied [19]. In [18], Ramaswamy introduces a structure to describe mixed–parallel programs: the Macro Dataflow Graph (MDG), a direct acyclic graph where nodes represent sequential or data–parallel computations and edges represent precedence constraints, with two distinguished nodes, one preceding and one succeeding all other nodes. Once the MDG is extracted from the code, a two step algorithm is applied to place and schedule tasks on the computing resources.

On a second hand matrix multiplication is the kernel of many scientific applications [16, 21] and several parallel implementations have been proposed, most of them using data–parallelism. Some algorithms substitute multiplications by additions and thus reduce the number of multiplications computed. Strassen [22] and Winograd [12] are such algorithms that are best suited for a practical implementation. They have been extensively studied on monoprocessor machines to increase the computational performances of numerical applications [1, 8, 14, 15, 23]. Several parallel implementations of Strassen and Winograd algorithms have been proposed. In [5], data-parallel implementations of Strassen and Winograd on a SIMD machine are presented. In [9], authors present task-parallel implementations of Strassen algorithm's on rings of processors that performs better than the classical ring algorithm. A mixed parallel implementation of Winograd is presented in [11]. The classical parallel algorithm is used as a kernel for a task parallel version of Winograd on hyper-grids with $7^k$ processors. Both implementations of [13] and [8] use data-parallel algorithms with efficient data-distributions. Finally, Ramaswamy [18] uses Strassen as a benchmark code for the Paradigm parallelization tool. Mixed parallelism is obtained through a two steps approach (computation of the number of processors needed and then scheduling of the parallel tasks). The data distribution is chosen by the tool and no library is used for low level kernels.

Our motivation is to build efficient parallel algorithms for client–server applications. We thus assume that matrices are distributed on disjoint grids of processors because of previous computations. Let compute the operation $C = AB$, where $A$ and $B$ are two square matrices

of dimension $M$ distributed on two separate square grids of processors of the same size. Moreover $C$ is distributed on the same grid as $A$. The "classical" way to compute this product is to align matrices and then call an efficient parallel routine, using a library like ScaLAPACK or a data–parallel implementation of Strassen algorithm. At the end of the computation, the result is finally redistributed. In this paper we propose a mixed way that keeps all matrices in place and evenly distributes tasks (e.g., additions[1] and products on matrix quarters) on the two grids. We use a sequential language like C or Fortran with high-performance libraries like ScaLAPACK [6] and its associated communication library, the BLACS. Mixed–parallelism is applied to Strassen and Winograd decompositions. Lower level products are computed using the standard high–performance level 3 PBLAS routine `pdgemm`.

The remainder of this paper is organized as follows. Section 2 recalls Strassen and Winograd algorithms. In Section 3, we present two optimized data–parallel implementations of these algorithms. Then in Section 4, we present the implementation choices leading us to the different mixed–parallel versions. Section 5 gives a theoretical evaluation of the different algorithms in terms of memory usage and execution time. Then in Section 6 we give some hints about an heterogeneous version of our mixed–parallel implementations. Experimental results that corroborate our theoretical study are shown in Section 7. Finally and before a conclusion, we explain why we only use one recursion step.

## 2    Strassen and Winograd Algorithms

In 1969, Strassen [22] introduced an algorithm to multiply $M \times M$ matrices which has a lower complexity than the classical $O(M^3)$. This algorithm and its MDG are presented in Figure 1. It is based on a scheme for the product of two $2 \times 2$ matrices which involves 7 multiplications and 18 additions instead of the usual 8 multiplications and 4 additions.

If we compute the total number of arithmetic operations done using Strassen algorithm, the 7 multiplications on quarters being computed with the traditional algorithm, we have: $7(2(M/2)^3 - (M/2)^2) + 18(M/2)^2 = (7/4)M^3 + (11/4)M^2$. The ratio between this complexity and the complexity required by the classical algorithm tends towards $7/8$ when $M$ gets large. This implies that for sufficiently large matrices, there is a theoretical gain of 12.5%.

This scheme can be easily applied to $2 \times 2$ block matrices. To compute the product $C = AB$, if $A$ and $B$ are distributed in square blocks of dimension $M/2$, we have:

$$\left( \begin{array}{cc} C_{11} & C_{12} \\ C_{21} & C_{22} \end{array} \right) = \left( \begin{array}{cc} A_{11} & A_{12} \\ A_{21} & A_{22} \end{array} \right) \left( \begin{array}{cc} B_{11} & B_{12} \\ B_{21} & B_{22} \end{array} \right)$$

Strassen algorithm can also be recursively applied on square matrices of dimension $M = 2^i$ to finally obtain a complexity of $O(M^{\log(7)}) = O(M^{2.807})$. Several other variations of this algorithm allow to handle matrices of arbitrary size, most of them being referenced or detailed in [15].

---

[1]In the following, we denote either additions or subtractions by addition.

**Input :** $Matrices\ A, B$

**Phase 1**

$T_1 = A_{11} + A_{22}$     $T_6 = B_{11} + B_{22}$
$T_2 = A_{21} + A_{22}$     $T_7 = B_{12} - B_{22}$
$T_3 = A_{11} + A_{12}$     $T_8 = B_{21} - B_{11}$
$T_4 = A_{21} - A_{11}$     $T_9 = B_{11} + B_{12}$
$T_5 = A_{12} - A_{22}$     $T_{10} = B_{21} + B_{22}$

**Phase 2**

$Q_1 = T_1 * T_6$         $Q_5 = T_3 * B_{22}$
$Q_2 = T_2 * B_{11}$       $Q_6 = T_4 * T_9$
$Q_3 = A_{11} * T_7$       $Q_7 = T_5 * T_{10}$
$Q_4 = A_{22} * T_8$

**Phase 3**

$U_1 = Q_1 + Q_4$         $U_2 = Q_5 - Q_7$
$U_3 = Q_3 + Q_1$         $U_4 = Q_2 - Q_6$
$C_{11} = U_1 - U_2$       $C_{12} = Q_3 + Q_5$
$C_{21} = Q_2 + Q_4$       $C_{22} = U_3 - U_4$

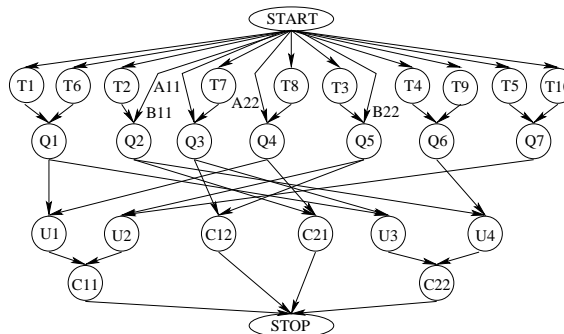**Output :** $C = (C_{ij})$



Figure 1: First level of recursion of Strassen algorithm (left) and its MDG representation (right).

Winograd variant of Strassen Algorithm, introduced in [12], uses the same number of multiplications but reduces the number of additions from 18 to 15. This algorithm and its MDG are presented in Figure 2.

# 3   Optimized Data–Parallel Algorithms

In this section, we present data–parallel implementations of Strassen and Winograd algorithms. As matrices $A$ and $B$ are distributed on two disjointed grids, they have to be aligned before computing the product using a ScaLAPACK parallel routine. This alignment implies two redistributions before the execution of the algorithm. Furthermore, at the end of the computation, we want $C$ to be aligned with $A$. This induces an other redistribution.

The following implementations for both algorithms focus on the reduction of temporary variables. They do thus not exhibit phases as shown in Figure 1 but an alternation between operations of phases 1 and 2. Figure 3 shows our data–parallel version of Strassen matrix multiplication algorithm, while Figure 4 shows a data–parallel implementation of Winograd variant, which can be found in [15].
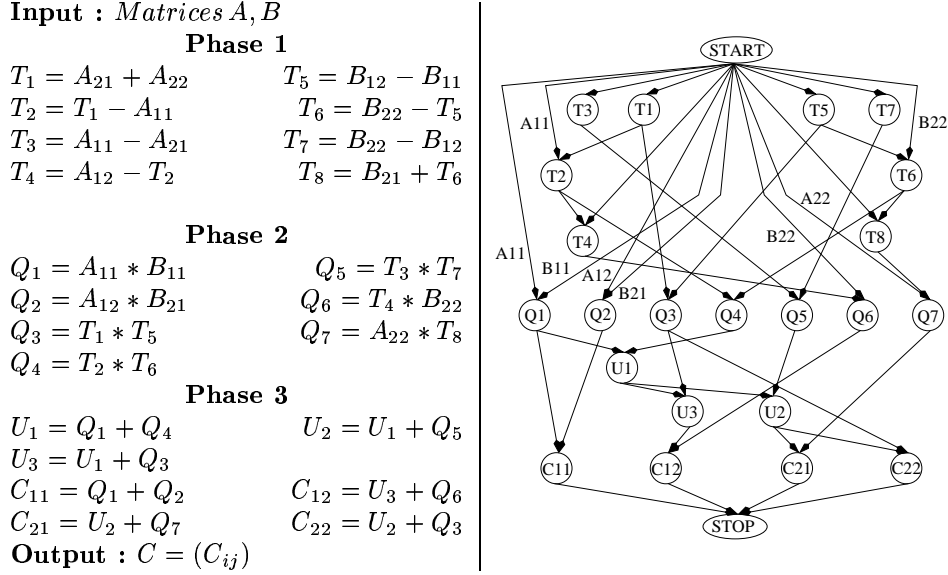
**Input :** *Matrices* $A, B$
**Phase 1**

$T_1 = A_{21} + A_{22}$     $T_5 = B_{12} - B_{11}$
$T_2 = T_1 - A_{11}$     $T_6 = B_{22} - T_5$
$T_3 = A_{11} - A_{21}$     $T_7 = B_{22} - B_{12}$
$T_4 = A_{12} - T_2$     $T_8 = B_{21} + T_6$

**Phase 2**

$Q_1 = A_{11} * B_{11}$     $Q_5 = T_3 * T_7$
$Q_2 = A_{12} * B_{21}$     $Q_6 = T_4 * B_{22}$
$Q_3 = T_1 * T_5$     $Q_7 = A_{22} * T_8$
$Q_4 = T_2 * T_6$

**Phase 3**

$U_1 = Q_1 + Q_4$     $U_2 = U_1 + Q_5$
$U_3 = U_1 + Q_3$
$C_{11} = Q_1 + Q_2$     $C_{12} = U_3 + Q_6$
$C_{21} = U_2 + Q_7$     $C_{22} = U_2 + Q_3$
**Output :** $C = (C_{ij})$



Figure 2: First level of recursion of the Winograd variant (left) and its MDG representation (right).

# 4    Mixed–Parallel Algorithms

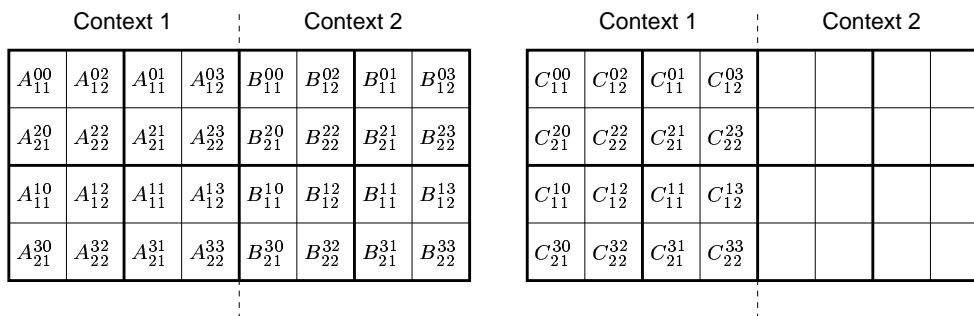## 4.1    Implementation choices

To benefit from the use of mixed-parallelism in Strassen algorithm, the strategy employed keeps matrices in place and distributes tasks among processors instead of aligning matrices before computing. As we use standard parallel numerical routines from ScaLAPACK, we need to keep the data distribution imposed by these kernels (full-block or block-cyclic distributions). Our goal is thus to reduce communications and to balance as much as possible computations among the processors. The two processor grids (or contexts) where matrices $A$ and $B$ are distributed are square and of dimension $p = 2^i$, but our algorithms are not limited to such values for $p$. This restriction is made in a pedagogical purpose. We consider these contexts are sub-grids of a virtual rectangular processor grid of size $p \times 2p$ (leading to a total of $2p^2$ processors). Processors of this global grid are row major numbered from $[0, 0]$ in the upper left corner to $[p - 1, 2p - 1]$ in the lower right. We will keep this numbering from the global context to identify processors until the end of this paper.

It is important to note that parallel versions of Strassen algorithm are better than the standard matrix multiplication algorithm if accumulations of matrices can be computed in linear time without communications. If $A$ (or $B$) is distributed following a *full-block* distribution, the addition of two quarters will introduce communications. To avoid these

**Input :** Matrices A and B
Temporary variables: $X$, $Y$, $Z$, $R_1$, $R_2$, $R_3$

| Store to: | | Computation | Algorithmic Variable |
|---|---|---|---|
| | | Redistribute $A \rightarrow X$ | |
| | | Redistribute $B \rightarrow Y$ | |
| | | | |
| $R_1$ | $\leftarrow$ | $X_{21} - X_{11}$ | $T_4$ |
| $R_2$ | $\leftarrow$ | $Y_{11} + Y_{12}$ | $T_9$ |
| $R_3$ | $\leftarrow$ | $R_1 * R_2$ | $Q_6$ |
| $R_1$ | $\leftarrow$ | $X_{21} + X_{22}$ | $T_2$ |
| $R_2$ | $\leftarrow$ | $R_1 * Y_{11}$ | $Q_2$ |
| $Z_{22}$ | $\leftarrow$ | $R_3 - R_2$ | $-U_4$ |
| $R_1$ | $\leftarrow$ | $Y_{21} - Y_{11}$ | $T_8$ |
| $R_3$ | $\leftarrow$ | $X_{22} * R_1$ | $Q_4$ |
| $Z_{21}$ | $\leftarrow$ | $R_2 + R_3$ | $Z_{21}$ |
| $R_1$ | $\leftarrow$ | $X_{11} + X_{22}$ | $T_1$ |
| $R_2$ | $\leftarrow$ | $Y_{11} + Y_{22}$ | $T_6$ |
| $Z_{11}$ | $\leftarrow$ | $R_1 * R_2$ | $Q_1$ |
| $Z_{22}$ | $\leftarrow$ | $Z_{22} + Z_{11}$ | |
| $Z_{11}$ | $\leftarrow$ | $Z_{11} + R_3$ | $U_1$ |
| $R_1$ | $\leftarrow$ | $Y_{12} - Y_{22}$ | $T_7$ |
| $R_2$ | $\leftarrow$ | $X_{11} * R_1$ | $Q_3$ |
| $Z_{22}$ | $\leftarrow$ | $Z_{22} + R_2$ | $Z_{22}$ |
| $R_1$ | $\leftarrow$ | $X_{11} + X_{12}$ | $T_3$ |
| $R_3$ | $\leftarrow$ | $R_1 * Y_{22}$ | $Q_5$ |
| $Z_{12}$ | $\leftarrow$ | $R_2 + S_3$ | $Z_{12}$ |
| $Z_{11}$ | $\leftarrow$ | $Z_{11} - R_3$ | |
| $R_1$ | $\leftarrow$ | $X_{12} - X_{22}$ | $T_5$ |
| $R_2$ | $\leftarrow$ | $Y_{21} + Y_{22}$ | $T_{10}$ |
| $R_3$ | $\leftarrow$ | $R_1 * R_2$ | $Q_7$ |
| $Z_{11}$ | $\leftarrow$ | $Z_{11} + R_3$ | $Z_{11}$ |
| | | | |
| | | Redistribute $Z \rightarrow C$ | |
| **Output :** | | $C = (C_{ij})$ | |

Figure 3: Data–parallel Strassen matrix multiplication algorithm.

communications, each processor involved in the computation must own a part of each matrix quarters. The bidimensional *block-cyclic* distribution is the most adapted to obtain such property, but the block size parameter has to be carefully chosen. Indeed a too little block size will induce extra memory accesses and will increase communication cost as we will see in section 5.2. $R = M/2p$ is the maximum block size that allows the local computation of all additions. Figure 5 shows an example of the chosen data distribution when $p = 2$. For each block, subscript gives its matrix quarter and superscript corresponds to the block-cyclic distribution.

**Input :** Matrices A and B
Temporary variables: $X$, $Y$, $Z$, $R_1$, $R_2$

| Store to: | | Computation | Algorithmic Variable |
|---|---|---|---|
| | | Redistribute $A \rightarrow X$ | |
| | | Redistribute $B \rightarrow Y$ | |
| | | | |
| $R_1$ | $\leftarrow$ | $X_{11} - X_{21}$ | $T_3$ |
| $R_2$ | $\leftarrow$ | $Y_{22} - Y_{12}$ | $T_7$ |
| $Z_{11}$ | $\leftarrow$ | $R_1 * R_2$ | $Q_5$ |
| $R_1$ | $\leftarrow$ | $X_{21} + X_{22}$ | $T_1$ |
| $R_2$ | $\leftarrow$ | $Y_{12} - Y_{11}$ | $T_5$ |
| $Z_{22}$ | $\leftarrow$ | $R_1 * R_2$ | $Q_3$ |
| $R_1$ | $\leftarrow$ | $R_1 - X_{11}$ | $T_2$ |
| $R_2$ | $\leftarrow$ | $Y_{22} - R_2$ | $T_6$ |
| $Z_{21}$ | $\leftarrow$ | $R_1 * R_2$ | $Q_4$ |
| $R_1$ | $\leftarrow$ | $X_{12} - R_1$ | $T_4$ |
| $R_2$ | $\leftarrow$ | $Y_{21} - R_2$ | $T_8$ |
| $Z_{12}$ | $\leftarrow$ | $R_1 * Y_{22}$ | $Q_6$ |
| $Z_{12}$ | $\leftarrow$ | $Z_{12} + Z_{22}$ | |
| $R_1$ | $\leftarrow$ | $X_{11} * Y_{11}$ | $Q_1$ |
| $Z_{21}$ | $\leftarrow$ | $Z_{21} + R_1$ | $U_2$ |
| $Z_{12}$ | $\leftarrow$ | $Z_{21} * Z_{12}$ | $Z_{12}$ |
| $Z_{21}$ | $\leftarrow$ | $Z_{21} + Z_{11}$ | $U_3$ |
| $Z_{11}$ | $\leftarrow$ | $X_{22} * R_2$ | $Q_2$ |
| $Z_{11}$ | $\leftarrow$ | $Z_{11} + R_1$ | $Z_{11}$ |
| $R_1$ | $\leftarrow$ | $X_{21} * R_2$ | $Q_7$ |
| $Z_{21}$ | $\leftarrow$ | $Z_{21} + R_1$ | $Z_{21}$ |
| | | | |
| | | Redistribute $Z \rightarrow C$ | |
| **Output :** | | $C = (C_{ij})$ | |

Figure 4: Data–parallel Winograd matrix multiplication algorithm.

## 4.2   Strassen

Once contexts are defined, tasks from the MDG have to be assigned as evenly as possible. We have developed two mixed–parallel implementations following two different approaches. The first is inspired of the basic representation of Strassen algorithm presented in Figure 1, as it follows the *phases* of this algorithm. The version given in this paper is an improvement of the one presented in [10]. The second implementation has been designed as it was the result of a *list* scheduling algorithm. We describe these two implementations in the next two sections.

| Context 1 | | | | Context 2 | | | |
|---|---|---|---|---|---|---|---|
| $A_{11}^{00}$ | $A_{12}^{02}$ | $A_{11}^{01}$ | $A_{12}^{03}$ | $B_{11}^{00}$ | $B_{12}^{02}$ | $B_{11}^{01}$ | $B_{12}^{03}$ |
| $A_{21}^{20}$ | $A_{22}^{22}$ | $A_{21}^{21}$ | $A_{22}^{23}$ | $B_{21}^{20}$ | $B_{22}^{22}$ | $B_{21}^{21}$ | $B_{22}^{23}$ |
| $A_{11}^{10}$ | $A_{12}^{12}$ | $A_{11}^{11}$ | $A_{12}^{13}$ | $B_{11}^{10}$ | $B_{12}^{12}$ | $B_{11}^{11}$ | $B_{12}^{13}$ |
| $A_{21}^{30}$ | $A_{22}^{32}$ | $A_{21}^{31}$ | $A_{22}^{33}$ | $B_{21}^{30}$ | $B_{22}^{32}$ | $B_{21}^{31}$ | $B_{22}^{33}$ |

| Context 1 | | | | Context 2 | | | |
|---|---|---|---|---|---|---|---|
| $C_{11}^{00}$ | $C_{12}^{02}$ | $C_{11}^{01}$ | $C_{12}^{03}$ | | | | |
| $C_{21}^{20}$ | $C_{22}^{22}$ | $C_{21}^{21}$ | $C_{22}^{23}$ | | | | |
| $C_{11}^{10}$ | $C_{12}^{12}$ | $C_{11}^{11}$ | $C_{12}^{13}$ | | | | |
| $C_{21}^{30}$ | $C_{22}^{32}$ | $C_{21}^{31}$ | $C_{22}^{33}$ | | | | |

Figure 5: Mapping of matrices $A$, $B$ and $C$ on a $2 \times 4$ processor grid.

### 4.2.1 Phase–based Algorithm

In this implementation we have studied which data are involved in each task to determine how to share computations out. In phase 1, all operations concern only $A$ or $B$. To keep locality, these tasks are mapped close to the data they use. In phase 2, each multiplication involves a matrix coming from context 1 and a matrix coming from context 2. When a computation needs data that are not distributed on the right context, copies of missing data from one context to the other have to be performed. In that phase, the placement of the tasks was driven by the reduction of communication cost. Products involving unmodified parts of $A$ (resp. $B$) are thus computed on context 1 (resp. context 2). Finally additions of phase 3 are executed on the context where matrix $C$ has to be distributed, in order to reduce the communication amount. This choice comes from the fact that addition cost is lower than communication cost for a given matrix size. These choices lead us to the phase–based mixed–parallel implementation of Strassen algorithm, shown by Figure 6.

### 4.2.2 List–Based Algorithm

In this section we detail the principle of our list–based algorithm. A major assumption is made in this implementation. Indeed we assume that we know how products are shared out before the beginning of the list scheduling algorithm. To determine that sharing, we consider the seven products as a chain, which is built as follow: There is a relation between two products if their results are both operands of a given addition in phase 3. Figure 7 shows the chain of products and the proposed cut-line determining the placement.

As for Strassen, we decide to compute accumulations of phase 1 close to the data they involve. Concerning the additions of phase 3, most of them ($U_1$,$U_2$, $U_4$, $C_{12}$, and $C_{21}$) can be computed without any data movement as we can see in Figure 7. Remaining additions are performed on the context where matrix $C$ has to be distributed to avoid extra communications.

**Input :** Matrix A
Temporary variables $R_1$, $R_2$, $R_3$, $R_4$, $R_5$, $R_6$, $R_7$

| Store to: | | Computation | Algorithmic variables |
|---|---|---|---|
| $R_1$ | $\leftarrow$ | $A_{11} + A_{22}$ | $T_1$ |
| $R_2$ | $\leftarrow$ | $A_{21} + A_{22}$ | $T_2$ |
| $R_3$ | $\leftarrow$ | $A_{11} + A_{12}$ | $T_3$ |
| $R_4$ | $\leftarrow$ | $A_{21} - A_{11}$ | $T_4$ |
| $R_5$ | $\leftarrow$ | $A_{12} - A_{22}$ | $T_5$ |

Redistribute $R_6 \leftarrow S_2$
Redistribute $R_1 \rightarrow S_2$
Redistribute $R_1 \leftarrow S_3$
Redistribute $R_2 \rightarrow S_3$
Redistribute $R_2 \leftarrow S_4$
Redistribute $R_3 \rightarrow S_4$
Redistribute $R_3 \leftarrow S_5$

| Store to: | | Computation | Algorithmic variables |
|---|---|---|---|
| $R_7$ | $\leftarrow$ | $A_{11} * R_6$ | $Q_3$ |
| $R_6$ | $\leftarrow$ | $A_{22} * R_1$ | $Q_4$ |
| $R_1$ | $\leftarrow$ | $R_4 * R_2$ | $Q_6$ |
| $R_2$ | $\leftarrow$ | $R_5 * R_3$ | $Q_7$ |

Redistribute $R_3 \leftarrow S_5$
Redistribute $R_4 \leftarrow S_6$
Redistribute $R_5 \leftarrow S_7$

| Store to: | | Computation | Algorithmic variables |
|---|---|---|---|
| $C_{11}$ | $\leftarrow$ | $R_3 + R_6$ | |
| $C_{11}$ | $\leftarrow$ | $C_{11} - R_5$ | |
| $C_{11}$ | $\leftarrow$ | $C_{11} + R_2$ | $C_{11}$ |
| $C_{12}$ | $\leftarrow$ | $R_7 + R_5$ | $C_{12}$ |
| $C_{21}$ | $\leftarrow$ | $R_4 + R_6$ | $C_{21}$ |
| $C_{22}$ | $\leftarrow$ | $R_3 + R_7$ | |
| $C_{22}$ | $\leftarrow$ | $C_{22} - R_4$ | |
| $C_{22}$ | $\leftarrow$ | $C_{22} + R_1$ | $C_{22}$ |

**Output :** $C = (C_{ij})$

---

**Input :** Matrix B
Temporary variables $S_1$, $S_2$, $S_3$, $S_4$, $S_5$, $S_6$, $S_7$

| Store to: | | Computation | Algorithmic variables |
|---|---|---|---|
| $S_1$ | $\leftarrow$ | $B_{11} + B_{22}$ | $T_6$ |
| $S_2$ | $\leftarrow$ | $B_{12} - B_{22}$ | $T_7$ |
| $S_3$ | $\leftarrow$ | $B_{21} - B_{11}$ | $T_8$ |
| $S_4$ | $\leftarrow$ | $B_{11} + B_{12}$ | $T_9$ |
| $S_5$ | $\leftarrow$ | $B_{21} + B_{22}$ | $T_{10}$ |

| Store to: | | Computation | Algorithmic variables |
|---|---|---|---|
| $S_5$ | $\leftarrow$ | $S_2 * S_1$ | $Q_1$ |
| $S_6$ | $\leftarrow$ | $S_3 * B_{11}$ | $Q_2$ |
| $S_7$ | $\leftarrow$ | $S_4 * B_{22}$ | $Q_5$ |

Figure 6: Phased–based mixed implemention of Strassen algorithm for Context 1 (left) and 2 (right).
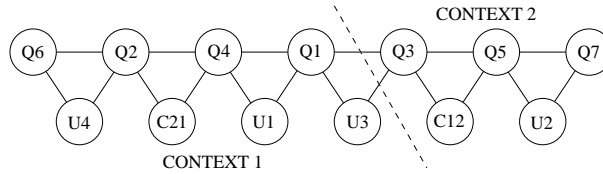


Figure 7: Chain of products and cut-line for Strassen algorithm.

To determine the scheduling of our algorithm, we use a list of ready tasks. These tasks are: additions, products, sends or receives. Addition and product tasks are considered ready as soon as both operands have been produced. For communication operations, a task is ready once a data to transfer is produced.

The choice of the ready task to execute is driven by several criteria. The most important is the aim to balance computations between contexts as long as possible. Then communication operations have priority over additions. Finally additions which produce a ready task have priority over the other ones. Following these rules we obtain the algorithm presented in Figure 8.

## 4.3   Winograd

To implement the phase- and list–based mixed–parallel versions of the Winograd variant, we kept the same notion of contexts and the same data distribution presented in section 4.1. But several optimizations have been made in both implementations due to the properties of the Winograd MDG. We detail these optimizations in this section.

### 4.3.1   Phase–based algorithm

This implementation of Winograd variant have been implemented aiming at reducing the communication amount. To achieve this goal we used the following optimizations. First we used redundant computation. Indeed the computation of $T_6$ can be performed on context 1 as both operands ($B_{22}$ and $T_5$) are already redistributed for the computation of $Q_6$ and $Q_3$ respectively. It allows us to exchange 6 matrix quarters instead of 7 between phases 1 and 2. Then phase 3 is computed only on one context as for Strassen. This choice unbalances the amount of computation between contexts but reduces the amount of data exchanged with regard to the implementation presented in [10]. Both algorithms for contexts 1 and 2 are presented in Figure 9.

### 4.3.2   List–based algorithm

As for list–based implementation of Strassen algorithm, we tried to find a chain of products. The structure obtained is not so simple, as shown in Figure 10, but it is still possible to determine a cut-line inducing a good task mapping. The algorithm presented in Figure 11 follows this mapping. We can see that the two additions of phase 3 that are computed on context 2 are overlapped by the computation of $Q_6$ on context 1.

## 5   Theoretical Evaluation

In this section, we evaluate the theoretical costs, in terms of memory usage and execution time, of different versions of Strassen and Winograd algorithms: data–parallel Strassen, phased- and list–based mixed–parallel Strassen, data–parallel Winograd, phased- and list–based mixed–parallel Winograd.

| **Store to:** | | **Computation** | **Algorithmic variables** | | **Store to:** | | **Computation** | **Algorithmic variables** |
|---|---|---|---|---|---|---|---|---|
| | | | | Redistribute $A_{11} \rightarrow S_1$ | | | | |
| | | | | Redistribute $R_1 \leftarrow B_{11}$ | | | | |
| $R_2$ | $\leftarrow$ | $A_{21} + A_{22}$ | $T_2$ | | $S_2$ | $\leftarrow$ | $B_{12} - B_{22}$ | $T_7$ |
| $R_3$ | $\leftarrow$ | $R_2 * R_1$ | $Q_2$ | | $S_3$ | $\leftarrow$ | $S_1 * S_2$ | $Q_3$ |
| $R_1$ | $\leftarrow$ | $A_{11} + A_{12}$ | $T_3$ | | $S_1$ | $\leftarrow$ | $B_{21} - B_{11}$ | $T_8$ |
| | | | | Redistribute $R_2 \leftarrow S_1$ | | | | |
| | | | | Redistribute $R_1 \rightarrow S_1$ | | | | |
| $R_1$ | $\leftarrow$ | $A_{22} * R_2$ | $Q_4$ | | $S_2$ | $\leftarrow$ | $S_1 * B_{22}$ | $Q_5$ |
| $C_{21}$ | $\leftarrow$ | $R_1 + R_3$ | $C_{21}$ | | $S_1$ | $\leftarrow$ | $S_2 + S_3$ | $C_{12}$ |
| | | | | Redistribute $C_{12} \leftarrow S_1$ | | | | |
| $R_2$ | $\leftarrow$ | $A_{21} - A_{11}$ | $T_4$ | | $S_1$ | $\leftarrow$ | $B_{21} + B_{22}$ | $T_{10}$ |
| $R_4$ | $\leftarrow$ | $A_{12} - A_{22}$ | $T_5$ | | $S_4$ | $\leftarrow$ | $B_{11} + B_{12}$ | $T_9$ |
| | | | | Redistribute $R_4 \rightarrow S_5$ | | | | |
| | | | | Redistribute $R_4 \leftarrow S_4$ | | | | |
| $C_{22}$ | $\leftarrow$ | $R_2 * R_4$ | $Q_6$ | | $S_4$ | $\leftarrow$ | $S_5 * S_1$ | $Q_7$ |
| $C_{22}$ | $\leftarrow$ | $C_{22} - R_3$ | $-U_4$ | | $S_1$ | $\leftarrow$ | $S_4 - S_2$ | $-U_2$ |
| | | | | Redistribute $C_{11} \leftarrow S_1$ | | | | |
| $R_2$ | $\leftarrow$ | $A_{11} + A_{22}$ | $T_1$ | | $S_4$ | $\leftarrow$ | $B_{11} + B_{22}$ | $T_6$ |
| | | | | Redistribute $R_3 \leftarrow S_3$ | | | | |
| $C_{22}$ | $\leftarrow$ | $C_{22} + R_3$ | | | | | | |
| | | | | Redistribute $R_3 \leftarrow S_4$ | | | | |
| $R_4$ | $\leftarrow$ | $R_2 * R_3$ | $Q_1$ | | | | | |
| $C_{11}$ | $\leftarrow$ | $C_{11} + R_4$ | | | | | | |
| $C_{11}$ | $\leftarrow$ | $C_{11} + R_1$ | $C_{11}$ | | | | | |
| $C_{22}$ | $\leftarrow$ | $C_{22} + R_4$ | $C_{22}$ | | | | | |

**Input :** Matrix A
Temporary variables $R_1$, $R_2$, $R_3$, $R_4$

**Input :** Matrix B
Temporary variables $S_1$, $S_2$, $S_3$, $S_4$, $S_5$

**Output :** $C = (C_{ij})$

Figure 8: List–based mixed implementation of Strassen algorithm for Context 1 (left) and 2 (right).

**Input :** Matrix A
Temporary variables $R_1$, $R_2$, $R_3$, $R_4$, $R_5$, $R_6$, $R_7$

**Input :** Matrix B
Temporary variables $S_1$, $S_2$, $S_3$, $S_4$, $S_5$, $S_6$

| Store to: | | Computation | Algorithmic variables | Store to: | | Computation | Algorithmic variables |
|---|---|---|---|---|---|---|---|
| $R_1$ | $\leftarrow$ | $A_{21} + A_{22}$ | $T_1$ | $S_1$ | $\leftarrow$ | $B_{12} - B_{11}$ | $T_5$ |
| $R_2$ | $\leftarrow$ | $R_1 - A_{11}$ | $T_2$ | $S_2$ | $\leftarrow$ | $B_{22} - S_1$ | $T_6$ |
| $R_3$ | $\leftarrow$ | $A_{11} - A_{21}$ | $T_3$ | $S_3$ | $\leftarrow$ | $B_{22} - B_{12}$ | $T_7$ |
| $R_4$ | $\leftarrow$ | $A_{12} - R_2$ | $T_4$ | $S_4$ | $\leftarrow$ | $B_{21} + S_2$ | $T_8$ |

Redistribute $R_3 \rightarrow S_5$
Redistribute $R_5 \leftarrow S_1$
Redistribute $A_{11} \rightarrow S_6$
Redistribute $R_6 \leftarrow B_{22}$
Redistribute $A_{22} \rightarrow S_1$
Redistribute $R_7 \leftarrow B_{21}$

| Store to: | | Computation | Algorithmic variables | Store to: | | Computation | Algorithmic variables |
|---|---|---|---|---|---|---|---|
| $R_3$ | $\leftarrow$ | $R_1 * R_5$ | $Q_3$ | $S_2$ | $\leftarrow$ | $S_6 * B_{11}$ | $Q_1$ |
| $R_1$ | $\leftarrow$ | $R_4 * R_6$ | $Q_6$ | $S_6$ | $\leftarrow$ | $S_5 * S_3$ | $Q_5$ |
| $R_4$ | $\leftarrow$ | $A_{12} * R_7$ | $Q_2$ | $S_5$ | $\leftarrow$ | $S_1 * S_4$ | $Q_7$ |
| $R_6$ | $\leftarrow$ | $R_6 - R_5$ | $T_6$ | | | | |
| $R_5$ | $\leftarrow$ | $R_2 * R_6$ | $Q_4$ | | | | |

Redistribute $R_6 \leftarrow S_2$

| Store to: | | Computation | Algorithmic variables |
|---|---|---|---|
| $C_{11}$ | $\leftarrow$ | $R_6 + R_4$ | $C_{11}$ |
| $R_4$ | $\leftarrow$ | $R_6 + R_5$ | $T_1$ |

Redistribute $R_2 \leftarrow S_6$
Redistribute $R_5 \leftarrow S_5$

| Store to: | | Computation | Algorithmic variables |
|---|---|---|---|
| $R_6$ | $\leftarrow$ | $R_4 + R_2$ | $T_2$ |
| $C_{12}$ | $\leftarrow$ | $R_4 + R_3$ | |
| $C_{12}$ | $\leftarrow$ | $C_{12} + R_1$ | $C_{12}$ |
| $C_{21}$ | $\leftarrow$ | $R_6 + R_5$ | $C_{21}$ |
| $C_{22}$ | $\leftarrow$ | $R_6 + R_3$ | $C_{22}$ |

**Output :** $C = (C_{ij})$

Figure 9: Phased–based mixed implementation of Winograd algorithm for Context 1 (left) and 2 (right).
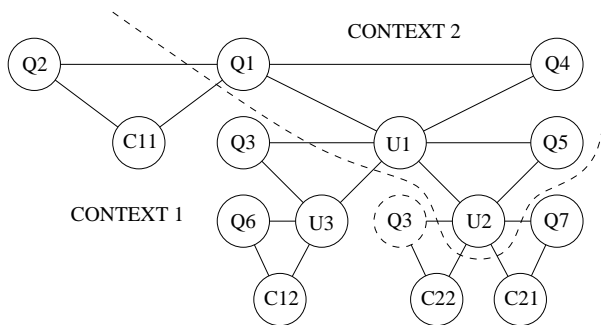
Figure 10: Chain of products and cut-line for Winograd Variant.

## 5.1  Temporary Variables Allocation and Memory Usage

The results of the 18 additions and the 7 multiplications of the Strassen algorithm have to be stored in temporary variables. If we consider the algorithm presented in Figure 1(left), 10 temporary variables are needed in phase 1, 7 in phase 2, and 4 in phase 3. As we compute the product of matrices of size $M$ distributed on square grids of size $p \times p$, all these temporary variables are of dimension $M/2p$. With a naive policy for temporary variables allocation, this algorithm needs $21(M/2p)^2$ temporary elements on each processor. In this section, we study and compare the different parallel implementations with regard to their optimizations in terms of memory usage.

In the data–parallel implementations presented by Figure 3 and 4, the alignment of $A$ and $B$ implies the allocation of two temporary variables of size $M/p \times M/2p$. Furthermore, at the end of the computation, the alignment of $C$ induces an other temporary variable of the same size. In addition of these variables that do no not depend of the computation, Strassen algorithm needs three temporary variables each being of size $M/2p \times M/4p$. This algorithm thus needs at least $15M^2/8p^2$ temporary elements per processor. Concerning Winograd variant, the algorithm needs only two temporary variables of size $M/2p \times M/4p$ leading to a total amount of $7M^2/4p^2$ temporary elements per processor.

In [10] we proposed implementations of Strassen and Winograd algorithms using mixed–parallelism which need $2(M/p)^2$ temporary elements on each processor of context 1 and $3(M/p)^2$ temporary elements on each processor of context 2. The implementations presented in this paper clearly reduce this amount. This reduction comes from a change in the communication policy which has an influence upon the number of needed temporary elements. In [10] we did message grouping during communication phases to reduce the number of latencies. To do that, we used big temporary variables of size $M/p$. Here the temporary variables are smaller ($M/2p$ as we give importance to temporary variable reuse instead of message grouping.

The phase–based implementations of Strassen and Winograd algorithms thus need $7(M/2p)^2$ temporary elements per processor. As the use of list scheduling allows a bet-

**Input** : Matrix A
Temporary variables $R_1$, $R_2$, $R_3$

| Store to: | | Computation | Algorithmic variables | Store to: | | Computation | Algorithmic variables |
|---|---|---|---|---|---|---|---|
| | | | | | | **Input** : Matrix B | |
| | | | | | | Temporary variables $S_1$, $S_2$, $S_3$, $S_4$, $S_5$ | |
| | | | | Redistribute $A_{11} \rightarrow S_1$ | | | |
| | | | | Redistribute $R_1 \leftarrow B_{21}$ | | | |
| $C_{11}$ | $\leftarrow$ | $A_{12} * R_1$ | $Q_2$ | $S_2$ | $\leftarrow$ | $S_1 * B_{11}$ | $Q_1$ |
| $R_1$ | $\leftarrow$ | $A_{11} + A_{22}$ | $T_1$ | $S_1$ | $\leftarrow$ | $B_{12} - B_{11}$ | $T_5$ |
| $R_2$ | $\leftarrow$ | $A_{11} - A_{21}$ | $T_3$ | $S_3$ | $\leftarrow$ | $B_{22} - B_{12}$ | $T_7$ |
| | | | | Redistribute $R_3 \leftarrow S_1$ | | | |
| | | | | Redistribute $R_2 \rightarrow S_4$ | | | |
| $C_{22}$ | $\leftarrow$ | $R_1 * R_3$ | $Q_3$ | $S_5$ | $\leftarrow$ | $S_4 * S_3$ | $Q_5$ |
| $R_2$ | $\leftarrow$ | $R_1 - A_{11}$ | $T_2$ | $S_3$ | $\leftarrow$ | $B_{22} - S_1$ | $T_6$ |
| $R_1$ | $\leftarrow$ | $A_{12} - R_2$ | $T_4$ | $S_1$ | $\leftarrow$ | $B_{21} + S_3$ | $T_8$ |
| | | | | Redistribute $R_2 \rightarrow S_4$ | | | |
| | | | | Redistribute $R_2 \leftarrow S_1$ | | | |
| $C_{21}$ | $\leftarrow$ | $A_{22} * R_2$ | $Q_7$ | $S_1$ | $\leftarrow$ | $S_4 * S_3$ | $Q_4$ |
| | | | | Redistribute $R_2 \leftarrow B_{22}$ | | | |
| $C_{12}$ | $\leftarrow$ | $R_1 * R_2$ | $Q_6$ | $S_1$ | $\leftarrow$ | $S_1 + S_2$ | $U_1$ |
| | | | | | | $S_5$ | $\leftarrow$ | $S_5 + S_1$ | $U_2$ |
| | | | | Redistribute $R_1 \leftarrow S_2$ | | | |
| $C_{11}$ | $\leftarrow$ | $C_{11} + R_1$ | $C_{11}$ | | | | |
| | | | | Redistribute $R_1 \leftarrow S_1$ | | | |
| $C_{12}$ | $\leftarrow$ | $C_{12} + R_1$ | | | | | |
| $C_{12}$ | $\leftarrow$ | $C_{12} + C_{22}$ | $C_{12}$ | | | | |
| | | | | Redistribute $R_1 \leftarrow S_5$ | | | |
| $C_{21}$ | $\leftarrow$ | $C_{21} + R_1$ | $C_{21}$ | | | | |
| $C_{22}$ | $\leftarrow$ | $C_{22} + R_1$ | $C_{22}$ | | | | |

**Output** :   $C = (C_{ij})$

Figure 11: List–based mixed implementation of Winograd algorithm for Context 1 (left) and 2 (right).

ter reuse of temporary variables, the list–based implementation of Strassen only needs four temporary variables for each processor owning $A$ and $C$, i.e., those of the first context, and five on those owning $B$, each of these variables being of size $M/2p$. Concerning Winograd variant the list–based implementation of Winograd needs less temporary variables (only 3 of size $M/2p$) on processors of the first context than on those of the second one (still 5).

As we want to evaluate the maximum amount of memory needed by our algorithms, including input and output data, we will only consider the memory usage of processors allocating this maximal amount and not an average value. Figure 12 shows the needed memory space of the different versions for $p = 2$ when $M$ varies. Input and ouput data are taken into account in this estimation.
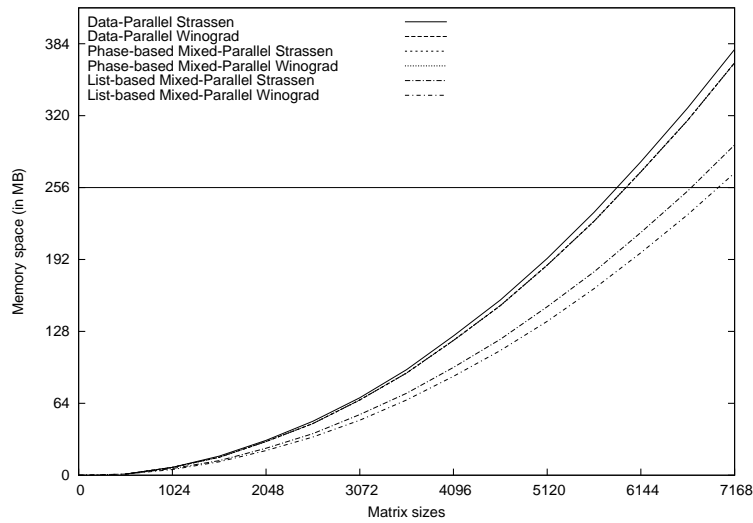


Figure 12: Memory space needed for the different implementations when $p = 2$.

We can see on this figure that the data–parallel implementation of Strassen algorithm is the worst in terms of memory usage. Data–parallel Winograd and phase–based mixed–parallel implementations need the same space. Then we find the list–based mixed–parallel implementations. We can conclude that with a limited available memory space per processor (for instance 256MB on Figure 12), mixed–parallelism using list-based algorithms allows to handle bigger sizes of matrices than data–parallel versions.

## 5.2 Time Cost Models

### 5.2.1 Parallel Machine and Basic Operation Models

Our communication model is the classical $\beta + L\tau$ where $\beta$ is the network latency, $L$ the message size, and $\tau$ the inverse of the network bandwidth. As our network is switched, we do not have to take contention into account.

In [15], a classification of operations involved in Strassen algorithm is done because each of those may have different execution speed. We can distinguish two main classes: Matrix multiplication and matrix addition. the former is computed by the routine `pdgemm` of the PBLAS library while the latter is handled by a modified version of a PBLAS internal routine. Since our block size is carefully chosen, we do not have cache effects. Now, we give detailed models for each of the basic operations involved in our different algorithms.

**Matrix multiplication** In [7] a model is given for the ScaLAPACK routine `pdgemm`, which is used in our algorithm to compute the seven inner matrix multiplications. Let be $p$ the number of rows, and $q$ the number of columns of the processor grid and assume that involved matrices are square and of size $M$. We have the following cost model

$$T_{Mult} = time\left(dgemm, \left\lceil \frac{M}{p} \right\rceil, \left\lceil \frac{M}{q} \right\rceil, M\right) + (\tau_p^q + \tau_q^p)M^2 + \left(\beta_p^q + \beta_q^p\right)\left\lceil \frac{M}{R} \right\rceil.$$

where *time* is a routine allowing us to acquire the execution time of a sequential routines for a given set of calling parameters, `dgemm` is the sequential counterpart of `pdgemm`, $R$ is the block size, and $\tau_q^p$ and $\beta_q^p$ (resp. $\tau_p^q$ and $\beta_p^q$) are functions of grid topology and communication pattern for bandwidth and latency in a row (resp. column) broadcast.

**Matrix addition** Because of our data distribution, all additions are executed locally without communications as explained in Section 4.1. The cost of an addition is given by

$$T_{Add} = time(add, \left\lceil \frac{M}{p} \right\rceil, \left\lceil \frac{M}{q} \right\rceil).$$

### 5.2.2 Data–Parallel Algorithms

The main difference between Strassen and Winograd algorithms is their number of additions. But both algorithms need 7 products and 3 redistributions to align source data and bring back the result on context 1.

To be still able to perform additions without communications once matrices are aligned, the block size has to change during the redistribution. For instance, Figure 13 shows the new data distribution of matrix $A$ when $p = 2$. As in Figure 5, subscript gives, for each block, its matrix quarter and superscript corresponds to the block-cyclic distribution.

|  | Context 1 |  |  | Context 2 |  |  |  |
|---|---|---|---|---|---|---|---|
| $A_{11}^{00}$ | $A_{12}^{04}$ | $A_{11}^{01}$ | $A_{12}^{05}$ | $A_{11}^{02}$ | $A_{12}^{06}$ | $A_{11}^{03}$ | $A_{12}^{07}$ |
| $A_{11}^{20}$ | $A_{12}^{24}$ | $A_{11}^{21}$ | $A_{12}^{25}$ | $A_{11}^{22}$ | $A_{12}^{26}$ | $A_{11}^{23}$ | $A_{12}^{27}$ |
| $A_{21}^{40}$ | $A_{22}^{44}$ | $A_{21}^{41}$ | $A_{22}^{45}$ | $A_{21}^{42}$ | $A_{22}^{46}$ | $A_{21}^{43}$ | $A_{22}^{47}$ |
| $A_{21}^{60}$ | $A_{22}^{64}$ | $A_{21}^{61}$ | $A_{22}^{65}$ | $A_{21}^{62}$ | $A_{22}^{66}$ | $A_{21}^{63}$ | $A_{22}^{67}$ |
| $A_{11}^{10}$ | $A_{12}^{14}$ | $A_{11}^{11}$ | $A_{12}^{15}$ | $A_{11}^{12}$ | $A_{12}^{16}$ | $A_{11}^{13}$ | $A_{12}^{17}$ |
| $A_{11}^{30}$ | $A_{12}^{34}$ | $A_{11}^{31}$ | $A_{12}^{35}$ | $A_{11}^{32}$ | $A_{12}^{36}$ | $A_{11}^{33}$ | $A_{12}^{37}$ |
| $A_{21}^{50}$ | $A_{22}^{54}$ | $A_{21}^{51}$ | $A_{22}^{55}$ | $A_{21}^{52}$ | $A_{22}^{56}$ | $A_{21}^{53}$ | $A_{22}^{57}$ |
| $A_{21}^{70}$ | $A_{22}^{74}$ | $A_{21}^{71}$ | $A_{22}^{75}$ | $A_{21}^{72}$ | $A_{22}^{76}$ | $A_{21}^{73}$ | $A_{22}^{77}$ |

Figure 13: Mapping of matrix $A$ on a $2 \times 4$ processor grid after the alignment step.

To perform such a redistribution, we used the redistribution routine of ScaLAPACK [17], based on the caterpillar algorithm which is efficient for most communication pattern. Its principle is the following. Source and destination processors are considered as a single list. Each processor computes the communication pattern, i.e., the amount of data it has to exchange with any other involved processor, including itself. Once this pattern computed, each processor has *rendez-vous* with other processors to exchange data. Figure 14 shows how a list of 8 processors rolls like a caterpillar to ensure that all communications are performed. During even steps, 2 processors execute self-communications, i.e., memory copies.

To estimate the cost of the redistribution involved in the data–parallel algorithms, we first have to determine the communication patterns. In this case each processor of context 1 (i.e., $p^2$ sending processors) sends 4 blocks of size $M/4p \times M/4p$ to 4 processors. The entire matrix is thus communicated. To simplify our analysis we assume there is one, and only one, communication per step of the caterpillar algorithm. Leading us to the following model for the alignment of a matrix from a $p \times p$ processor grid to a $p \times 2p$ processor grid

$$T_{Align} = M^2 \tau + 2p^2 \beta$$

Once matrices are aligned both algorithms perform 7 products even if the execution order is different. These products involve quarters of matrices of size $M$ distributed on a $p \times 2p$ processor grid. As said before the block size has been changed during the redistribution step and is now equal to $R' = M/4p$. The cost model corresponding to products is then
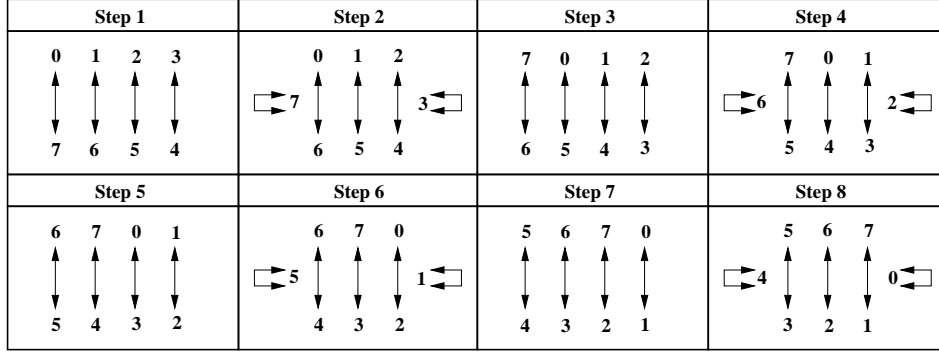
Figure 14: Communication pattern of the caterpillar algorithm.

$$
T_{Mult} = 7 \left( time \left( dgemm, \left\lceil \frac{\frac{M}{2}}{p} \right\rceil, \left\lceil \frac{\frac{M}{2}}{2p} \right\rceil, \frac{M}{2} \right) \right)
$$
$$
+ 7 \left( \left( \frac{M}{2} \right)^2 \left( \tau_{2p}^p + \tau_p^{2p} \right) + \frac{\frac{M}{2}}{\frac{M}{4p}} \left( \beta_{2p}^p + \beta_p^{2p} \right) \right).
$$

Models are distinguished for additions as 18 are computed in Strassen and 15 in Winograd. As for products, these additions involve quarters of matrices of size $M$ distributed on a $p \times 2p$ processor grid, leading to

$$
T_{Add}^{Strassen} = 18 \left( time \left( add, \left\lceil \frac{\frac{M}{2}}{p} \right\rceil, \left\lceil \frac{\frac{M}{2}}{2p} \right\rceil \right) \right), \text{ and}
$$
$$
T_{Add}^{Winograd} = 15 \left( time \left( add, \left\lceil \frac{\frac{M}{2}}{p} \right\rceil, \left\lceil \frac{\frac{M}{2}}{2p} \right\rceil \right) \right).
$$

If we sum these cost, we obtain

$$
T_{Strassen}^{Data//} = 7 \left( time \left( dgemm, \left\lceil \frac{M}{2p} \right\rceil, \left\lceil \frac{M}{4p} \right\rceil, \frac{M}{2} \right) \right) + 18 \left( time \left( add, \left\lceil \frac{M}{2p} \right\rceil, \left\lceil \frac{M}{4p} \right\rceil \right) \right)
$$
$$
+ \frac{7M^2}{4} \left( \tau_{2p}^p + \tau_p^{2p} \right) + 3M^2 \tau + 14p \left( \beta_{2p}^p + \beta_p^{2p} \right) + 6p^2 \beta,
$$

for Strassen and

$$T_{Winograd}^{Data//} = 7 \left( time \left( dgemm, \left\lceil \frac{M}{2p} \right\rceil, \left\lceil \frac{M}{4p} \right\rceil, \frac{M}{2} \right) \right) + 15 \left( time \left( add, \left\lceil \frac{M}{2p} \right\rceil, \left\lceil \frac{M}{4p} \right\rceil \right) \right)$$

$$+ \frac{7M^2}{4} \left( \tau_{2p}^p + \tau_p^{2p} \right) + 3M^2\tau + 14p \left( \beta_{2p}^p + \beta_p^{2p} \right) + 6p^2\beta,$$

for the Winograd variant.

### 5.2.3   Mixed–Parallel Strassen Algorithms

When a data is missing on a context to execute a computation, we have to perform a inter–context copy. In both implementations, we use locally blocking send and receive functions to perform these communications. Because the two sub-grids have same size and same shape, data exchange can be done in parallel between processor pairs. Processor $[MyRow, MyCol]$ from context 1 exchanges data with processor $[MyRow, MyCol + p]$ in context 2.

In both versions there are 10 Send/Receive operations where a matrix of size $M/2p$ is communicated from a context to the other. It gives us a communication cost model equal to

$$T_{Redist} = \frac{5}{2p^2} M^2\tau + 10\beta.$$

4 products are computed on the first context and 3 on the second one. As each product deals with matrices of size $M/2$, we have

$$T_{Mult} = 4 \left( time \left( dgemm, \left\lceil \frac{M}{2p} \right\rceil, \left\lceil \frac{M}{2p} \right\rceil, M/2 \right) + 2(M/2)^2 \tau_p^p + 2\frac{M/2}{M/2p} \beta_p^p \right)$$

$$= 4 \left( time(dgemm, R, R, M/2) \right) + 2M^2\tau_p^p + 8p\beta_p^p.$$

Concerning additions we have to distinguish phase- and list–based algorithm. Indeed as phase 3 is not handled the same way in both implementations, the critical path varies. It has 12 additions for the phased–based algorithm and only 10 for the list–based one. Assuming that all operations work on matrices of size $M/2$, we get

$$T_{Add}^{Phase} = 12 \left( time \left( add, \left\lceil \frac{M}{2p} \right\rceil, \left\lceil \frac{M}{2p} \right\rceil \right) \right) = 12 \left( time(add, R, R) \right).$$

$$T_{Add}^{List} = 10 \left( time \left( add, \left\lceil \frac{M}{2p} \right\rceil, \left\lceil \frac{M}{2p} \right\rceil \right) \right) = 10 \left( time(add, R, R) \right).$$

If we sum these costs, we get the following cost model for each version of mixed–parallel Strassen algorithm

$$T_{Strassen}^{Phase} = 4\left(time(dgemm,\ R,\ R,\ M/2)\right) + 12\left(time(add,\ R,\ R)\right)$$
$$+ 2M^2\tau_p^p + \frac{5}{2p^2}M^2\tau + 8p\beta_p^p + 10\beta.$$

$$T_{Strassen}^{List} = 4\left(time(dgemm,\ R,\ R,\ M/2)\right) + 10\left(time(add,\ R,\ R)\right)$$
$$+ 2M^2\tau_p^p + \frac{5}{2p^2}M^2\tau + 8p\beta_p^p + 10\beta.$$

### 5.2.4 Mixed–parallel Winograd Variants

In the Winograd variant of Strassen algorithm, the basic operations are the same as in Strassen version. If we consider the critical paths of phase- and list–based implementations we have: 8 sends/receives, 12 additions and 4 products for the phase–based implementation and 10 sends/receives, 9 additions, 4 products for the list–based implementation.

The cost model of the 4 products, the common part of both critical paths, has already been given in section 5.2.3. If we detail the remaining component of these critical paths, we have

$$T_{Redist}^{Phase} = \frac{9M^2}{4p^2}\tau + 9\beta,\ \text{and}$$
$$T_{Redist}^{List} = \frac{5M^2}{2p^2}\tau + 10\beta,$$

for the communications and

$$T_{Add}^{Phase} = 12\left(time\left(add,\ \left\lceil\frac{M}{2p}\right\rceil,\ \left\lceil\frac{M}{2p}\right\rceil\right)\right) = 12\left(time(add,\ R,\ R)\right),\ \text{and}$$

$$T_{Add}^{List} = 9\left(time\left(add,\ \left\lceil\frac{M}{2p}\right\rceil,\ \left\lceil\frac{M}{2p}\right\rceil\right)\right) = 9\left(time(add,\ R,\ R)\right),$$

for additions. If we sum all these costs (communications, additions and products) we obtain the following models for our mixed–parallel implementations of Winograd variant

$$T_{Winograd}^{Phase} = 4\left(time(dgemm,\ R,\ R,\ M/2)\right) + 12\left(time(add,\ R,\ R)\right)$$
$$+ 2M^2\tau_p^p + \frac{9}{4p^2}M^2\tau + 9p\beta_p^p + 8\beta.$$

$$T^{List}_{Winograd} = 4 \left(time(dgemm,\ R,\ R,\ M/2)\right) + 9 \left(time(add,\ R,\ R)\right)$$

$$+2M^2\tau^p_p + \frac{5}{2p^2}M^2\tau + 8p\beta^p_p + 10\beta.$$

### 5.2.5 Comparisons and Analysis

Tables 1 and 2 respectively give a summary of the different components of the computation and communication models for the different versions of Strassen and Winograd algorithms: data–parallel Strassen, phased- and list–based mixed–parallel Strassen, data–parallel Winograd, phased- and list–based mixed–parallel Winograd.

| Algorithm | Computation | | | | | | |
|---|---|---|---|---|---|---|---|
| | product | | | | addition | | |
| | nb | M | N | K | nb | M | N |
| Strassen Data | 7 | $\frac{M}{2p}$ | $\frac{M}{4p}$ | $\frac{M}{2}$ | 18 | $\frac{M}{2p}$ | $\frac{M}{4p}$ |
| Strassen Phase | 4 | $\frac{M}{2p}$ | $\frac{M}{2p}$ | $\frac{M}{2}$ | 12 | $\frac{M}{2p}$ | $\frac{M}{2p}$ |
| Strassen List | 4 | $\frac{M}{2p}$ | $\frac{M}{2p}$ | $\frac{M}{2}$ | 10 | $\frac{M}{2p}$ | $\frac{M}{2p}$ |
| Winograd Data | 7 | $\frac{M}{2p}$ | $\frac{M}{4p}$ | $\frac{M}{2}$ | 15 | $\frac{M}{2p}$ | $\frac{M}{4p}$ |
| Winograd Phase | 4 | $\frac{M}{2p}$ | $\frac{M}{2p}$ | $\frac{M}{2}$ | 12 | $\frac{M}{2p}$ | $\frac{M}{2p}$ |
| Winograd List | 4 | $\frac{M}{2p}$ | $\frac{M}{2p}$ | $\frac{M}{2}$ | 9 | $\frac{M}{2p}$ | $\frac{M}{2p}$ |

Table 1: Summary of computation costs.

| Algorithm | Communication | |
|---|---|---|
| | $M^2$ | Latency |
| Strassen Data | $3\tau + 7/4 \left(\tau^p_{2p} + \tau^{2p}_p\right)$ | $14p \left(\beta^p_{2p} + \beta^{2p}_p\right) + 6p^2\beta$ |
| Strassen Phase | $5\tau/2p^2 + 2\tau^p_p$ | $8p\beta^p_p + 10\beta$ |
| Strassen List | $5\tau/2p^2 + 2\tau^p_p$ | $8p\beta^p_p + 10\beta$ |
| Winograd Data | $3\tau + 7/4 \left(\tau^p_{2p} + \tau^{2p}_p\right)$ | $14p \left(\beta^p_{2p} + \beta^{2p}_p\right) + 6p^2\beta$ |
| Winograd Phase | $9\tau/4p^2 + 2\tau^p_p$ | $8p\beta^p_p + §9\beta$ |
| Winograd List | $5\tau/2p^2 + 2\tau^p_p$ | $8p\beta^p_p + 10\beta$ |

Table 2: Summary of communication costs.

In communication costs, values of $\tau_p^p$, $\tau_{2p}^p$, $\tau_p^{2p}$, $\beta_p^p$, $\beta_{2p}^p$, and $\beta_p^{2p}$ can be replaced by their actual values (assuming that we have a tree based broadcast), respectively $\lceil \log_2(p) \rceil * \tau/p$, $(\lceil \log_2(p) \rceil + 1) * \tau/p$, $(\lceil \log_2(p) \rceil + 1) * \tau/p$, $\lceil \log_2(p) \rceil \beta$, $(\lceil \log_2(p) \rceil + 1) * \beta$, and $(\lceil \log_2(p) \rceil + 1) * \beta$. Table 3 gives the new communication costs.

| Algorithm | Communication | |
|---|---|---|
| | $M^2 \tau$ | $\beta$ |
| Strassen Data | $3 + 7(2 \lceil \log_2(p) \rceil + 1)/4p$ | $14p(2 \lceil \log_2(p) \rceil + 1) + 6p^2$ |
| Strassen Phase | $5/2p^2 + 2 \lceil \log_2(p) \rceil /p$ | $8p \lceil \log_2(p) \rceil + 10$ |
| Strassen List | $5/2p^2 + 2 \lceil \log_2(p) \rceil /p$ | $8p \lceil \log_2(p) \rceil + 10$ |
| Winograd Data | $3 + 7(2 \lceil \log_2(p) \rceil + 1)/4p$ | $14p(2 \lceil \log_2(p) \rceil + 1) + 6p^2$ |
| Winograd Phase | $9/4p^2 + 2 \lceil \log_2(p) \rceil /p$ | $8p \lceil \log_2(p) \rceil + 9$ |
| Winograd List | $5/2p^2 + 2 \lceil \log_2(p) \rceil /p$ | $8p \lceil \log_2(p) \rceil + 10$ |

Table 3: Summary of communication costs as functions of $\beta$ and $\tau$.

Data–parallel implementations of Strassen and Winograd have almost the same cost model. Indeed the only difference is the number of additions (18 vs. 15). As the complexity of addition is in $O(M^2)$, this difference is negligible with regard to the rest of the cost model. For the same reason, we claim that the mixed–parallel implementations of Strassen and the list–based implementation of Winograd have comparable cost models. Concerning the phase–based implementation of Winograd, the redundant computation induces a reduction of the communication cost.

Figure 15 shows instantiations of some cost models with experimentally measured values for $\tau$, $\beta$ and sequential execution times of products and additions. These values correspond to a cluster of Pentium III connected by a Fast Ethernet network. We can see that a mixed–parallel implementation achieves better performance than data–parallel implementations when one has to multiply matrices distributed on disjointed grids. This gain can be greater than 45% on a cluster such as the one simulated in Figure 15.

# 6   Heterogeneous Version

Few work has been proposed for the implementation of numerical kernels on heterogeneous platforms. In [3, 4], the authors prove the NP-completeness of the data-distribution problem for the classical matrix-multiplication problem with different processors speed and present a polynomial column-based heuristic. The algorithms presented are very efficient but the distribution used is highly irregular and leads to high redistribution costs when using other kernels before (and after) the matrix-matrix product.
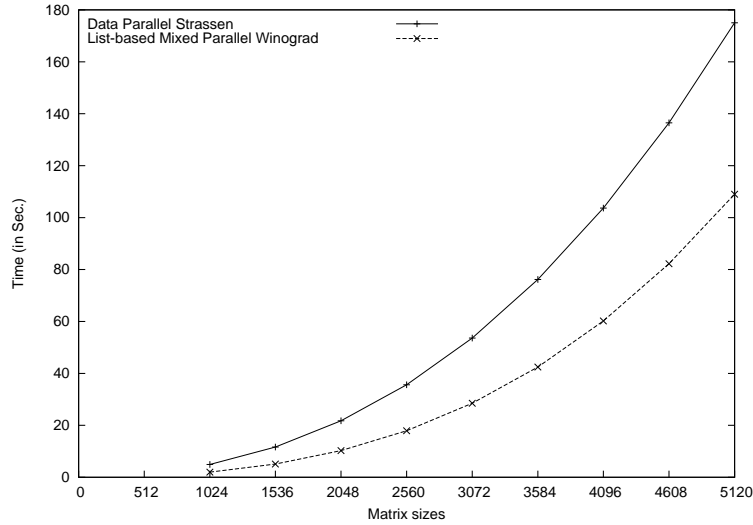
Figure 15: Theoretical performance of data–parallel Strassen, phase- and list–based mixed–parallel Winograd on a cluster of Pentium III connected by a Fast Ethernet network.

Self adapting libraries to variable loads have also been studied [20]. Parameters (block size and grid shape) are computed at run-time. However, the data distribution is still regular. This optimization improves the performance but it is not really tolerant to high differences between the processors speed.

To derive the heterogeneous algorithm from our homogeneous implementation of Strassen algorithm, we have to move the cut-line depending on the relative speed of processors in each context. Figure 16 shows the chain of products and the proposed cut-line determining the placement. As the matrix multiplication is the most time consuming task of Strassen's algorithm, an efficient algorithm has to evenly balance the matrix multiplication load between the two contexts. Figure 17 shows the sharing of tasks between contexts given by the list algorithm. We can see that each computation task executed on context 2 (right) corresponds, as much as possible, to two tasks of the same kind executed on context 1 (left).
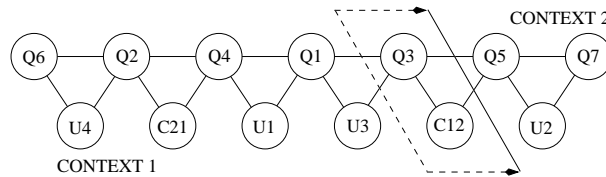


Figure 16: Strassen chain of products and cut-line for the heterogeneous algorithm.

**Input :** Matrix A
Temporary variables $R_1$, $R_2$, $R_3$, $R_4$, $R_5$, $R_6$, $R_7$

**Input :** Matrix B
Temporary variables $S_1$, $S_2$, $S_3$, $S_4$

| Store to: | | Computation | Algorithmic variables | | Store to: | | Computation | Algorithmic variables |
|---|---|---|---|---|---|---|---|---|
| | | | | Redistribute $R_1 \leftarrow B_{11}$ | | | | |
| $R_2$ | $\leftarrow$ | $A_{11} + A_{22}$ | $T_1$ | | $S_1$ | $\leftarrow$ | $B_{11} + B_{22}$ | $T_6$ |
| $R_3$ | $\leftarrow$ | $A_{21} + A_{22}$ | $T_2$ | | | | | |
| | | | | Redistribute $R_4 \leftarrow S_1$ | | | | |
| $R_5$ | $\leftarrow$ | $A_{11} + A_{12}$ | $T_3$ | | $S_1$ | $\leftarrow$ | $B_{12} - B_{22}$ | $T_7$ |
| $R_6$ | $\leftarrow$ | $A_{21} - A_{11}$ | $T_4$ | | | | | |
| | | | | Redistribute $R_5 \rightarrow S_2$ | | | | |
| | | | | Redistribute $R_5 \leftarrow S_1$ | | | | |
| $R_7$ | $\leftarrow$ | $R_2 * R_4$ | $Q_1$ | | $S_1$ | $\leftarrow$ | $S_2 * B_{22}$ | $Q_5$ |
| $R_2$ | $\leftarrow$ | $A_{11} * R_5$ | $Q_3$ | | | | | |
| $C_{22}$ | $\leftarrow$ | $R_7 + R_2$ | $U_3$ | | $S_2$ | $\leftarrow$ | $B_{21} - B_{11}$ | $T_8$ |
| $R_4$ | $\leftarrow$ | $A_{12} - A_{22}$ | $T_5$ | | | | | |
| | | | | Redistribute $R_4 \rightarrow S_3$ | | | | |
| | | | | Redistribute $R_4 \leftarrow S_2$ | | | | |
| | | | | | | $S_2$ | $\leftarrow$ | $B_{21} + B_{22}$ | $T_{10}$ |
| $R_5$ | $\leftarrow$ | $R_3 * R_1$ | $Q_2$ | | $S_4$ | $\leftarrow$ | $S_3 * S_2$ | $Q_7$ |
| $R_1$ | $\leftarrow$ | $A_{22} * R_4$ | $Q_4$ | | | | | |
| $C_{21}$ | $\leftarrow$ | $R_5 + R_1$ | $C_{21}$ | | $S_2$ | $\leftarrow$ | $B_{11} + B_{12}$ | $T_9$ |
| $C_{11}$ | $\leftarrow$ | $R_7 + R_1$ | $U_1$ | | | | | |
| | | | | Redistribute $R_1 \leftarrow S_2$ | | | | |
| $R_7$ | $\leftarrow$ | $R_6 * R_1$ | $Q_6$ | | $S_2$ | $\leftarrow$ | $S_4 - S_1$ | $-U_2$ |
| | | | | Redistribute $R_1 \leftarrow S_1$ | | | | |
| | | | | Redistribute $R_3 \leftarrow S_2$ | | | | |
| $C_{22}$ | $\leftarrow$ | $C_{22} - R_5$ | | | | | | |
| $C_{22}$ | $\leftarrow$ | $C_{22} + R_7$ | $C_{22}$ | | | | | |
| $C_{12}$ | $\leftarrow$ | $R_2 + R_1$ | $C_{12}$ | | | | | |
| $C_{11}$ | $\leftarrow$ | $C_{11} + R_3$ | $C_{11}$ | | | | | |

**Output :** $C = (C_{ij})$

Figure 17: Heterogeneous mixed implementation of Strassen algorithm for Context 1 (left) and 2 (right).

# 7 Experimental Results

To experimentally verify the impact of mixed–parallelism on parallel implementations of Strassen and Winograd algorithms, we ran tests on homogeneous and heterogenous platforms. The former is *icluster* which is a cluster of HP e–vectra nodes (Pentium III 733 MHz with 256 MB of memory per node) connected through a Fast Ethernet network via HP Procurve 4000 switches. The latter is built upon the connection of two homogeneous clusters by a Fast Ethernet link. The first cluster is composed of 4 Pentium Pro 200 MHz connected through an Ethernet network, while the second includes 4 Pentium II 450 MHz connected through a Fast Ethernet network. This implies a 2.25 speed ratio between the two contexts.

For all experiments, we used for communications a version of the BLACS library on top of MPI (LAM on *icluster* and MPICH on the heterogeneous platform). The PBLAS v1.0 library is used for inner matrix multiplications. The sequential BLAS kernel is the one generated by ATLAS. Here we compare performance of all of the algorithms presented in Section 5.

## 7.1 Homogeneous Implementations

We ran each implementation of Strassen and Winograd algorithms on 8 and 32 processors of *icluster*. Figures 18 shows the resulting executions times on 8 and 32 processors.

On 8 processors, the measured execution times match the theoretical times of Figure 15. This result corroborates our analysis. Phase–based implementations achieve slightly better performance than list–based implementations. This may be due to a better synchronization between contexts in the phase–based version.

We obtain the same kind of results on 32 processors. We can also see that using 4 times more processors, we can deal with 4 times bigger matrices. The scalability of our algorithms in term of memory consumption is thus verified. For comparison purpose we ran a pure ScaLAPACK code that aligns of $A$ and $B$, computes a `pdgemm` on all processors and then redistributes the result $C$. We can see that the execution time of this code is very close to those of our mixed–parallel implementation. But, as we will see in the next section, our algorithms can easily be adapted to run on heterogeneous platforms.

## 7.2 Heterogeneous Implementations

To validate our heterogeneous version of the list–based mixed–parallel Strassen algorithm, we compared it on two heterogeneous platforms. First, we simulated heterogeneity on *icluster*. To achieve this processors belonging to context 2 execute each computational task several time. The number of consecutive executions defines the "speed ratio" between contexts. It has to be noticed that the network is still homogeneous in this platform. Figure 19 (top) presents the results of this experiment. We can see that the gain increases as the speed ratio between the two contexts. Of course, we loose time when the entire cluster is homogeneous.
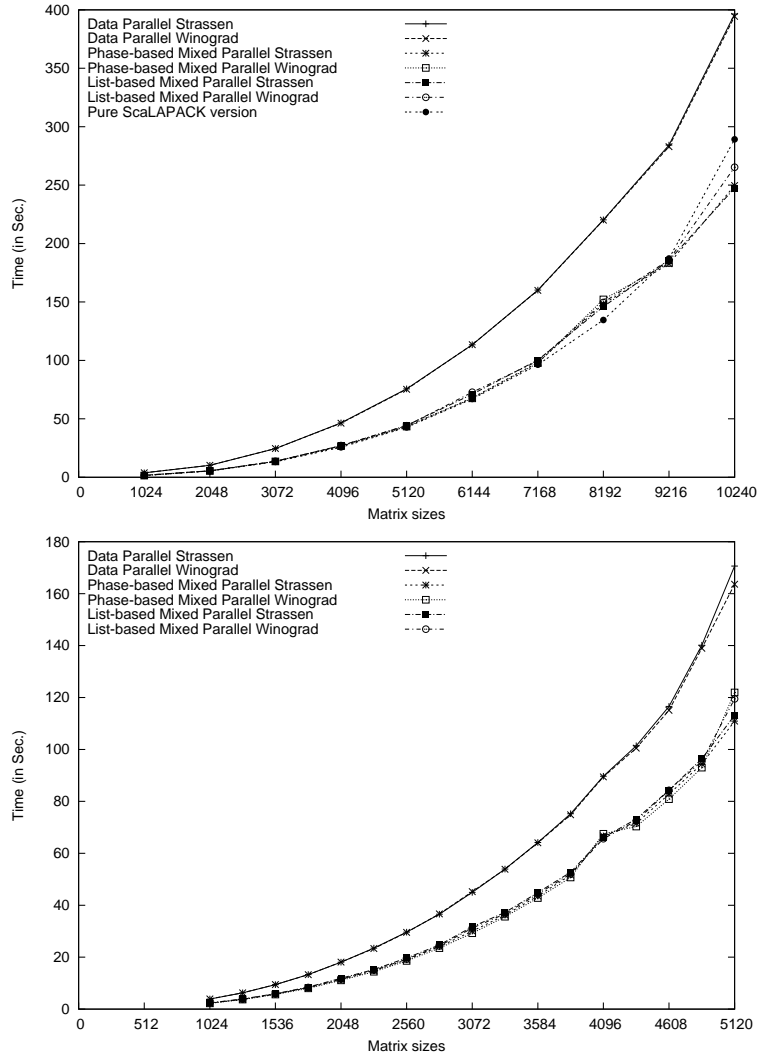
Figure 18: Comparison of the execution time of the different homogeneous implementations of the matrix product kernel presented in this paper on 8 (top) and 32 (bottom) processors of *icluster*.

Then we run our mixed–parallel implementations (homogeneous and heterogeneous) and a ScaLAPACK code that performs the alignment of $A$ and $B$, the multiplication and the redistribution of the result on our cluster of clusters. Figure 19 (bottom) shows that the
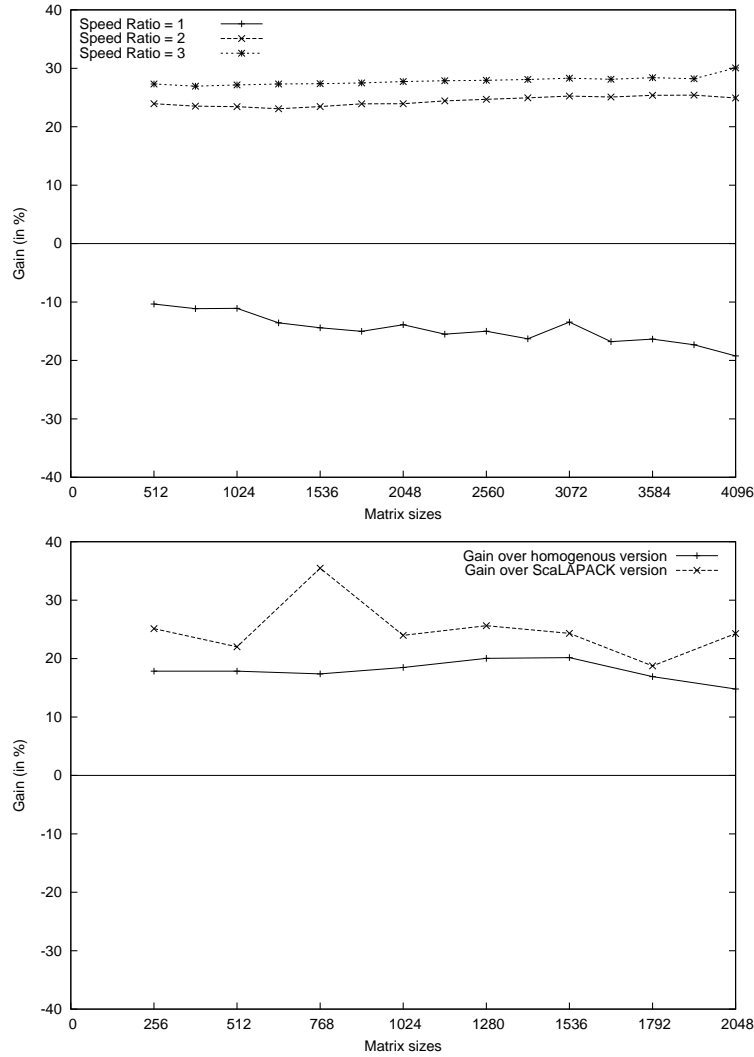
Figure 19: Gains of the heterogenous implementation over the homogeneous version on a simulated heterogeneous platform with different speed ratios (top). Gain obtained over ScaLAPACK and homogeneous version by the heterogenous version (bottom).

gain is clear between the homogeneous version of the mixed implementation of Strassen algorithm as well as the ScaLAPACK code and this even for small matrices.

# 8    A Word about Recursion

The $O(M^{2.807})$ asymptotic complexity of Strassen and Winograd algorithms can only be achieved if the decomposition is applied recursively. We can easily derive recursive versions of our mixed–parallel implementations. Indeed each of the seven inner products involves a matrix distributed on context 1 and a matrix distributed on context 2. The assumptions we made to build our algorithms are thus respected. We just have to write versions where the result is distributed on context 2. For instance such a list–based mixed–parallel implementation of Strassen algorithm can be derived of the sharing of products presented in Figure 20. Moreover the block size is the only parameter to adapt, as a distribution allowing the computation of additions without communications has to be kept until the most inner level. $R = M/2^{i}p$, where $i$ is the number of recursion steps is a size satisfying this condition.
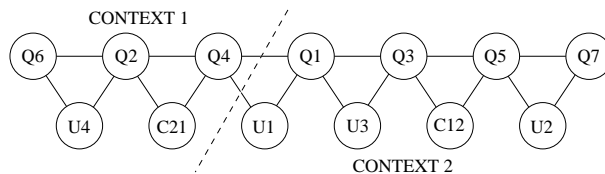


Figure 20: Strassen chain of products and cut-line for version where the result  on context 2.

But if a recursive mixed–parallel implementation of Strassen algorithm reduces the number of products, it introduces a non-negligible amount of communications. On a platform with fast processors and a slow network, such as *icluster*, the gain in terms of computation can not divide out the cost of this communication amount increase. Performance is thus worst than this of a non–recursive implementation. We plan to study whether recursion can improve performance on a platform with a high communication/computation ratio.

# 9    Conclusion and Future Works

In this paper, we have presented mixed–parallel implementations of Strassen and Winograd algorithms in the scope of client–server applications (i.e., the different data involved in computations are already distributed on disjoint grids). We chose these algorithms because they are composed of several data–independant tasks which are easier to schedule and place compared to a classical matrix multiplication. Our algorithms are simple and allow the use of high performance numerical kernels (e.g., ScaLAPACK). After giving details about the proposed mixed–parallel algorithms and their related issues, we gave theoretical models of these algorithms. We also showed how an implementation of the list–based version of Strassen targeting heterogenous platforms can be derived from the homogeneous implementation. We claim that Strassen algorithm is more interesting than the classical matrix-matrix product for an implementation on a heterogeneous platform. To validate our approach, we compared theoretical models and experimental results of our work with data–parallel implementations of Strassen and Winograd algorithms. Experiments corroborated our theoretical analysis.

We aim to develop recursive versions of our mixed–parallel implementations. Indeed the complexity of $O(M^{2.807})$ can only be reached if the Strassen decomposition is applied recursively. We also plan to study an other way to perform matrix multiplication on an heterogeneous platform using mixed–parallelism. The idea is to keep the algorithm designed for the homogeneous case given in section 4.2.2 and execute it on a platform such as the one used in section 7. But in this case the aggregate powers of the clusters will be quite equivalent, as more slow processors will be involved. Our future work finally concerns the automatization of the scheduling process and the implementation of other numerical kernels.

# References

[1] D. Bailey, K. Lee, and H. Simon. Using Strassen's Algorithm to Accelerate the Solution of Linear Systems. *Journal of Supercomputing*, 4(4):357–371, Jan 1991.

[2] H. Bal and M. Haines. Approaches for Integrating Task and Data Parallelism. *IEEE Concurrency*, 6(3):74–84, Jul-Sep 1998.

[3] O. Beaumont, V. Boudet, A. Legrand, F. Rastello, and Y. Robert. Heterogeneous matrix-matrix multiplication, or partitioning a square into rectangles: NP-completeness and approximation algorithms. In *EuroMicro Workshop on Parallel and Distributed Computing (EuroMicro'2001)*, pages 298–305. IEEE Computer Society Press, 2001.

[4] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert. Matrix multiplication on heterogeneous platforms. *IEEE TPDS*, 12(10):1033–1051, 2001.

[5] P. Bjorstad, F. Manne, T. Sorevik, and M. Vajtersic. Efficient Matrix Multiplication on SIMD Computers. 13(1):386–401, January 1992.

[6] L.S. 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. *ScaLA-PACK Users' Guide*. SIAM, 1997.

[7] E. Caron and F. Suter. Parallel Extension of a Dynamic Performance Forecasting Tool. In *proc. of the Int. Symposium on Parallel and Distributed Computing*, Iasi, July 2002.

[8] S. Chatterjee, A. Lebeck, P. Patnala, and M. Thottethodi. Recursive Array Layouts and Fast Parallel Matrix Multiplication. In *Proceedings of Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, Saint-Malo, France, June 1999.

[9] C. Chou, Y. Deng, and Y. Wang. A Massively Parallel Method for Matrix Multiplication Based on Strassen's Method. Technical Report SUNYSB-AMS-93-17, Center for Scientific Computing, The University of Stony Brook, November 1993.

[10] F. Desprez and F. Suter. Mixed Parallel Implementations of the Top Level Step of Strassen and Winograd Matrix Multiplication Algorithms. In *proc. of the 15th Int. Parallel and Distributed Processing Symposium (IPDPS'01)*, San Francisco, April 2001.

[11] B. Dumitrescu, J.L. Roch, and D. Trystram. Fast Matrix Multiplication Algorithms on MIMD Architecture. *Parallel Algorithms and Applications*, 4(2):53–70, 1994.

[12] P. Fischer and R. Probert. Efficient Procedures for Using Matrix Algorithms. In *Automata, Languages and Programming*, volume 14 of LNCS, pages 413–427. Springer-Verlag, Berlin, 1974.

[13] B. Grayson and R. Van de Geijn. A High Performance Parallel Strassen Implementation. *Parallel Processing Letters*, 6(1):3–12, 1996.

[14] N.J. Higham. Exploiting Fast Matrix Multiplication Within the Level 3 BLAS. Technical Report TR89-984, Dept of CS–Center of Applied Math.–Cornell Univ., April 1989.

[15] S. Huss-Lederman, E. Jacobson, J. Johnson, A. Tsao, and T. Turnbull. Strassen's Algorithm for Matrix Multiplication: Modeling, Analysis, and Implementation. Technical Report CCS-TR-96-147, Center for Computing Sciences, Argonne National Lab., 1996.

[16] 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, Umeå University, October 1995.

[17] L. Prylli and B. Tourancheau. Fast Runtime Block Cyclic Data Redistribution on Multiprocessors. *Journal of Parallel and Distributed Computing*, 45(1):63–72, Aug 1997.

[18] S. Ramaswany. *Simultaneous Exploitation of Task and Data Parallelism in Regular Scientific Applications*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[19] T. Rauber and G. Rünger. Scheduling of Data Parallel Modules for Scientific Computing. In SIAM, editor, *Proceedings of Ninth SIAM Conference on Parallel Processing for Scientific Computing (PP99)*, San Antonio, Texas, March 1999.

[20] K. J. Roche and J. J. Dongarra. Deploying Parallel Numerical Library Routines to Cluster Computing in a Self Adapting Fashion, April 2002. Submitted to Parallel Computing, `http://www.netlib.org/netlib/utk/people/JackDongarra/PAPERS/dyn_pnumlib%s.pdf`.

[21] W. Rönsch and H. Strauß. The Level 3 BLAS Forms of Parallel Factorization Methods. In D.J. Evans, G.R. Joubert, and F.J. Peters, editors, *Parallel Computing 89*, pages 85–92. Elsevier Science Publisher B.V., 1989.

[22] V. Strassen. Gaussian Elimination Is Not Optimal. *Numerische Mathematik*, 14(3):354–356, 1969.

[23] M. Thottethodi, S. Chatterjee, and A. Lebeck. Tuning Strassen's Matrix Multiplication for Memory Efficiency. In *Proceedings of Supercomputing'98*, Orlando, Nov 1998.