View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Open Archive Toulouse Archive Ouverte

OATAO
Open Archive Toulouse Archive Ouverte

Open Archive TOULOUSE Archive Ouverte (OATAO)

OATAO is an open access repository that collects the work of Toulouse researchers and makes it freely available over the web where possible.

This is an author-deposited version published in : http://oatao.univ-toulouse.fr/
Eprints ID : 16822

The contribution was presented at VECPAR 2014 :
http://www.vecpar.org/2014/

To cite this version : Amestoy, Patrick and L'Excellent, Jean-Yves and Rouet, François-Henry and Sid-Lakhdar, Mohamed *Modeling 1D distributed-memory dense kernels for an asynchronous multifrontal sparse solver*. (2014) In: 11th International Meeting High-Performance Computing for Computational Science (VECPAR 2014), 30 June 2014 - 3 July 2014 (Eugene, Oregon, United States).

Any correspondence concerning this service should be sent to the repository administrator: staff-oatao@listes-diff.inp-toulouse.fr

Modeling 1D Distributed-Memory Dense Kernels for an Asynchronous Multifrontal Sparse Solver

Patrick R. Amestoy[1], Jean-Yves L'Excellent[2], François-Henry Rouet[3], and Wissam M. Sid-Lakhdar[4](✉)

[1] University of Toulouse, INPT(ENSEEIHT)-IRIT, Toulouse, France
[2] University of Lyon, Inria and LIP (CNRS, ENS Lyon, Inria, UCBL), Lyon, France
[3] Lawrence Berkeley National Laboratory, Berkeley, USA
[4] University of Lyon, ENS Lyon and LIP (CNRS, ENS Lyon, Inria, UCBL), Lyon, France
mohamed.sid_lakhdar@ens-lyon.fr

Abstract. To solve sparse systems of linear equations, multifrontal methods rely on dense partial *LU* decompositions of so-called frontal matrices; we consider a parallel asynchronous setting in which several frontal matrices can be factored simultaneously. In this context, to address performance and scalability issues of acyclic pipelined asynchronous factorization kernels, we study models to revisit properties of left and right-looking variants of partial *LU* decompositions, study the use of several levels of blocking, before focusing on communication issues. The general purpose sparse solver MUMPS has been modified to implement the proposed algorithms and confirm the properties demonstrated by the models.

1 Introduction

Multifrontal methods [2] are widely used to solve sparse systems of equations of the form $Ax = b$, where A is a sparse matrix, b is the right-hand side and x the unknown. They cast the factorization of the sparse matrix A into a series of partial factorizations of smaller dense matrices, called *fronts*, or *frontal matrices*. The dependency graph between those partial dense factorizations is a tree (the *assembly tree*), processed from the leaves to the root, such that the Schur complement so called *contribution block (CB)* produced after the partial factorization of a front is used at the parent node to build the front of the parent in a so-called *assembly operation*, before the parent node is in turn partially factored.

In this paper, we focus on the dense factorization kernels used in multifrontal methods for unsymmetric matrices where an *LU* decomposition is applied. For more information on multifrontal methods, we refer the reader to [11,15]. Much work has been done and is being done by the dense linear algebra community on *LU* factorizations, using for example static 2D block-cyclic data distributions [8], sometimes 2.5D communications [19], or DAG-based tiled algorithms in both shared memory [1,7] and distributed-memory environments [5]. Recent asynchronous approaches often rely on a task scheduling engine [4,6] and on fine-grain parallelism for an efficient utilization of the computing resources. Most often, the choice of using an asynchronous approach with fine-grain parallelism in both directions (2D) implies relaxed pivoting strategies (such as *tournament pivoting*, typically used in communication-avoiding algorithms [13]). This is because neither full rows nor full columns are available to test for pivots stability. This is especially the case in distributed-memory environments, with the exception of the (synchronous) ScaLAPACK library [8].

In multifrontal-based, asynchronous, distributed-memory sparse factorization methods, many dense frontal matrices may be factorized simultaneously. Processes might thus be involved in more than one dense factorization, depending on dynamic scheduling decisions based on current CPU load and memory usage of each process and this is thus quite difficult to predict. We are also concerned with numerical accuracy and thus want to maintain standard numerical threshold pivoting [10] even in a distributed-memory context, which is quite a unique feature for a general purpose distributed-memory solver. In this context, a one-dimensional distribution of the dense factorization of fronts makes sense and has been adapted [3]. We are thus interested in analyzing and pushing the limits of this one-dimensional distribution. As we will show, analytical models can be complex because of the discrete nature of the phenomena. We have therefore also developed simulator that models parallel executions for standard blocked variants (so-called left and right-looking [12]) of the dense factorization of multifrontal fronts. We note that our objective here is to model the individual dense multifrontal kernels and not an entire sparse multifrontal factorization, although the findings will have an impact on the overall peformance of a sparse multifrontal factorization. Although cyclic pipelined factorizations have been modeled in the past [9], we are not aware of a clear illustration of the natural and intuitive properties of left-looking and right-looking approaches in a context comparable to ours, with acyclic factorizations, and where the process in charge of factorizing rows of the matrix does it either in an LL or RL way whereas other processes always perform their updates as soon as possible in a RL manner. For ScaLAPACK which relies on a 2D block cyclic distribution, right-looking is preferred over left-looking [8]; however, with our 1D technique, our conclusion is different, as will be illustrated in this paper.

The paper is organized as follows. In Sect. 2, we first study the theoretical behaviour of left-looking and right-looking variants for both one and two levels blocked algorithms. In order to better reveal and illustrate some of the intrinsic properties of those algorithms, we first consider a network with infinite bandwith. Communication models are then studied in more detail in Sect. 3 where we also analyse buffer memory requirements, cost of asynchronous one-to-many communications and impact on the blocked variants. This analysis has been used to modify the general purpose distributed-memory solver MUMPS [3] and to illustrate in Sect. 4 the benefits of the proposed approach in a distributed-memory environment.

2 Modeling Left-Looking and Right-Looking Computations

We consider a distributed-memory dense partial factorization relying on a dynamic asynchronous pipelined algorithm. A one-dimensional (1D) data distribution is used to allow for efficient pivot searches without synchronization between processes. In order to partially factorize the first *npiv* rows/columns of a front of order *nfront* using *nproc* MPI processes, one process designated as the *master* will handle the factorization of the *npiv* rows and the *nproc*-1 other processes (called *workers*) will manage the update of the so called *CB rows* of size $ncb = nfront - npiv$ (see Fig. 1). The master uses a blocked *LU* algorithm with threshold partial pivoting: pivots are checked against the magnitude of the row but pivots can only be chosen within the first $npiv \times npiv$ block. After factorizing a panel of size *npan*, the master sends it to the workers in a non-blocking way, along with pivoting information. The master can immediately update its remaining non-factored rows (right-looking approach) or postpone this to when the next panel will start (left-looking approach). In parallel, the workers update all their rows at each panel reception. Thus, the behaviour of the workers always follows a right-looking scheme. The factorization operations rely on BLAS1 and BLAS2 routines inside panels, whereas update operations (both on master and workers) rely on BLAS3 routines, where TRSM is used to update columns of newly eliminated pivots and GEMM is used to update the remaining columns. For the sake of clarity, we consider that CB rows are uniformly distributed over the workers.

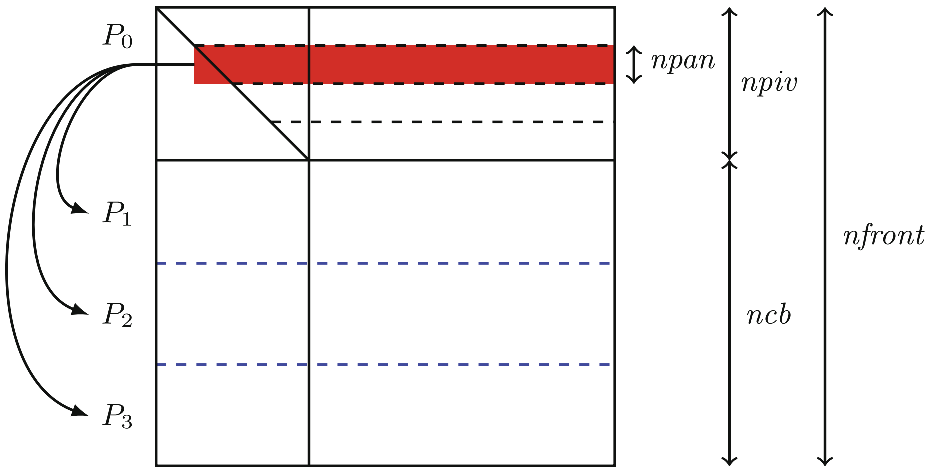


Fig. 1. Partial factorization of a front of order *nfront*, with *npiv* variables to eliminate by panels of *npan* rows, and $ncb = nfront - npiv$ rows to be updated.

We have first modeled the factorizations analytically. Figure 2 shows the context and main notations. We have used the *MAPLE* software to help in this task, due to the complexity of the equations arising when solving problems such as finding optimal parameters of the factorizations.

Equation 1 represents the number of floating-point operations necessary for the factorization of a panel of k rows and $k+n$ columns.

$$Wf(k,n) \rightarrow \left(\frac{2}{3}\right) k^3 + \left(n - \frac{1}{2}\right) k^2 - \left(n + \frac{1}{6}\right) k \tag{1}$$

This is the result of the sum of the floating-point operations of the factorization of each row:

$$Wf(k,n) \rightarrow \sum_{i=k-1}^{0} i + 2 * i * (i+n) \tag{2}$$

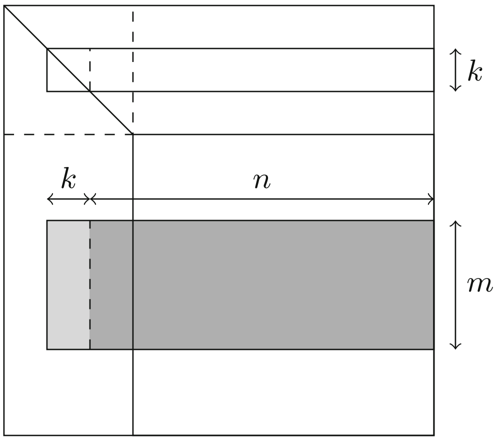


Fig. 2. Illustration of the factorization of a panel of size $k \times (k+n)$ on the master and the corresponding update on a worker. The light and dark gray areas represent the pieces of the front on a worker on which a TRSM and GEMM are applied respectively.

Equation. 3 represents the number of floating-point operations necessary for the update of a block (factorization of the L factors and update of the contribution part) of m rows and $k+n$ columns by a panel of k rows and $k+n$ columns (we thus assume a right-looking algorithm).

$$Wu(m,n,k) \rightarrow W_{TRSM}(m,k) + W_{GEMM}(m,n,k) \tag{3}$$

with

$$W_{TRSM}(m,k) \rightarrow mk^2 \tag{4}$$

and

$$W_{GEMM}(m,n,k) \rightarrow 2\,mnk \tag{5}$$

- Given a GFlops/s rate β for update operations (TRSM and GEMM), MU_i, the time of update related to the i^{th} panel by the master is given by:

$$\begin{aligned} MU_i = \beta \times\ Wu\left(npiv - \min\left(npiv, i * npan\right), npiv + ncb- \right., \\ \left. \min\left(npiv, i * npan\right) \min\left(npan, npiv - \left(i-1\right) npan\right)\right) \end{aligned} \tag{6}$$

- SU_i, the time of update related to the i^{th} panel by a worker is given by:

$$SU_i = \beta \times Wu\left(\frac{ncb}{nslave}, npiv + ncb - \min\left(npiv, i * npan\right),\right.$$
$$\left.\min\left(npan, npiv - (i-1)\, npan\right)\right) \qquad (7)$$

- Given a GFlops rate α for the panel factorization (including some BLAS2 operations), MF_{i+1}, the time of factorization of the $(i+1)^{th}$ panel (if it exists) by the master is given by:

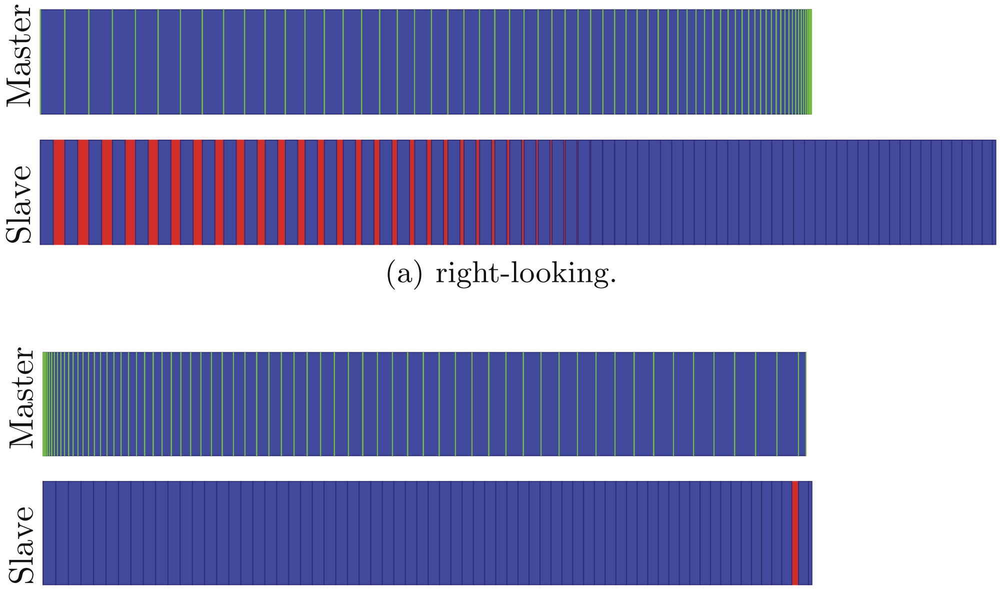
$$MF_{i+1} = \alpha \times Wf\left(\min\left(npan, npiv - npan \min\left(i, floor\left(\frac{npiv}{npan}\right)\right)\right),\right.$$
$$\left. npiv + ncb - (i+1) * npan\right) \qquad (8)$$

The total factorization time of a RL factorization is then given by Eq. 9:

$$TRight = MF_1 + \sum_{i=1}^{ceil\left(\frac{npiv}{npan}\right)} \max\left(SU_i, MU_i + MF_{i+1}\right) \qquad (9)$$

An algorithm where the master uses an LL factorization can be modeled in a similar way. Furthermore, communication costs can also be taken into account in Formula 9 in a simple manner. At the price of complicated formulas it is then possible with the help of Maple to build analytical formulas to express some properties (efficiency, speed-up, ...). However, we have preferred to consider the implementation of a simpler Python simulator for distributed-memory factorizations. Our simulator is naturally able to take into account varying communication and computation models and to produce Gannt-charts of the factorization. In order to illustrate some intrinsic properties of the algorithms that do not depend on the network bandwidth, we consider, to start with, that communications take place on a network with infinite bandwidth γ and that computations take place at a constant GFlops rate ($\alpha = \beta$). Because the messages sent are always reasonably large, we consider that the network latency is always negligible.

Right-Looking and Left-Looking Algorithms. In order to better characterize the main properties of our algorithms, we consider here a situation where the number of floating-point operations (flops) on the master is equal to that of each worker. Figure 3 represents the Gantt charts for $nfront = 10000$ and $nproc = 8$ (in this case $npiv = 2155$ to equilibrate flops) using both right-looking (RL) and left-looking (LL) blocked factorizations on the master, while workers perform their updates at each received panel, in a right-looking way. In each subfigure, the Gantt chart on the top represents the activity of the master and the bottom one that of a single worker. Because all workers theoretically behave the same way, only one worker is represented in the Gantt chart. Figure 3(a)



(a) right-looking.

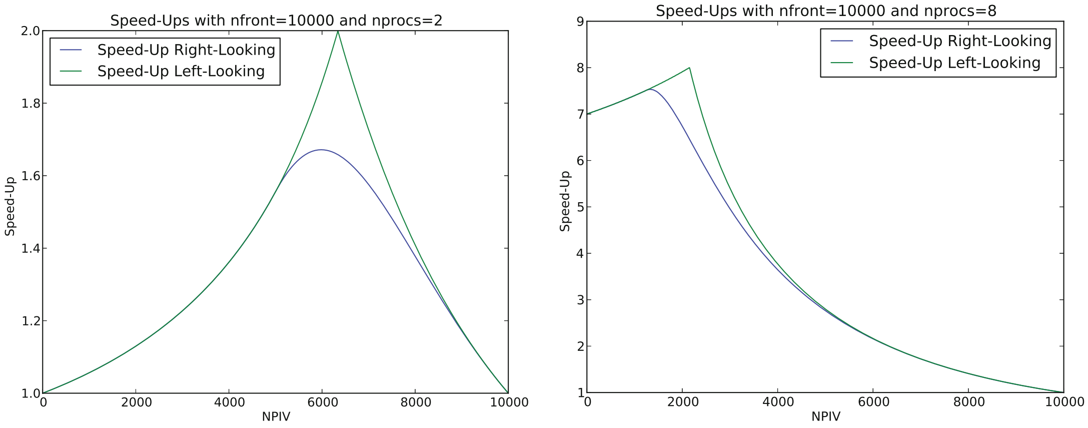
(b) left-looking.

Fig. 3. Gantt chart of the RL and LL algorithms. Factorization in green, updates in blue and idle times in red (Color figure online).

clearly illustrates the weakness of the RL approach. Given that *npiv* balances the total amount of work (flops) between master and workers, one would expect all processes to finish at the same time. However, the workers finish much later because they have idle phases that sum up to the gap between master and workers completion times. When computing the first panels, the master process performs more update operations than the workers, which makes them become idle. The amount of update operations relative to each panel decreases faster on the master process than on the workers, and idle times decrease. When panels get smaller, the master process performs less operations than the workers and sends panels to the workers quicker than the workers manage to perform the corresponding updates; the workers then work continuously, desperately trying to catch up with their delay. As the consumption of factored panels is critical on the workers, the master should produce panels as soon as possible, delaying its own updates as much as possible. A solution consists in applying on the master a left-looking algorithm instead, resulting in the perfect Gantt chart of Fig. 3(b). In the following subsections, we compare the behavior of both variants.

Load Balance and Scalability. Although the ratio between *npiv* and *nfront* is mainly defined by the sparsity pattern of the matrix to be factored, we will show at the end of this section that we have some leeway to modify this ratio; in Fig. 4(a), we study the influence of *npiv* for a fixed *nfront*. We distinguish three parts, depending on *npiv*. In the first part, for *npiv* under a certain value $npiv_0$ ($npiv_0 \approx 5000$), LL and RL algorithms behave exactly the same: workers are the bottleneck because they have much more work than the master. For

$npiv > npiv_0$, LL becomes better than RL: $npiv_0$ is the value above which the time to apply the update (RL) of the first panel and factorize the second one on the master becomes bigger than time to apply the update associated to the first panel on the worker. Both variants reach their peak speed-up but for different values of *npiv*. Then, for large values of *npiv*, the master has much more work to do than the workers and becomes the bottleneck, leading to an asymptotic speed-up of one.



(a) Speed-up with $nproc = 2$. (b) Speed-up with $nproc = 8$.

Fig. 4. Influence of *npiv* on LL and RL algorithms with 2 (left) and 4 (right) processes: speed-ups with respect to the serial version ($nfront = 10000$).

When *nproc* is larger — Fig. 4(b), the maximum speed-ups of RL and LL tend to get closer. LL reaches its maximum speed-up when all processes (master and worker) get the same amount of computations *Flops equilibrium* (*eqFlops*), so that neither the master nor the workers are bottlenecks to each other. On the other hand, RL reaches its maximum speed-up when all processes (master and worker) are roughly assigned the same number of rows *Rows equilibrium* (*eqRows*). This latter approximation relies on the fact that this keeps workers always busy, leading to a speed-up at least equal to $nproc - 1$.

The previous theoretical model showed interesting results. However, in order to benefit from them, we must first ensure that some fundamental hypotheses hold true in practice. We show here the observed discrepancies and the algorithms and techniques we applied to fix or reduce them.

Generalization to Multiple Levels of Panels and to Arbitrary Front Shapes. The previous models showed that front factorizations are efficient when the ratio $\frac{npiv}{nfront}$ respects *eqRows* and *eqFlops* for RL and LL, respectively. In order to improve locality and BLAS3 effects on the master, recursive algorithms can be used [20]. However, at the first level of recursion, the update of the second block with the first one would take a significant amount of time, possibly making the workers idle for a huge period. The adopted solution consists in using multiple levels of blocking (in our case, two levels), which means computing an external panel using internal ones. Because the GFlops rate on the master may still be slightly lower than on the workers, corresponding to a smaller value of α than β in the models, one must slightly modify the *eqFlops* ideal $\frac{npiv}{nfront}$ ratio (for LL) in order to obtain $\frac{flops_{master}}{GFlops\ rate_{master}} = \frac{flops_{worker}}{GFlops\ rate_{worker}}$.

Another issue is that in practice, the multifrontal method results in frontal matrices that often have an $\frac{npiv}{nfront}$ ratio larger than the ideal one, especially for large *nproc*. Fortunately, assembly trees are not rigid entities and can be reshaped, for example using two standard operations known as *amalgamation* and *splitting*. Amalgamation consists in merging two related fronts into a single one (a child and its parent, usually). It has the advantage of generating larger fronts, which increases factorizations efficiency, sometimes at the cost of extra fill-ins, inducing more computations and memory requirements. Contrarily, splitting consists in cutting a front into a so called *split chain* of fronts such that in the chain, the Schur complement of a child is considered as a new, parent, front. We note that remapping may have to be done between two successive fronts in a chain, and that, although we consider *nproc* constant in our models and experiments, dynamic scheduling strategies may imply variations of *nproc* between two successive pipelined factorizations. Lost processes can be assigned to other fronts in other subtrees; vice versa, new processes can be assigned to parent fronts in the chain. In both cases, the shape of the fronts and the length of the chain should be modified accordingly, with the aim to obtain a correct balance of the work between master and workers in all intermediate fronts (except, possibly, for the last one). Simple models of such chains were discussed in [16] and have been revisited in [17]. In Fig. 5, we report the simulated speed-ups with varying *npiv* when this generalized approach is applied, with *eqRows* for RL and *eqFlops* for LL. For both RL and LL, the speed-ups are much less sensitive to

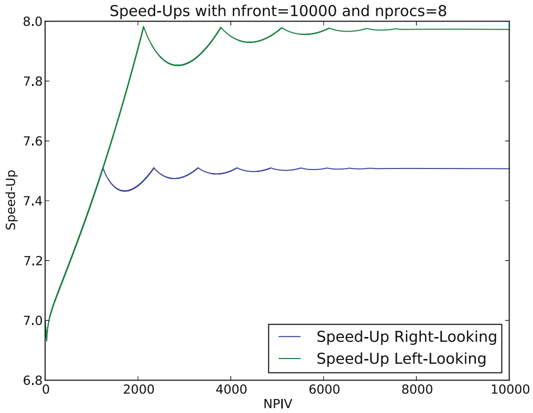


Fig. 5. Simulated generalized 1D factorization (*nfront* =10000, *nproc* =8) with varying *npiv*. LL (resp. RL) uses *eqFlops* (resp. *eqRows*).

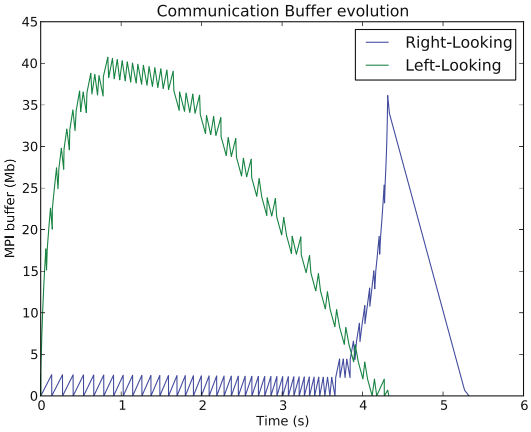


Fig. 6. Amount of data sent but not ready to be received using RL an LL algorithms with *eqFlops* (*nproc* =8, *nfront* =10000, *npiv* =2155).

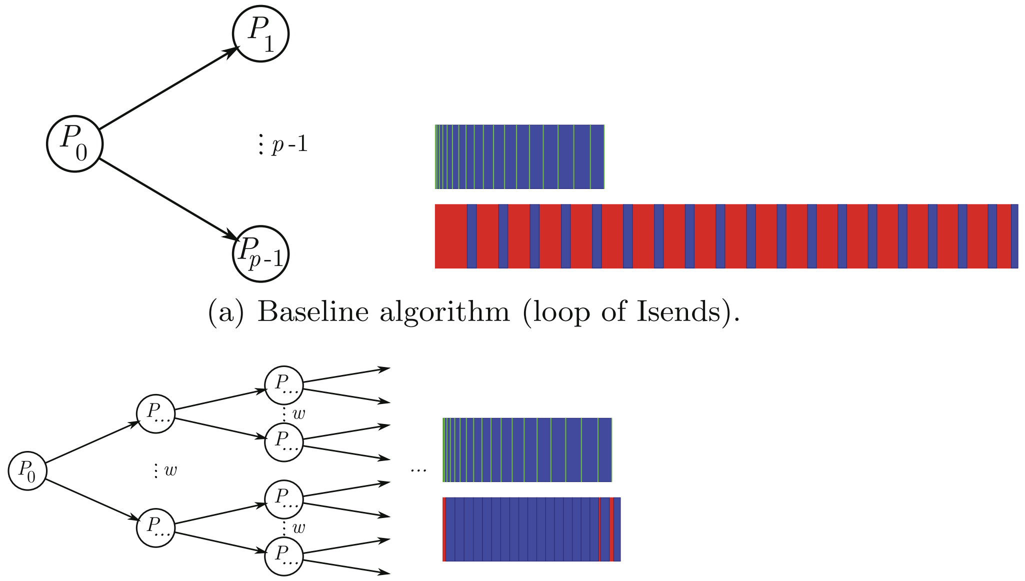
npiv (compared to Fig. 4) because each intermediate 1D factorization is now well-balanced. RL speed-ups are not as good as LL ones because of idle times on the master. When targeting an entire sparse matrix factorization rather than focusing on a single front or a chain of fronts, new kinds of load balancing issues arise, which are handled in our context study by a dynamic and asynchronous scheduling approach, which adapts to the load of the processes.

3 Modeling Communications

Memory for Communication. Assuming that sends are performed as soon as possible, Fig. 6 represents the evolution of the memory utilization in the send buffer for LL and RL factorizations, both with *eqFlops*. This send buffer is the place in memory where panels computed by the master are temporarily stored (contiguously) and sent using non-blocking primitives; when the workers start receiving, send buffer can often be freed. This allows for an overlap of computations and communications, and allows the main process to manage its memory independently of the advancement of communications. The memory utilization in the send buffer then represents the volume of data that has been sent and that is not received yet. Its size needs to be controlled and limited: a full send buffer implies in practice that the sender will wait for receptions to occur before being able to perform a new send. Most of the time, the buffer in the RL variant only contains one panel, immediately consumed by the workers; When master computations shrink (for the last panels), the master rapidly produces many panels that cannot be consumed immediately. In contrast, the LL variant always has enough panels ready to be sent. This is because RL with *eqFlops* is not able to correctly feed the workers, whereas the LL does. Second, the peak of buffer memory used for RL is 36 MB while it is 41 MB for LL. The scheduling advantage of LL thus comes at the price of a higher buffer memory usage. However, this additional memory becomes significant in comparison to the total memory used by the master process for the factorization ($nfront * npiv * sizeof(double)$ = 172 Mb). Send buffers may have a given limited size in practice, smaller than the peaks from Fig. 6 (36 MB and 41 MB for RL and LL variants, respectively). If only a few panels can fit in buffer memory, the master must wait when the send buffer is full, leading to some performance loss. Instead, we prefer to copy new panels to the send buffer only when space is available in the buffer, independently of the fact that many more panels may have been computed. This study also shows that, in order to control buffer memory, messages should *not* be sent as soon as possible (but should still be sent early enough so that receivers do not have to wait).

Limited Bandwidth and Asynchronous Collective Communications. We observed experimental results to be very similar to those of the model, as long as the ratio between computations and communications remains large enough (*nfront* relatively large compared to *nproc*). Strong scaling, i.e., increasing *nproc* for a given *nfront*, globally increases the amount of communications while keeping the amount of computations identical. The master process sends a copy of each panel to more workers, decreasing the bandwidth dedicated to the transmission of a panel to each worker: the maximal master bandwidth is divided by *nworkers* in this one-to-many communication pattern, making the communication of the panels from master to workers a possible bottleneck.

Many efficient broadcast implementations exist for MPI [21], and asynchronous collective communications are part of the MPI-3 standard. However the semantic of these operations requires that all the processes involved in the collective operation call the same function (`MPI_IBCAST`). This is constraining for our asynchronous approach which is such that any process, at any time, receives and treats any kind of message and task: we want to keep a generic approach where processes do not know in advance if the next message to receive in the main reception buffer is a factored panel or some other message. Furthermore, we need an asynchronous, pipelined broadcast algorithm which means that a binomial broadcast tree would not be appropriate since once a process has received a panel and forwarded it, its bandwidth will be needed to process next panel. For these reasons, we have designed our own asynchronous pipelined broadcast algorithm based on `MPI_ISEND` calls using a classical w-ary broadcast tree, as illustrated in Fig. 7(b). The Gantt charts of Fig. 7 show the impact of the communication patterns with limited bandwidth per process, using our Python simulator. With the baseline communication algorithm, the workers are most often idle, spending their time waiting for the communications to finish, before doing the corresponding computations, whereas the tree-based (here using a binary tree) has a perfect behaviour: the Gantt chart of the worker is only slightly translated



(a) Baseline algorithm (loop of Isends).

(b) Tree-based algorithm.

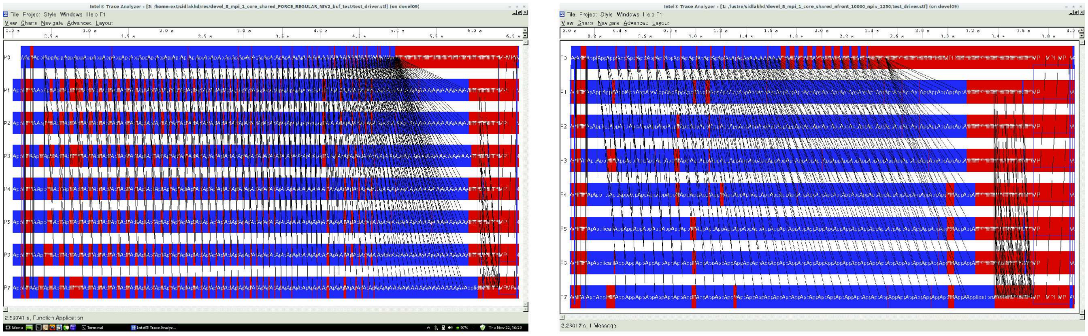
Fig. 7. Influence of the IBcast communication pattern with a limited bandwidth per proc (γ=1.2 Gb/s, α=10 GFlops/s) on LL algorithm with *nfront* = 10000, *npan* = 32, *nproc* = 32 and *npiv* chosen to balance work (idle times in red) (Color figure online).

in time (due to the time it takes to receive the first panel) and the remaining communications overlap well with computations. When further increasing *nproc* or with more cores per process, we did not always observe such a perfect overlap of communications and computations, but the tree-based algorithm always led to an overall transmission time for each panel of $\frac{nfront \times npan \times w \times \log_w(nproc)}{\gamma}$, much smaller than that of the baseline algorithm $\frac{nfront \times npan \times (nproc-1)}{\gamma}$. An IBcast-like scheme is thus of great importance when the number of processes grows.

4 Preliminary Experimental Results

In order to study the left-looking and right-looking variants of the 1D pipelined factorization algorithm from Sect. 2 on arbitrary fronts, we generalized the asynchronous factorization algorithms available in the MUMPS solver [3] in order to implement left-looking and right-looking variants with several levels of blocking. We use a Sandy Bridge-based cluster with 4×8 core nodes (*ada*, from IDRIS) as well as a Xeon-based SGI Altix ICE 8200 with 2×4 core nodes (*hyperion*, from CALMIP) Intel BLAS (MKL) and MPI libraries are used and, because asynchronous communications only progressed inside MPI calls, we use a progress thread [14] to force `MPI_TEST` calls every milisecond.

Figures 8(a) and 8(b) show the Gantt-charts of executions of a dense partial right-looking LU factorization on a front of size $nfront = 10000$ with $nproc = 8$ MPI processes, with a number of pivots to be eliminated following *eqFlops* and *eqRows*, respectively. We can see that Fig. 8(a) is very similar to what our model predicted (See Fig. 3). Moreover, we can see on Fig. 8(b) that the fact of respecting *eqRows* in the RL variant makes the workers wait much less than in the *eqFlops* case, which confirms the observations made thanks to our models.



(a) *eqFlops* (b) *eqRows*

Fig. 8. Gannt-chart of execution of a dense partial right-looking LU factorization on a front of size $nfront = 10000$ with $nproc = 8$ MPI processes, with a number of pivots to be eliminated either respecting *eqFlops* (on the left) or *eqRows* (on the right), on a shared-memory node (to validate the communication-less model).

Other experiments with real-life Gantt charts confirmed that *eqRows* is more adapted to RL and *eqFlops* is more adapted to LL. However, and as mentioned before, due to the fact that computations on the master (that partly uses BLAS2) are slower than on the workers, *eqFlops* (in case of LL) has to be slightly modified and was replaced by *eqTime*, such that $\frac{flops_{master}}{GFlopsrate_{master}} = \frac{flops_{worker}}{GFlopsrate_{worker}}$.

Table 1 confirms the interest of a tree-based pipelined *IBcast* algorithm. It also illustrates the interest of using two levels of panels. In all cases, we used a RL algorithm for internal panels, that was observed to be more efficient than LL on small blocks. Also, and as predicted in the models, *eqFlops* (and *eqTime*) led to bad results for RL; this is why we use *eqRows* in that table. Remark that, although *eqTime* would have been better suited to LL, we used *eqRows* even for LL in order to be able to compare the times of RL and LL on a front with the same characteristics.

Table 1. Influence of *IBcast* and of double-blocking on the factorization time (seconds) of a front, for RL and LL variants on the most external panels; "-" in column *npan2* indicates that a single level of panels is used.

Machine	*nfront*	*nproc*	(*ncores*)	*IBcast* tree	*npan1*	*npan2*	RL	LL
ada	100000	64	(512)	No *IBcast*	32	-	35.7	29.8
ada	100000	64	(512)	depth 2	32	-	22.8	26.2
ada	100000	64	(512)	binary	32	-	21.8	22.0
ada	100000	64	(512)	binary	64	-	21.2	21.1
ada	100000	64	(512)	binary	32	64	20.5	19.8
hyperion	64000	8	(64)	binary	32	-	203	204
hyperion	64000	8	(64)	binary	128	-	117	110
hyperion	64000	8	(64)	binary	64	128	97	93

Table 2 shows the impact of the asynchronous broadcast algorithm on the performance for a generalized frontal matrix with a binary *IBcast* tree when two levels of panels are used. It is interesting to note that *IBcast* gains are larger when more cores are used per process, showing that communications become more critical in that case. When considering the factorization of an entire sparse matrix in a limited-memory environment [16], more workers have to be mapped on each front of the assembly tree. On 128 MPI processes of *hyperion*, on the factorization of an entire sparse matrix arising from a 3D finite-difference Laplacian problem

Table 2. Influence of *IBcast* on *hyperion* with *nfront* =*npiv* =64000. Factorization times in seconds.

Cores	Cores/ MPI process	Without	With
64	1	1702	1341
512	8	1380	404

on a 128^3 grid, we observed a time reduction from 805 to 505 s thanks to *IBcast* (see [17] for further results).

5 Conclusion

We modeled a dense asynchronous kernel for multifrontal factorizations, targeting large matrices and large numbers of cores. We studied both communication and computation aspects. The approach allows for standard threshold numerical pivoting, and can be integrated in a fully asynchronous environment with dynamic, distributed schedulers. Such an environment is precisely the one of the MUMPS solver [3], on which this work was shown to have a strong performance impact.

In the future, we plan to further optimize multithreaded kernels (inside each MPI process), and optimize the communication volume when remapping needs to be done between two successive pipelined factorizations. Topology-aware broadcast algorithms [18] are also a promising approach to further improve the cost of broadcasting factorized panels. Moreover, comparisons between models and experiments of dense factorizations will allow us to improve the performance results on full sparse multifrontal factorizations. Comparison with techniques used in HPL[1] would also be interesting.

Acknowledgement. This work was granted access to the HPC resources of CALMIP under the allocation 2013-0989 and GENCI/IDRIS resources under allocation x2013065063.

References

1. Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. J. Phys. Conf. Ser. **180**(1), 012037 (2009)
2. Amestoy, P.R., Buttari, A., Duff, I.S., Guermouche, A., L'Excellent, J.-Y., Uçar, B.: The multifrontal method. In: Padua, D. (ed.) Encyclopedia of Parallel Computing, pp. 1209–1216. Springer, Heidelberg (2011)
3. Amestoy, P.R., Duff, I.S., Koster, J., L'Excellent, J.-Y.: A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM J. Matrix Anal. Appl. **23**(1), 15–41 (2001)
4. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency Comput.: Pract. Experience **23**(2), 187–198 (2011). Special Issue: Euro-Par 2009
5. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, A., Herault, T., Kurzak, J., Langou, J., Lemarinier, P., Ltaief, H., Luszczek, P., Yarkhan, A., Dongarra, J.J.: distibuted dense numerical linear algebra algorithms on massively parallel architectures: DPLASMA. In: Proceedings of the 25th IEEE International Symposium on Parallel & Distributed Processing Workshops and Ph.D. Forum (IPDPSW'11). PDSEC 2011, pp. 1432–1441. Anchorage, USA (2011)

[1] http://www.netlib.org/hpl/.

6. Bosilca,G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., Dongarra, J.: DAGuE: A generic distributed DAG engine for high performance computing. In: 16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'11) (2011)
7. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Comput. **35**(1), 38–53 (2009)
8. Choi, J., Dongarra, J.J., Ostrouchov, L.S., Petitet, A.P., Walker, D.W., Whaley, R.C.: Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. Sci. Program. **5**(3), 173–184 (1996)
9. Desprez, F., Dongarra, J.J., Tourancheau, B.: Performance complexity of LU factorization with efficient pipelining and overlap on a multiprocessor. LAPACK working note 67, Computer Science Department, University of Tennessee, Knoxville, Tennessee (1994)
10. Duff, I.S., Erisman, A.M., Reid, J.K.: Direct Methods for Sparse Matrices. Oxford University Press, London (1986)
11. Duff, I.S., Reid, J.K.: The multifrontal solution of unsymmetric sets of linear systems. SIAM J. Sci. Stat. Comput. **5**, 633–641 (1984)
12. Golub, G.H., Van Loan, C.F.: Matrix Computations, 2nd edn. Johns Hopkins Press, Baltimore (1989)
13. Grigori, L., Demmel, J., Xiang, H.: CALU: a communication optimal LU factorization algorithm. SIAM J. Matrix Anal. Appl. **32**(4), 1317–1350 (2011)
14. Hoefler, T., Lumsdaine, A.: Message progression in parallel computing - to thread or not to thread? In: IEEE International Conference on Cluster Computing, pp. 213–222 (2008)
15. Liu, J.W.H.: The multifrontal method for sparse matrix solution: theory and practice. SIAM Rev. **34**, 82–109 (1992)
16. Rouet, F.-H.: Memory and performance issues in parallel multifrontal factorizations and triangular solutions with sparse right-hand sides. Ph.D. thesis, Institut National Polytechnique de Toulouse, October 2012
17. Sid-Lakhdar, W.M.: Scaling multifrontal methods for the solution of large sparse linear systems on hybrid shared-distributed memory architectures. Ph.D. dissertation, ENS Lyon (2014, In preparation)
18. Solomonik, E., Bhatele, A., Demmel, J.: Improving communication performance in dense linear algebra via topology aware collectives. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2011, pp. 77:1–77:11. ACM, New York (2011)
19. Solomonik, E., Demmel, J.: Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part II. LNCS, vol. 6853, pp. 90–109. Springer, Heidelberg (2011)
20. Toledo, S.: Locality of reference in lu decomposition with partial pivoting. SIAM J. Matrix Anal. Appl. **18**(4), 1065–1081 (1997)
21. Wadsworht, D.M., Chen, Z.: Performance of MPI broadcast algorithms. In: Proceedings of the 22nd International Parallel and Distributed Processing Symposium (IPDPS 2008), pp. 1–7 (2008)