Task Scheduling for Parallel Multifrontal Methods

Olivier Beaumont and Abdou Guermouche

INRIA Futurs LaBRI, UMR CNRS 5800
Bordeaux, France
Olivier.Beaumont@labri.fr, Abdou.Guermouche@labri.fr

Abstract. We present a new scheduling algorithm for task graphs arising from parallel multifrontal methods for sparse linear systems. This algorithm is based on the theorem proved by Prasanna and Musicus [1] for tree-shaped task graphs, when all tasks exhibit the same degree of parallelism. We propose extended versions of this algorithm to take communication between tasks and memory balancing into account. The efficiency of proposed approach is assessed by a set of experiments on a set of large sparse matrices from several libraries.

Keywords. Sparse matrices, multifrontal method, scheduling, memory.

1 Introduction

The solution of sparse systems of linear equations is a central kernel in many simulation applications. Because of their robustness and performance, direct methods can be preferred to iterative methods. In direct methods, the solution of a system of equations $Ax = b$ is generally decomposed into three steps: (i) an analysis step, that considers only the pattern of the matrix, and builds the necessary data structures for numerical computations; (ii) a numerical factorization step, building the sparse factors (e.g., L and U if we consider an unsymmetric LU factorization); and (iii) a solution step, consisting of a forward elimination (solve $Ly = b$ for y) and a backward substitution (solve $Ux = y$ for x).

In this paper, we will work on an existing parallel sparse direct solver, `MUMPS` [2] (for MUltifrontal Massively Parallel Solver). We will study how to improve the parallel behavior of the solver. The main idea is to use theoretically proved techniques to improve the global behavior of the solver. Thus, Section 2 will be devoted to the presentation of parallel multifrontal method. Then, we will focus in Section 3 on the theoretical model and its application to `MUMPS` solver. We will present the adaptation of the algorithm for scheduling parallel tasks on homogeneous platforms proposed by Prasanna and Musicus [3,1] in Section 3.2. Finally, we present in Section 4 an experimental comparison between the existing `MUMPS` scheduling strategies and the techniques inspired from the work of Prasanna and Musicus. We will assess the interest and the limitations of new proposed approaches and then draw conclusions and give some words on future work.

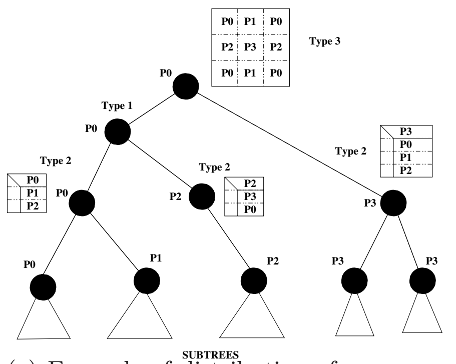
2 Parallel multifrontal method

We present in this section the parallel multifrontal method as implemented in the software package MUMPS [2].

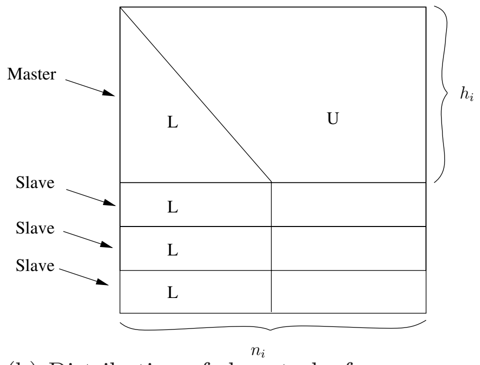
2.1 Task graphs within MUMPS

MUMPS uses a combination of static and dynamic approaches. The tasks dependency graph is indeed a tree (also called *assembly tree*), that must be processed from the leaves to the root. Each node of the tree represents the partial factorization of a dense matrix called *frontal matrix* or *front*. Once the partial factorization is complete, a block of temporary data (i.e. contribution block) is passed to the parent node. When contributions from all children are available on the parent, they can be consumed or assembled (*i.e.* summed with the values contained in the frontal matrix of the parent). Contribution blocks represent temporary data of the algorithm whereas factors represent final data.

The shape of the tree and costs of the tasks depend on the linear system to be solved and on the reordering of the unknowns of the problem. Furthermore, tasks are generally computationally larger near to the root of the tree where the parallelism of the tree is limited. Figure 1(a) summarizes the different types of parallelism available in MUMPS:



(a) Example of distribution of an assembly tree over four processors.



(b) Distribution of slave tasks for unsymmetric cases.

Fig. 1. Parallelism management in MUMPS.

The first type only uses the intrinsic parallelism induced by the tree (since branches of the tree can be processed in parallel). A type one node is a sequential task, that can be activated when results from children nodes have been communicated. Leave subtrees are a set of tasks all assigned to the same processor. Those are determined using a top-down algorithm [4] and a subtree-to-process mapping is used to balance the computational work of the subtrees onto the processors. The second type corresponds to parallel tasks; a 1D parallelism of large frontal matrices is applied: the front is distributed by blocks of rows. A *master* processor is chosen statically during the symbolic preprocessing step and all the others (*slaves*) are chosen dynamically by the master from a set of *candidate processors* based on load balance considerations, which can be either the number of floating-point operations still to be done, or the memory usage. The number and the choice of candidate processors is guided by a relaxed *proportional mapping* (see Pothen and Sun [5]) consisting of a recursive assignment of processors to subtrees according to their associated computational work. Note that once the partial factorization is done, the *master* processor eliminates the first block of rows, while slaves perform the updates on the remaining Schur complement (see Figure 1(b)). Finally, the task corresponding to the root of the tree uses a 2D parallelism, and do not require dynamic decisions: ScaLAPACK [6] is applied, with a 2D block cyclic static distribution.

The choice of the type of parallelism is done statically and depends on the height in the tree, and on the size of frontal matrices. The mapping of the masters of parallel tasks is static and only aims at balancing the memory of the corresponding factors. During the execution, several slave selections can be made independently by different master processors.

2.2 Dynamic scheduling strategy

A workload-based strategy under memory constraints is used to select slaves for parallel tasks. This strategy [7] is based on the number of floating-point operations still to be done. Each processor takes into account the cost of a task once it can be activated. In addition, each processor has as initial load the cost of all its subtrees.

The slave selection for parallel tasks (Type 2 nodes) is done such that selected slaves give the best workload balance. The matrix blocking for these nodes is an irregular 1D-blocking by rows. In addition, there are granularity constraints on the sizes of the subtasks for issues related to either performance or size of some internal communication buffers. Furthermore, this strategy dynamically estimates and uses information relative to the amount of memory available on each processor to constrain the schedule.

3 Load balancing and minimization of communication cost

3.1 Related works

In this section, we consider the problem of finding a schedule that both balances the load throughout the computation and minimizes the overall volume of communications induced by the algorithm. The problem of balancing memory requirements between processors will be addressed in Section 3.2.

Let us recall the algorithm defined in Section 2 from a scheduling point of view. The task graph corresponding to the execution of `MUMPS` is a tree (see Figure 1(a)) and communications take place along the edges of the tree. Each node of the tree can in turn be executed on several processors, which means that we do not consider tasks at the finest level of granularity: in the context of `MUMPS`, tasks are associated to partial LU decompositions.

This approach is closely related to malleable tasks scheduling (see [8] for a survey). A malleable task is a computational unit that can be itself processed in parallel. For each possible number of used processor, the time to process the malleable task is given, and communications are taken into account via a penalty factor. In [9], the authors propose a $4(1+\epsilon)$ approximation algorithm for scheduling trees of malleable tasks, if communications between malleable tasks are not taken into account. In the context of `MUMPS`, these theoretical results can nevertheless be improved, since all malleable tasks correspond to the same routine (partial LU factorization) on different data and for different problem size. In this context, all malleable tasks have the same profile (i.e. the penalty depends only on the number of processors, but not on the specific data). This problem has been addressed by Prasanna and Musicus [3,1]. In their model, it is assumed that the execution time of any malleable task is given by $\frac{L}{p^\alpha}$, where L is the length of the task on one processor, p is the number of processors allocated to this task and α is a penalty factor, that expresses the degree of parallelization of the malleable task (α close to 1 corresponds to ideal parallel task, whereas α close to 0 means that the task is intrinsicly sequential). It is worth noting that α does not depend on L or p (the value of α for our specific application will be discussed in next Section). For trees of such regular malleable tasks, Prasanna and Musicus [1] propose an optimal algorithm (that nevertheless allots rational number of processors to tasks), that will be described in more details in Section 3.2.

3.2 Parallelization of factorization in `MUMPS`

Fitting the model We experimentally measured values of α using frontal matrices having an order (n_i) of 10000 and various sizes of master task (an illustration of h_i is given in Figure 1(b)). We observed that for reasonable ratios of h_i/n_i, α has a constant value of 1.15 (note that if a task has a large master part, we can split it into a chain of tasks that have reasonable size of master tasks). This means that α is constant for all the tasks of the tree. Note that, the α parameter may vary depending on the platform used (network characteristics, processor speed, . . .).

However, we also observe super-linear speedups (in the sense that α is larger than 1). This surprising behavior is due to the fact that the processor that handles thes master task does not take part to the processing of the slave part of the task (a processor cannot be a master and a slave in the same partial factorization). Thus, for example, when doubling the number of processors for a given task from 2 to 4, the number of slaves varies from 1 to 3 (inducing a speedup near to 3 in an ideal case). This explains the observed super-linearity.

We plan to change the `MUMPS` management of these tasks to allow a processor to be both a master and a slave in the same task. A more powerful implementation would be to allow the parallelization of the master part of the task, and would remove this super-linear speedup.

Proposed solution In previous Section, we have shown that current implementation of partial LU factorization induces a super-linear speedup, thus preventing us to use directly optimality results derived by Prasanna and Musicus [3,1]. Nevertheless, we also observed that this super-linear speedup will disappear with the newly version of 1D partial LU factorization. Moreover, the optimal solution for $\alpha > 1$ would lead to execute each task on the whole set of processors, what would induce huge communication costs. Therefore, we propose to use the mapping algorithm proposed by Prasanna and Musicus even if $\alpha > 1$. It has been proved [3] that if $\alpha < 1$, the solution is as depicted in Figure 2. More precisely, the set of

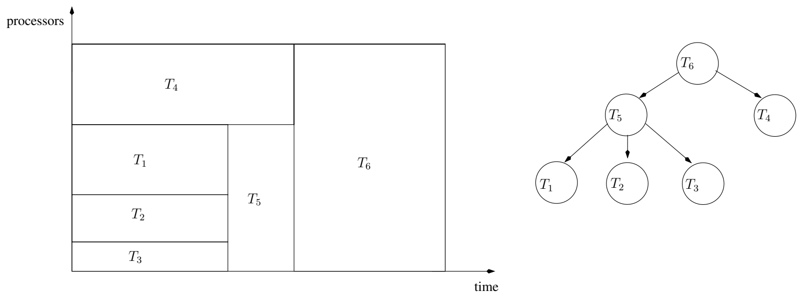


Fig. 2. Description of the optimal schedule

processors allocated to a task does not change over time, the set of processors allocated to a given node is the same as the set of processors allocated to its subtree, and all the children of a given task finish their execution at the same time. Given these observations, it is possible to determine the exact number of processors allocated to each task of the tree. For instance, if p_5 denotes the number of processors allocated to task T_5, the number of processors p_i allocated to tasks $T_i,\ 1 \leq i \leq 3$ is given by

$$p_i = p_5 \frac{L_i^{\frac{1}{\alpha}}}{L_1^{\frac{1}{\alpha}} + L_2^{\frac{1}{\alpha}} + L_3^{\frac{1}{\alpha}}},$$

where L_i denotes the execution time of T_i on one processor.

As already noted, the optimal solution associates each task T_i to a fractional number of processors, given as an interval with rational bounds $[l_i, r_i] \subset [0, P]$, where P is the overall number of processors. Therefore, each task T_i is associated to $k_i \geq 0$ processors (denoted as base processors in what follows) that will be completely devoted to the execution of T_i, and possibly $k'_i (k'_i \leq 2)$ extra processors (denoted as candidate processors in what follows), that will be also partially allocated to other tasks during the execution of T_i. Candidate processors for task T_i will be dynamically allocated to the execution of T_i during the execution, given their current load at the beginning of the execution of T_i.

Proposed solution therefore achieves perfect load balancing (all processors work during the whole process) and good locality of communications (since the set of processors allocated to a given task is the union of the set of processors allocated to its children tasks). On the other hand, with the current implementation of partial LU factorization in `MUMPS`, the length of the schedule is not optimal since $\alpha > 1$. Note that the main difference between this approach and the default `MUMPS` scheduling strategy (see Section 2), which is based on proportional mapping, is that it cases the communications induced by parallelism into account through the task performance model.

Minimization of memory requirements In this section, we consider the minimization of memory requirements once the tree of tasks and the set of processors (base processors and candidate processors) allocated to each task have been determined. More specifically, we concentrate on the minimization of the memory needed to store L and U factors and do not consider the memory needed to store intermediate factors that will be sent (and then removed from memory) to the father task. Let us consider the elementary partial factorization depicted in Figure 1(b). The processor responsible for computing the upper part will have to store $n_i h_i$ elements of L and U, whereas the $(p-1)$ processors responsible for the lower part will be in charge of storing $\frac{h_i(n_i-h_i)}{p-1}$ elements of L. In order to decide which processor will be in charge of computing the upper part, we propose a heuristic based on the $\frac{4}{3}$ approximation algorithm Minimum Multiprocessor Scheduling [10] where task lengths are independent of the processor's choice [11]. More precisely, we sort the tree nodes by decreasing values of h_i and we consider tasks in this order, allocating the upper part of task T_i to the less loaded processor, while updating the memory charge for all processors participating to the computation of T_i. We present in next section the results obtained by this simple heuristic.

4 Experimental results

We should first mention that the algorithms presented in Sections 3.2 and 3.2 have been implemented inside the `MUMPS` package. In order to compare the proposed algorithm with the default scheduling strategy described in Section 2, we experiment them on several problems (see Table 1) extracted from various sources including Tim Davis's collection at University of Florida [1] or the PARASOL collection[2]. The tests have been performed on the IBM SP system of

[1] `http://www.cise.ufl.edu/ davis/sparse/`
[2] `http://www.parallab.uib.no/parasol`

IDRIS[3] composed of several nodes of either 4 processors at 1.7 GHz or 32 processors at 1.3 GHz. Note that all the experiments are done using unsymmetric matrices. The extension to symmetric ones is natural.

Matrix name	Order	nnz	$nnz(L\|U) \times 10^6$	Description
CONV3D64	836550	12548250	2693.9	provided by CEA-CESTA; generated using AQUILON (http://www.enscpb.fr/master/aquilon)
GRID	729000	7905960	1430.5	Regular 90-90-90 grid
MHD1	485597	24233141	1250.3	unsymmetric magneto-hydrodynamic 3D problem, provided by Pierre Ramet
ULTRASOUND80	531441	33076161	981.4	Propagation of 3D ultrasound waves, provided by M. Sosonkina

Table 1. Test problems.

We have tested the algorithms presented in previous Sections on 32 and 64 processors of the above-described platform. By default, we used the METIS package [12] to reorder the variables of the matrices. In the following experimental study we will use `default` to denote the default scheduling strategy of the solver, `P&M` to denote the experiments where we used the Algorithm presented in Section 3.2, and `P&M*` to denote the variant described in Section 3.2. In addition, for each set of experiments, we forced the same tree topology (same amount of splitting etc ...). Finally, in the case of `P&M` approaches, we strictly follow the static schedule, produced during the analysis phase, during factorization.

We report in Table 2 factorization times on the IBM platform using 32 and 64 processors. The factorization time is reduced when using the `P&M` approaches on both 32 and 64 processors. The gains can reach more than 30% (for the conv3d64 on 64 processors) and variants of the Prasanna & Musicus do not strongly differ in terms of factorization time. Note that on 32 processors, the `P&M*` seems to be less efficient than the `P&M` approach. This is principally due to our round-off management when assigning tasks to processors. Indeed, as mentioned in Section 3.2, if a processor is not fully assigned to a task, we choose dynamically if it will take part to the processing of the task or not. These dynamic decisions explain the difference in terms of performances between the two variants. Presented results illustrate the good behavior of both variants and show that they produce a better balanced schedule than the default strategy.

We will now focus on the volume of data exchanged during the factorization. We give in table 3 the volume of communication measured during the factorization. We can see that both `P&M` and `P&M*` give a very reduced amount of communication (up to a factor of 2.5). This reduction is explained by the fact that the algorithms based on Prasanna and Musicus approach have a natural locality in the distribution of tasks over processors (see Section 3.2). In addition, the number of sequential subtrees assigned to a single processor is smaller in the case of Prasanna and Musicus variants (which means that more work is done without communications).

[3] Institut du Dveloppement et des Ressources en Informatique Scientifique

Matrix name	Time for facto. (32 procs.)			Time for facto. (64 procs.)		
	`default`	`P&M`	`P&M*`	`default`	`P&M`	`P&M*`
CONV3D64	-	-	-	390	280	274
GRID	237	199	208	169	117	115
MHD1	186	168	175	116	102	99
ULTRASOUND80	89	88	95	82	63	62

Table 2. Factorization time (in seconds) on 32 and 64 processors. Not enough memory was available to run the conv3d64 matrix on 32 processors

Matrix name	Vol. of comm. (32 procs.)			Vol. of comm. (64 procs.)		
	`default`	`P&M`	`P&M*`	`default`	`P&M`	`P&M*`
CONV3D64	-	-	-	618	234	229
GRID	184	74	73	308	131	120
MHD1	138	66	61	219	110	103
ULTRASOUND80	79	34	34	167	77	69

Table 3. Volume of communication (in GigaBytes) on 32 and 64 processors. Not enough memory was available to run the conv3d64 matrix on 32 processors

Finally, we will focus on the memory behavior of the different approaches. We report in Table 4 both the memory peak over the set of processors for performing the factorization and the final size of factors (we give the maximum size of factors over all processors). A first observation is that the memory behavior of the algorithms based on Prasanna & Musicus approach is worst than the `default` one. This is mainly due to the fact that in the `default` strategy, memory constraints are injected in both static and dynamic decisions. This leads to better memory behavior (especially for the management of temporary data (i.e. contribution blocks).

Matrix name	Mem. peak (32 procs.) / Size of factors (32 procs.)			Mem. peak. (64 procs.) / Size of factors (64 procs.)		
	`default`	`P&M`	`P&M*`	`default`	`P&M`	`P&M*`
CONV3D64	-	-	-	86 / 53	109 / 87	102 / 57
GRID	89 / 55	122 / 77	112 / 68	52 / 27	61 / 36	59 / 35
MHD1	79 / 48	108 / 56	121 / 56	41 / 21	56 / 33	53 / 32
ULTRASOUND80	59 / 37	82 / 49	82 / 49	30 / 19	45 / 25	46 / 25

Table 4. Peak of memory (in the left) and size of Factors per processors (in the right) (in millions of entries) on 32 and 64 processors. Not enough memory was available to run the conv3d64 matrix on 32 processors

From the factor (terminal data) size point of view, we can see as expected that the `P&M*` approach has a slightly better behavior than the `P&M` one, what illustrates the benefits of the mechanism described in Section 3.2. However, we can also see that `default` strategy gives a more balanced distribution of the size of factors. This is due to the fact that in this approach the layer of sequential subtrees is determined by a combination of workload and memory criteria whereas in the P&M and P&M* strategies it is built based on workload information only. Thus, the size of sequential subtrees (which are considered as a single task) is bigger in the P&M and P&M* approaches, what makes the distribution of factors over processors more difficult.

5 Conclusion and Future work

We presented in this paper a study of scheduling strategies for the parallel multifrontal method implemented in the MUMPS software package. We showed that the model of the application fits a state of the art model and how to adapt the scheduling algorithm implemented in the solver. Finally, we presented an experimental study showing the potential of the approach. We observe that new schedules improve the performances of the solver by achieving better load-balancing and a reduce volume of communication. However, we also observe that these techniques induce an increase of memory requirements. This last issue is critical, especially in the area of sparse direct solvers (where memory is often the bottleneck). Thus, we have to work on approaches that will slightly degrade performance to improve the memory behavior, either by injecting memory information during the static allocation or by dynamically relaxing proposed allocation. Another approach could be to study bi-criteria techniques aiming at finding the best tradeoff between these two criteria. Finally, we plan to study how this work can be extended to the context of parallel out-of-core sparse direct solvers (in this new context, I/Os have to be taken into account).

References

1. Prasanna, G.N.S., Musicus, B.R.: Generalized multiprocessor scheduling and applications to matrix computations. IEEE Trans. Parallel Distrib. Syst. **7**(6) (1996) 650–664
2. Amestoy, P.R., Duff, I.S., Koster, J., L'Excellent, J.Y.: A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIMAX **23**(1) (2001) 15–41
3. Prasanna, G.N.S., Musicus, B.R.: Generalised multiprocessor scheduling using optimal control. In: SPAA. (1991) 216–228
4. Geist, A., Ng, E.: Task scheduling for parallel sparse Cholesky factorization. Int J. Parallel Programming **18** (1989) 291–314
5. Pothen, A., Sun, C.: A Mapping Algorithm for Parallel Sparse Cholesky Factorization. SISC **14(5)** (1993) 1253–1257
6. Choi, J., Demmel, J., Dhillon, I., Dongarra, J., Ostrouchov, S., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. Technical Report LAPACK Working Note 95, CS-95-283, University of Tennessee (1995)
7. Amestoy, P.R., Guermouche, A., L'Excellent, J.Y., Pralet, S.: Hybrid scheduling for the parallel solution of linear systems. Parallel Computing **32**(2) (2006) 136–156
8. Dutot, P.F., Mounie, G., Trystram, D.: Scheduling parallel tasks: Approximation algorithms. In Leung, J., ed.: Handbook on Scheduling algorithms: Algorithms, Models and Performance Analysis. CRC press (2004)

9. Lepere, R., Mounie, G., Trystram, D.: An approximation algorithm for scheduling trees of malleable tasks. European Journal of Operational Research **142** (2002) 242–249
10. Ausiello, G., Crescenzi, P., Gambosi, G., Kann, V., Marchetti-Spaccamela, A., Protasi, M.: Complexity and Approximation: Combinatorial optimization problems and their approximability properties. Springer Verlag (1999)
11. S., H.D., Shmoys, D.B.: Using dual approximation algorithms for scheduling problems: theoretical and practical results. J. ACM **34** (1987) 144–162
12. Karypis, G., Kumar, V.: MeTiS – A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0. University of Minnesota. (September 1998)