One-Step Algorithm for Mixed Data and Task Parallel Scheduling Without Data Replication

Vincent Boudet, Frédéric Desprez, Frédéric Suter

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 5668

One-Step Algorithm for Mixed Data and Task Parallel Scheduling Without Data Replication

V. Boudet,
F. Desprez,
F. Suter

October 2002

Research Report N° 2002-34

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr

One-Step Algorithm for Mixed Data and Task Parallel Scheduling Without Data Replication

V. Boudet, F. Desprez, F. Suter

October 2002

Abstract

In this paper we propose an original algorithm for mixed data and task parallel scheduling. The main specificities of this algorithm are to simultaneously perform the allocation and scheduling processes, and avoid the data replication. The idea is to base the scheduling on an accurate evaluation of each task of the application depending on the processor grid. Then no assumption is made with regard to the homogeneity of the execution platform. The complexity of our algorithm are given. Performance achieved by our schedules both in homogeneous and heterogeneous worlds, are compared to data-parallel executions for two applications: the complex matrix multiplication and the Strassen decomposition.

Keywords: Mixed–parallelism, Scheduling, Data Replication

Résumé

Dans cet article, nous proposons un algorithme original d'ordonnancement mixte. Les principales spécificités de cet algorithme sont d'effectuer simultanément l'allocation et l'ordonnancement, et d'interdire la réplication de données. L'idée est de baser l'ordonnancement sur une évaluation précise de chacune des tâches de l'application en fonction de la grille de processeurs utilisée. Aucune hypothèse concernant l'homogénéité de la plate-forme n'est donc effectuée. La complexité de notre algorithme est donnée. Les performances obtenues par nos ordonnancements, aussi bien en homogène qu'en hétérogène, sont comparées à des exécutions utilisant le parallélisme de données pour deux applications : la multiplication de matrices complexes et la décomposition de Strassen.

Mots-clés: Parallélisme mixte, Ordonnancement, Réplication de données

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.

1.1 Related Work

Most of the researches about mixed–parallelism have been done in the area of programming languages. Bal and Haines present in [1] a survey of several languages aiming at integrating both forms of parallelism. To perform such an integration, two approaches are possible. The former introduces control in a data–parallel language [6, 7, 13] while the latter add data–parallel structures in task–parallel languages [4, 14].

One way to obtain a simultaneous exploitation of data– and task–parallelism consists in considering task graphs whose nodes can be data–parallel routines and then find a schedule that minimizes the total completion time of the application.

In [11] Ramaswamy introduces a structure to describe mixed–parallel programs: the Macro Dataflow Graph (MDG). It is a direct acyclic graph where nodes are sequential or data–parallel computations and edges represent precedence constraints, with two distinguished nodes, one preceding and one succeeding all other nodes. MDGs can be extracted form codes written in C, Fortran or even Matlab. A node is weighted by the computation cost of the task, estimated using Amdahl's law. This execution time also includes data transfer latencies which depend on source and destination data distributions. An edge between two nodes is weighted by communication times, *i.e.,* the total amount of data to transfer divided by the bandwidth of the network. The scheduling algorithm proposed by Ramaswamy is based convex programming, allowed by posynomial properties of chosen cost models, and some properties of the MDG structure. The *critical path* is defined as

the longest path in the graph while the *average area* which provides a measure of the processor-time area required by the MDG. The first step of the TSAS (Two Step Allocation and Scheduling) algorithm aims at minimizing the completion time using these two metrics. An allocation is determined for each task during this step. A list scheduling algorithm is used in the second step to schedule the allocated tasks to processors.

Rădulescu *et al.* have also proposed two-step mixed–parallel scheduling algorithms: CPA [9] and CPR [10]. Both are based on the reduction of application critical path. The main difference between these algorithms is that the allocation process is totally decoupled of the scheduling in CPA. In the allocation step, both algorithms aims at determining the most appropriate number of processors for the execution of each task. The technique used is first to allocate one processor per task and then add processors one by one until the execution time increases or the number of available processors is reached. Once again estimations of computation costs are based on Amdahl's law. The second step schedules the tasks using a list scheduling algorithm.

Rauber and Rünger [12] limit their study to graphs built by serial and/or parallel compositions. In the former case, a sequence of operations with data dependencies is allocated on the wole set of processors and then executed sequentially. In the latter the set of processors is divided in an optimal number of subsets. This number is determined by a greedy algorithm and the optimality criterion is the reduction of completion time. The computation costs of the parallel routines are estimated by runtime formulas depending on communication costs and sequential execution times.

1.2 Our Contributions

The specificity of our algorithm is to base the scheduling on an accurate evaluation of each task of the application. Thanks to a tool named FAST [8] and its parallel extension [5], we are able to determine the size and the shape of the best execution platform for a given routine. Moreover it is quite easy with this tool to estimate the cost of a redistribution between two processor grids by combining an analysis of the communication pattern to information about current network status.

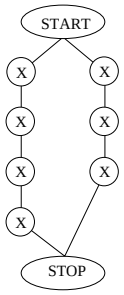
We aim at improving some aspects of mixed data and task parallel scheduling algorithms presented before. First these algorithms dissociate allocation and scheduling. Such a separation may lead to not detect a possible concurrent execution if less processors are allocated to some tasks. In section 1.3 we give an example motivating the simultaneous execution of allocation and scheduling processes.

To limit the multiple data copies, and thus the memory consumption

induced by the use of mixed parallelism, we have forbidden data replication in the conception of our algorithm. This constraint is not taken into account in the previous algorithms. In the next section, we justify this policy by presenting on a simple example, the gain in terms of execution time coming from mixed parallelism and the evolution of memory consumption depending on the chosen solution. Then we detail our algorithm in section 3. Finally, in section 4 we present schedules produced by our algorithm for complex matrix multiplication and the Strassen decomposition.

1.3 Motivating Examples

To show why it is mandatory to not separate allocation an scheduling processes, we have considered the task graph presented by Figure 1 (left), in which all tasks are dense matrix products involving matrices of same size. The platform where the tasks have to be allocated is composed of 13 processors. The data-parallel execution of a matrix multiplication on such a number of processors achieves poor performance. A schedule based on mixed parallelism may be efficient in such a case. In that example, we considered three sub-grids, respectively composed of 4, 9, and 12 processors, in addition to the whole grid. Execution times achieved on each of these configurations are given in Figure 1 (right).



Configuration	Time
2×2	$11,56$
3×3	$8,44$
3×4	$6,84$
1×13	$16,70$

Figure 1: Example of task graph (left) and execution times of each of these tasks on different processor grids (right).

When allocation and scheduling are dissociated, the first step aims at allocating each task on the grid which achieves the best performance, *i.e.,* the 3×4 grid. This solution implies that tasks will be serially scheduled in the second step. In such a case the completion time will be 48 seconds.

But if allocation and scheduling are performed simultaneously, associating tasks to estimations of execution time, it is possible to find a solution allocating less processors to some tasks to perform some others in parallel.

Our algorithm is thus able to produce a schedule whose completion time will be $32,5$ seconds. Then we obtain a gain of 30% with regard to the previous solution.

To find this mixed parallel schedule, the studied algorithms use a minimization function, typically of the critical path, in their allocation step. Several iterations are then needed to reach the optimal solution.

To justify our choice to forbid data replication, let us study the application of mixed parallelism to the following complex matrix multiplication operation

$$C = \begin{cases} C_r = A_r \times B_r - A_i \times B_i \\ C_i = A_r \times B_i + A_i \times B_r \end{cases}$$

Let us assume we dispose of two processor grids of size p for this experiment. We also assume that the input data of that problem are distributed as follows: A_r and A_i are aligned on the first p^2 processors while B_r and B_i are distributed on the last p^2 processors. Finally we add an additional constraint imposing on C_r and C_i to be distributed on the same grid as A at the end of the computation. Figure 2 shows completion times achieved using respectively data-parallelism (a), mixed-parallelism with data replication (b) and mixed-parallelism without replication (c).

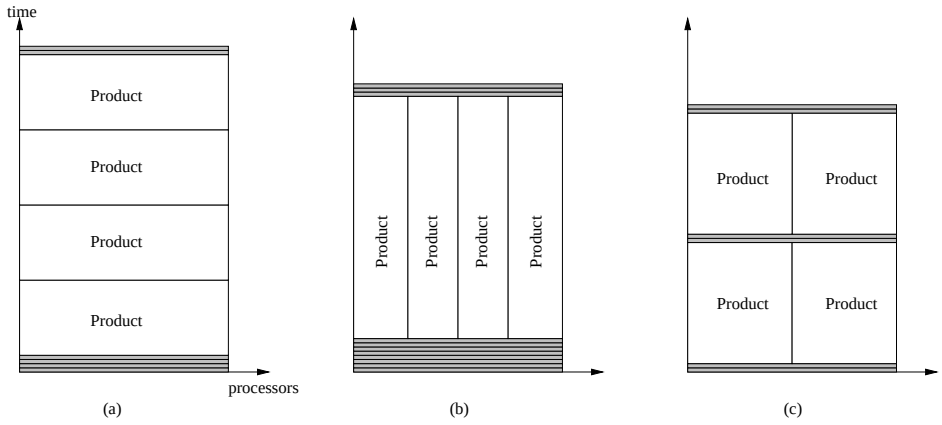


Figure 2: Comparison between data-parallel, mixed-parallel with data replication and mixed-parallel without data replication scheduling for the complex matrix multiplication.

On this figure shaded parts represent data redistributions between processor grids. Both addition and subtraction operations are not reported on that figure as their execution times are negligible with regard to the rest of the application. Moreover we can see that, when data have already a source distribution, replication increases the number of redistributions performed.

We compared the memory space needed by each of the studied solutions. If each matrix involved is of size N, the data-parallel version and our version

without replication will both need $3N^2/4$ temporary elements per processor, while the mixed-parallel version with replication will need $3N^2/2$ elements. When data have already a distribution, we can thus claim that using mixed parallelism without replication allows to reduce execution time and not increase the memory consumption in a too large proportion.

2 Task Graph Model

Our one-step algorithm is based on a task graph structure close to the MDG structure proposed by Ramaswamy [11]. An application is then described by direct acyclic graph $G = (V, E)$ where nodes represent potentially parallel tasks and edges shows dependencies between tasks. Two particular nodes are added to this structure. The `START` node precedes all other nodes. Tasks having this node for father involve initial data of the application. The `STOP` node succeeds all other nodes. Tasks having this node for son involve terminal data of the application.

Initial and terminal data of the application have fixed distributions the schedule has to respect. For instance if an application uses two matrices, the former distributed on one half of the available processors while the latter is distributed on the other half, a data-parallel schedule will have to include the redistribution costs needed to align those matrices on the whole set of processors. The same rule applies for the results produced by the application.

As considered tasks correspond to ScaLAPACK routines, each of them implies at most three input data. On the other hand the number of sons of a task is not limited as a data can be used by several other tasks.

As mentioned before, our algorithm relies on information relative to each task of the application which are integrated in our task graph structure as follows. First a redistribution cost table is associated to each edge. We also associate to each node a *startup time*, the location of task input data and a list of couples {configuration; execution time}. Each of these configurations is defined as a tuple $\{p, q, mb, nb, list\}$, where p and q are respectively the numbers of rows and columns of the processor grid employed; mb and nb are the dimensions of a distribution block, as ScaLAPACK routines use block-cyclic distributions on a virtual grid of processors; finally the last field of this tuple is the *list* of the processors composing the grid (See C_1, C_2 and C_3 in Figure 3). We choose to use a list instead of coordinates of a distinguished processor to be able to handle non contiguous grids as shown in Figure 3. However we assume that contention is negligible.

Theses configurations allow us to avoid any strong constraint about the homogeneity of the whole platform. Indeed, to achieve optimal performance

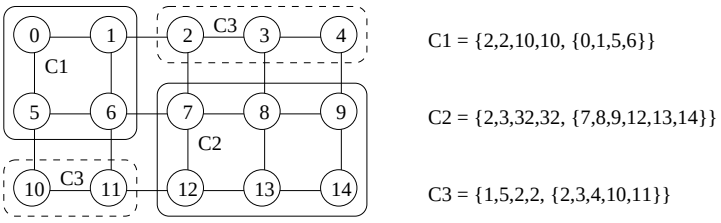


Figure 3: Configurations and description samples associated.

in mixed-parallel execution, we only need homogeneity inside configurations. Moreover if routines targeting heterogeneous platform exist, there is no more assumption made about homogeneity. Finally we can take into account the impact of the grid shape on performance using such configurations.

3 Description of the One-Step Algorithm

Our goal is to determine for each task whether it is better to compute it on the whole set of processors in a data-parallel way or assign a subset of the processors to this task. In the latter case some processors are available for another task. We first describe the algorithm which determines the decision to take when we are confronted to a set of tasks independent of another one. Next we extend this algorithm to the case of a task graph representing a whole application.

3.1 Mixed-Parallel Scheduling of Independent Tasks

Let us consider a task T_1 and a set of tasks $\mathcal{I} = \{T_2, \cdots, T_n\}$ which are independent of T_1. Each task T_i in $T_1 \cup \mathcal{I}$ is associated to a subset of $\mathcal{C}_i = \{C_1, \cdots, C_m\}$, the set of possible configurations. The computation time of a task T_i on a configuration C_j is denoted as $t_{i,j}$. This time does not include the possible communications needed to transfer the input data to the processors of the chosen configuration.

The main idea to determine if we assign T_1 either to the whole set of processors or one of its configurations associated is the following. We have to compute t_{mixed} and $t_{//}$ which are respectively the mixed- and the data-parallel execution times of a set of tasks. Assume that we assign T_1 to a given configuration C_i. The time when all the processors of this configuration are ready to receive the needed data is denoted as $EST(C_i)$[1]. For each data $D_i \in \mathcal{D}$, where $\mathcal{D}$ is the set of data needed to execute T_1, C_{D_i} represents

[1]Earlier Startup Time

the configuration where this data is distributed and $R(D_i)$ is the time to transfer D_i to C_i. The **Update** algorithm 1 is then used to update the EST of each configuration involved in this redistribution. This guarantees that a processor can not begin to compute before it finished to participate to the redistribution of the data it owns.

Algorithm 1 EST update algorithm.

Update $(C, \mathcal{D})$
  $t \longleftarrow EST(C)$
  For $i = 1$ to $\|\mathcal{D}\|$
    $m \longleftarrow \max(t, EST(C_{D_i}))$
    $t = m + R(D_i)$
    For each C_i in which at least one processor owns a part of D_i
      $EST(C_j) = m + R(D_i)$

When trying to schedule another tasks while executing T_1 using mixed-parallelism, we have to consider the subset $\mathcal{U}$ constituted of couples {task; configuration} where each task is independent of T_1 and each configuration involves only processors of $\mathcal{P} \backslash C_i$. It has to be noticed that a same task can appear several times in $\mathcal{U}$ if more than one of its configurations associated involve no processor of C_i. Elements of this set are sorted with regard to the amount of redistribution they induce due to the mapping of T_1 on C_i. Moreover tasks which produce a terminal data are not included in $\mathcal{U}$. If this choice prevents us to execute more tasks in parallel of T_1, it also simplifies our algorithm.

Let us consider one of these couples $\{T_j, C_k\}$. Task T_j is supposed to start at $EST(C_k)$. However it may happen that the communication needed to execute T_j on C_k involves some processors of C_i. To prevent the serialization of these tasks, we have to perform this communication before starting the execution of T_1. In such a case the $ESTs$ of C_i and each configuration for which at least one processor owns a part of data to transfer have to be increased as Figure 4 shows. The **Update** algorithm 1 is called again.

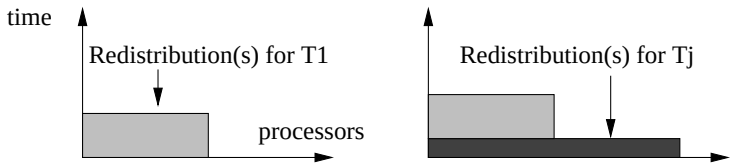


Figure 4: Increase of $EST(C_i)$ due to the redistribution induced by T_j.

Intuitively the condition given in equation 1 ensures that the mapping

of T_j on C_k, and the redistribution induced, do not increase the time of any configuration such that it exceeds the execution time of T_1 on C_i.

$$\max_k \left(EST(C_k) + \sum_j t_{j,k} \right) \leq EST(C_i) + t_{1,i}. \quad (1)$$

If this condition is preserved, then T_j is scheduled using mixed-parallelism and the corresponding couple is added to $\mathcal{S}$, the set of already scheduled tasks. Moreover $\mathcal{U}$ is updated. Each couple involving T_j is removed from the set and if the scheduling of T_j produces ready tasks which are independent of T_1, then the corresponding couples are added to $\mathcal{U}$.

On the other hand, if the condition is no more true, the update of ESTs is canceled, the couple $\{T_j, C_k\}$ is removed from $\mathcal{U}$ and the algorithm considers the next candidate.

Once all couples in $\mathcal{U}$ have been considered, $t_{mixed} = EST(C_i) + t_{1,i}$ is the completion time of already scheduled tasks. It may happen that the only task to schedule is T_1. In such a case the proposed mixed-parallel solution will be to assign T_1 on C_i and leave the other configurations idle.

After reinitializing the EST of the whole set of processors, ignoring the previous updates, it is possible to compute the completion time corresponding to the data-parallel execution of the same tasks , *i.e.,* $t_{//} = \sum_i (t_{i,\mathcal{P}} +$ redistribution cost associated to $T_i)$ where data needed by each T_i are respectively distributed on C_i. If t_{mixed} is less than $t_{//}$ then the mixed-parallel schedule is better than the data-parallel one. The algorithm then returns $\mathcal{S}$, *i.e.,* T_1 will be mapped on C_i and each T_j will be assigned on the corresponding C_k. Otherwise we try to assign T_1 to another configuration. If none of the configurations available for T_1 produces a better schedule than a data-parallel execution, T_1 is allocated on the whole set of processors.

The **Decision** algorithm 2 allows us to decide whether or not it is possible to perform other tasks in a mixed-parallel way during the execution of a given task T_1. Let us denote by c the maximum number of configurations associated to a task. In the worst case, a mixed-parallel schedule will never be better than the data-parallel solution. So all the configurations associated to T_1 will be tested and the "for" loop will be executed c_1 times. In a given iteration, each couple is only considered once. The maximum number of these couples is $\|I\| \times c$. As the number of data needed for a task is less than 3, the complexity of the update algorithm is in $\mathcal{O}(c)$. Indeed each configuration has only to be checked once. For each couple in $\mathcal{U}$, we call the **Update** routine at most twice, to compute the mixed- and the data-parallel solutions. Finally, the complexity of the **Decision** algorithm is $\mathcal{O}(c \times (\|I\| \times c) \times c) = \mathcal{O}(c^3 \|I\|)$. Practically, the maximum number of configurations c will be often less than

4 or 5, so the complexity of this algorithm is linear in the number of tasks.

Algorithm 2 Mixed-parallel execution decision algorithm.

Decision$(T_1, \mathcal{I})$
  For each C_i associated to T_1
    $t_{//} = EST(\mathcal{P})$
    $\mathcal{S} \longleftarrow \{T_1, C_i\}$
    Update $(C_i, \mathcal{D})$
    Construction of $\mathcal{U}$
    While $\mathcal{U} \neq \emptyset$
      Let $\{T_j, C_k\}$ be the first couple of $\mathcal{U}$
      Update $(C_k, \mathcal{D})$
      If $\max_k \left(EST(C_k) + \sum_j t_{j,k}\right) \leq EST(C_i) + t_{1,i}$
        Add $\{T_j, C_k\}$ in $\mathcal{S}$ and update $\mathcal{U}$
      Else
        Cancel the EST update
        Remove $\{T_j, C_k\}$ from $\mathcal{U}$
    $t_{mixed} = EST(C_i) + t_{1,i}$
    $EST(\mathcal{P}) = t_{//}$
    For each $T_i \in \mathcal{S}$
      Update $(\mathcal{P}, \mathcal{D})$
      $t_{//} = t_{//} + t_{i,\mathcal{P}}$ + Time to redistribute terminal data
      $EST(\mathcal{P}) = t_{//}$
    If $t_{mixed} \leq t_{//}$
      Return $\mathcal{S}$
  $\mathcal{S} \longleftarrow \{T_1, \mathcal{P}\}$
  Return $(\mathcal{S})$

3.2 Algorithm

Now, for a set of tasks $\mathcal{I}$ independent of an other task T_1, we are able to decide if it is better to compute T_1 in a data-parallel way or execute T_1 in parallel of a subset of $\mathcal{I}$.

Let us consider the task graph extracted from the application. At each time, there exists a possibly large number of ready tasks. The idea of the algorithm is to select a task and consider the set of tasks independent of it. We apply the **Decision** algorithm 2 on these inputs. The criterion to sort the list of ready task is the *bottom level* defined as the longest path from a task

to the STOP node. To determine the cost of this path, we use a data-parallel configuration to compute the execution time of each task belonging to it.

Let $\mathcal{T}$ be the sorted list of ready tasks and T_1 the first task of this list, *i.e.,* the task with the highest bottom level. The remaining tasks are flow-independent of T_1 but not necessary data-independent because they may use a common input data. As data are not replicated, if two tasks share an input, they can not be computed in parallel. So we have to compute the subset of ready tasks that are independent of T_1. For each task T_i in $\mathcal{T}$, if this task shares a predecessor with T_1 then T_i is not independent of T_1.

Our one-step mixed data- and task-parallel scheduling algorithm, presented in **Scheduling** algorithm 3, first builds the list of entry tasks, *i.e.,* those having the START node as predecessor, and sorts it with regard to their bottom level. The algorithm then determines the set of tasks independent of the first task of the list and decides if some mixed-parallelism can be found using **Decision** algorithm 2. If it is not the cases, this task is assigned to the whole set of processors and the algorithm considers the next tasks until the list of ready tasks becomes empty.

Algorithm 3 One-Step Mixed Parallel Scheduling Algorithm.

Scheduling $(\mathcal{P}, G)$
- Compute the bottom level of each task
- $\mathcal{T} \longleftarrow$ Sorted set of entry tasks
- While $\mathcal{T} \neq \emptyset$
  - $T_1 \longleftarrow$ first task of $\mathcal{T}$
  - $\mathcal{T} = \mathcal{T} \backslash T_1$
  - $\mathcal{I} \longleftarrow$ set of independent tasks
  - $(C, T_f) \longleftarrow$ **Decision**$(T_1, \mathcal{R})$
  - For each task $T_i \in \mathcal{S}$
    - Assign T_i to C_i
    - Update $\mathcal{T}$

In the worst case, the One-Step Mixed Parallel scheduling algorithm produces a data-parallel schedule, *i.e.,* only one task is assigned at each call to the **Decision** algorithm. So this algorithm is called $\|V\|$ times. The worst case complexity is then $\mathcal{O}(c^3\|V\|^2)$.

4 Experimental Validation

To compare the schedules produced by our algorithm to data-parallel executions we studied two examples : the complex matrix multiplication (CMM)

and the Strassen decomposition. The task graphs corresponding to these applications are presented by Figure 5.

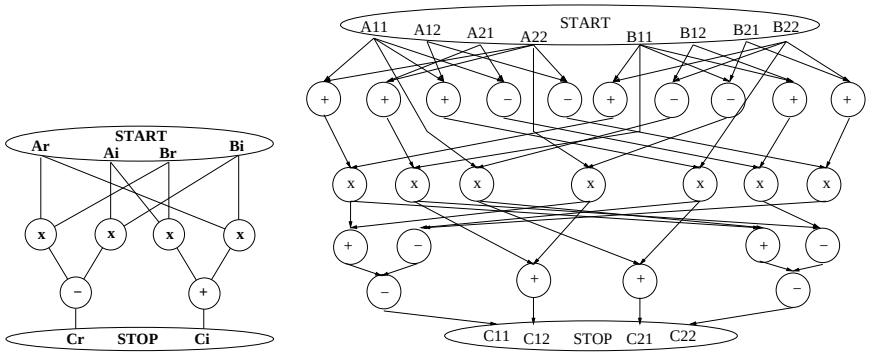


Figure 5: Task graphs for complex matrix multiplication (left) and Strassen decomposition (right).

Experimental conditions are similar for both applications. Matrix A is distributed on a 2×2 processor grid (C_1) while B is distributed on a grid of same size but totally disjointed (C_2). A data-parallel product will then be performed on a 2×4 grid ($\mathcal{P}$). Finally, the result C has to be distributed on the same grid as A.

We first scheduled these applications on an homogeneous platform. Then we aimed at validating the behavior of our algorithm when the platform is heterogeneous. The two following sections present some results for these two kinds of platforms.

4.1 Homogeneous Mixed-Parallel Scheduling

Table 1 presents the execution times of the operations that compose the CMM and Strassen task graphs. Each matrix involved in these computations is of size 2048×2048. Table 2 presents transfer times of a 2048×2048 matrix between the considered configurations.

	2×2	2×4
Product	$23,59$	$14,13$
Addition	$0,11$	$0,05$

Table 1: Execution times of the operations used in CMM and Strassen.

	C_1	C_2	$\mathcal{P}$
C_1	0	$1,18$	$0,75$
C_2	$1,18$	0	$0,75$
$\mathcal{P}$	$0,75$	$0,75$	0

Table 2: Redistribution cost a matrix between the configurations.

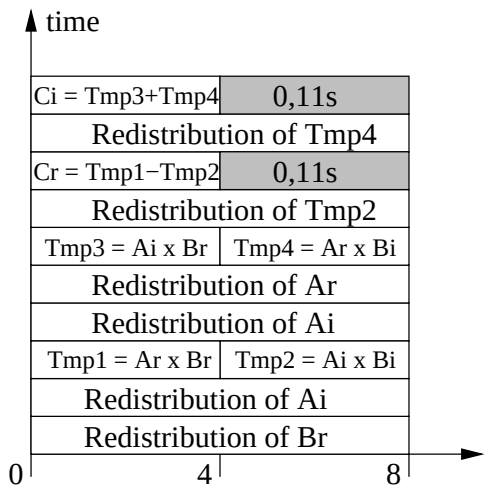


Figure 6: Mixed-parallel schedule for the CMM.

Figure 6 shows the schedule produced by our algorithm for the CMM. For readability reasons, we do not have respected the scale between tasks. On the other hand the shaded parts represent the idle time of a configuration. We can see that products are performed by pair, while addition and subtraction are computed serially on C_1. This assignment comes from the fact that these two tasks produce terminal data. Furthermore this solution is less expensive than aligning matrices on $\mathcal{P}$, performing the operation and then redistributing the result.

Figure 7 shows the schedule produced by our algorithm for the strassen decomposition. We can see that additions are mapped close to the data they involve. The first six products are computed pairwise, using mixed-parallelism. The seventh and last product is executed on the whole set of processors, as there does not remain any "long" task to schedule in parallel of it. Mixed-parallelism also appears in the execution of the additions needed to compute C_{11} and C_{22}. Finally the four last additions which produce terminal data are performed on C_1.

	1024	2048	3072	4096
CMM	14	9	-	8.3
Strassen	23.7	15	17	-

Table 3: Gains of mixed-parallel schedules over data-parallel ones.

Table 3 presents the gain achieved by schedules produced by our algorithm over data-parallel schedules. The decrease can be explained by the evolution

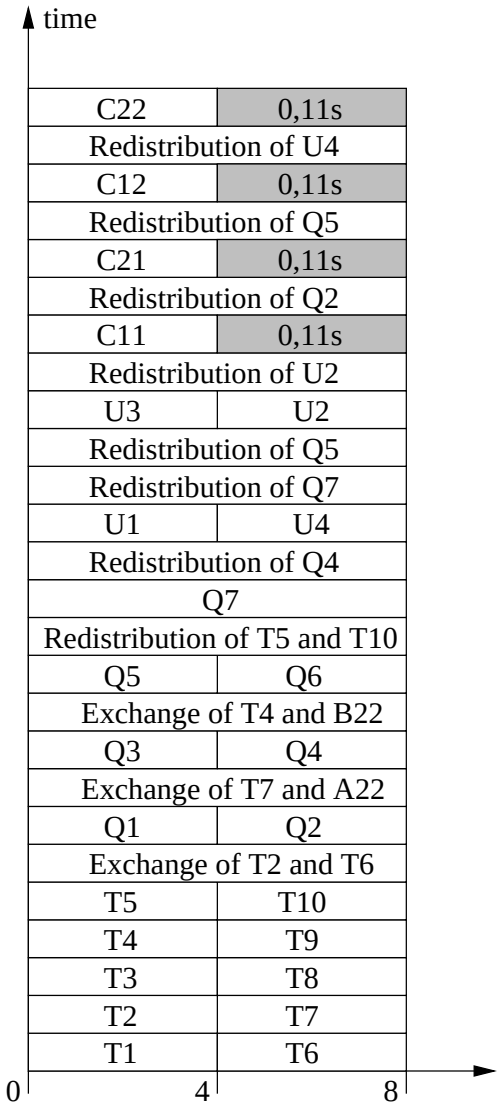


Figure 7: Mixed-parallel schedule for Strassen decomposition.

of the execution time of a product respectively on 4 and 8 processors. When matrices get large, the ratio between these times tends towards two. The time needed to compute two products using mixed-parallelism thus tends towards the time to serially execute the same operations using data-parallelism.

4.2 Heterogeneous Mixed-Parallel Scheduling

As mentioned before, no assumption has been made about the homogeneity of the execution platform during the design of our algorithm. We scheduled the Strassen decomposition on a heterogeneous platform composed of two homogeneous clusters of different processing speeds connected through a Fast Ethernet link. Denote C_1 as the "slow" configuration, C_2 as the "fast" configuration and $\mathcal{P}$ as the whole set of processors. Matrices A and B are respectively distributed on C_1 and C_2.

Table 4 presents the execution times of the operations composing the Strassen task graph. Each matrix involved in these computations is of size 1024×1024. Table 5 presents transfer times of a 1024×1024 matrix between the considered configurations.

	C_1	C_2	$\mathcal{P}$
Produit	$25,1$	$5,7$	$23,1$
Addition	$0,06$	$0,04$	$0,02$

Table 4: Execution times of the operations used in Strassen.

	C_1	C_2	$\mathcal{P}$
C_1	0	$0,35$	$0,22$
C_2	$0,35$	0	$0,22$
$\mathcal{P}$	$0,22$	$0,22$	0

Table 5: Redistribution cost a matrix between the configurations.

We can see that a data-parallel schedule will achieve poor performance on such a platform. Indeed the time to compute a product on the whole set of processors is more than four times bigger than the execution time achieved on the fast configuration.

Figure 8 shows the mixed-parallel schedule produced for the execution of the Strassen decomposition on an heterogeneous platform. If this schedule is far optimal – processors of C_2 are idle more than 22 seconds – the gain obtained over a data-parallel schedule is very good. Indeed the completion time of our schedule is around 56 seconds while the data-parallel schedule finishes in 165 seconds. So we have a gain of 66%. Furthermore, none of the existing algorithms have been designed to schedules an application targeting heterogenous platforms.

Concerning the specific matrix-matrix multiplication problem, a few work has been proposed for the implementation of this numerical kernel on hetero-

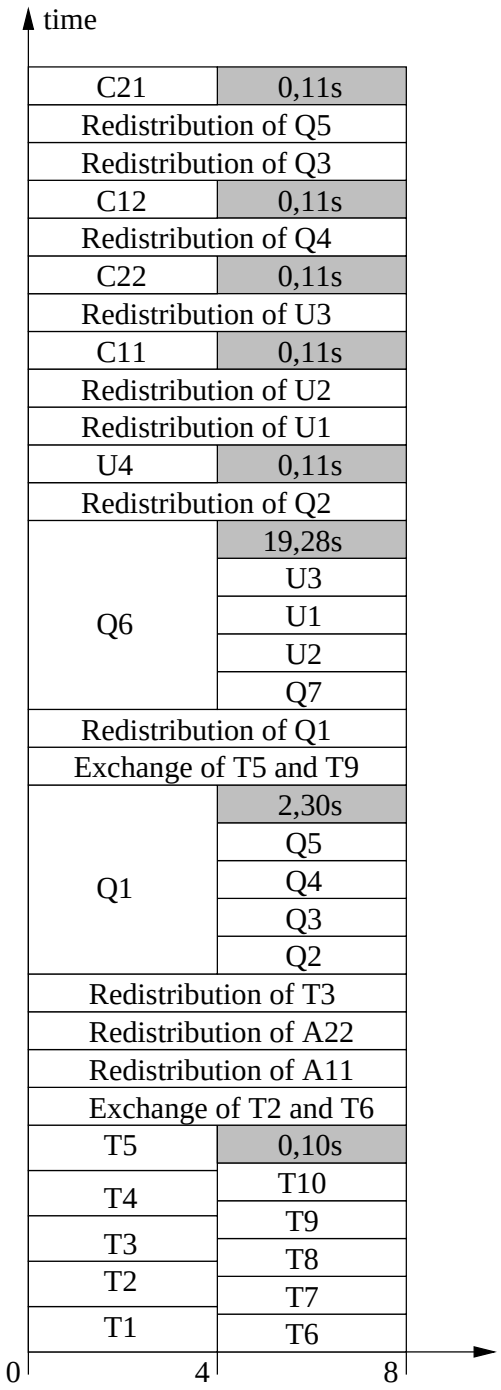


Figure 8: Mixed-parallel schedule for Strassen decomposition on an heterogeneous platform.

geneous platforms. In [2, 3], the authors prove the NP-completeness of the data-distribution 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.

5 Conclusion

In this paper, we have proposed an original algorithm using to schedule applications mixed-parallelism when data can not be replicated. These applications are represented by task graphs and are composed of dense linear algebra operations. The principle of this algorithm is to associate a list of execution platforms to each node of the task graph. For each of these configurations, we are able to estimate the execution time of the associate task. We can then simultaneously perform the allocation and scheduling processes. We applied this algorithm to two applications : the complex matrix multiplication and the Strassen decomposition. The produced schedules obtain completion times 15% better than those achieved by data-parallel schedules. On an heterogeneous platform this gain even raises up to 66%.

We plan to improve our algorithm to reduces idle times, by studying more cases. In its current version, the algorithms stops at the first mixed-parallel version which is better than the data-parallel one. But this solution may not be optimal. However if we increase the search space, we also increase the complexity of the algorithm. We also aim at better handling tasks which produce terminal data. Indeed this kind of tasks can be scheduled using mixed-parallelism only if the redistribution of the result is handled wisely, *i.e.,* delayed until all processors are available. But this delay should not prevent us to map other on this configuration during the idle time induced.

References

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

[2] Olivier Beaumont, Vincent Boudet, Arnaud Legrand, Fabrice Rastello, and Yves 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.

[3] Olivier Beaumont, Vincent Boudet, Fabrice Rastello, and Yves Robert. Matrix Multiplication on Heterogeneous Platforms. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1033–1051, 2001.

[4] Saniya Ben Hassen and Henri Bal. Integrating Task and Data Parallelism Using Shared Objects. In *Proceedings of the 10th Conference on Supercomputing*, pages 317–324. ACM Press, 1996.

[5] Eddy Caron and Frédéric Suter. Parallel Extension of a Dynamic Performance Forecasting Tool. In *Proceedings of the International Symposium on Parallel and Distributed Computing*, Iaşi, Romania, July 2002.

[6] Barbara Chapman, Matthew Haines, Piyush Mehrotra, Hans Zima, and John Van Rosendale. Opus: A Coordination Langage for Multidisplinary Applications. *Scientific Programming*, 6(2):345–362, 1997.

[7] Ian Foster and K. Mani Chandy. FORTRAN M : A Language for Modular Parallel Programming. *Journal of Parallel and Distributed Computing*, 26(1):24–35, 1995.

[8] Martin Quinson. Dynamic Performance Forecasting for Network-Enabled Servers in a Metacomputing Environment. In *Proceedings of the International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS'02)*, Fort Lauderdale, April 2002.

[9] Andrei Radulescu, Cristina Nicolescu, Arjan van Gemund, and Pieter Jonker. Mixed Task and Data Parallel Scheduling for Distributed Systems. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, April 2001.

[10] Andrei Radulescu and Arjan van Gemund. A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling. In *Proceedings of the 15th International Conference on Parallel Processing (ICPP)*, Valencia, Spain, September 2001.

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

[12] Thomas Rauber and Gudula Rünger. Compiler Support for Task Scheduling in Hierarchical Execution Models. *Journal of Systems Architecture*, 45:483–503, 1998.

[13] Jaspal Subhlok and Bwolen Yang. A New Model for Integrated Nested Task and Data Parallel Programming. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Pratice of Parallel Programming*, pages 1–12, Las Vegas, June 1997. ACM Press.

[14] Emily West and Andrew Grimshaw. Braid: Integrating Task and Data Parallelism. In *Proceedings of the Fifth Symposium on Frontiers of Massively Parallel Computation*, pages 211–219. IEEE CS Press, 1995.