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] Univ. of Toulouse, INPT(ENSEEIHT)-IRIT, France
[2] Univ. of Lyon, Inria and LIP (CNRS, ENS Lyon, Inria, UCBL), France
[3] Lawrence Berkeley National Laboratory, USA
[4] Univ. of Lyon, ENS Lyon and LIP (CNRS, ENS Lyon, Inria, UCBL), France

Abstract. To solve sparse linear systems 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 [1] are widely used to solve sparse systems of equations of the form $Ax = b$, where A is a sparse matrix. 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 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. 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 [6], or DAG-based tiled algorithms [5], in both shared and distributed-memory environments. Recent asynchronous approaches usually rely on a task scheduling engine [3,4] 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 [9]). 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 [6].

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 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 [2]. We are thus interested in analyzing and pushing the limits of this distribution. Because of the complexity of analytical models [11] due to the discrete nature of the phenomena, we have developed a simulator that models parallel executions for standard blocked variants (so-called left and right-looking [8]). Although cyclic pipelined factorizations have been modeled in the past [7], 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). For ScaLAPACK that relies on a 2D block cyclic distribution, right-looking is preferred over left-looking [6]; however, with our 1D technique, our conclusion is different, as will be illustrated in this paper.

The paper is organized as follows. In Section 2, we first study the theoretical behaviour of left-looking and right-looking variants for both one and two levels blocked algorithms in the context of a network with infinite bandwith. Communication models are then studied in more detail in Section 3 where we 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 [2] and to illustrate in Section 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 pipelined dynamic asynchronous 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 *slaves*) 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 slaves 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 next panel will start (left-looking). In parallel, the slaves update all their rows at each panel reception. For the sake of clearness, we consider that CB rows are uniformly distributed over the slaves. We have designed both complicated analytical models with MAPLE [11] and a simpler Python simulator, which is designed to model both computations and communications. In order to illustrate some intrinsic properties of the algorithms, we first consider that communications take place on a network with infinite bandwidth γ and that computations take place at a constant GFlops rate α.

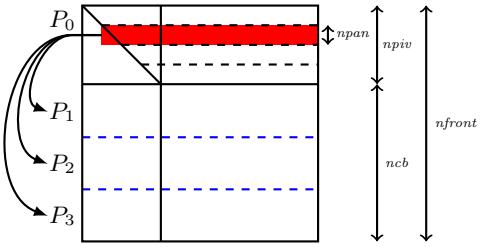
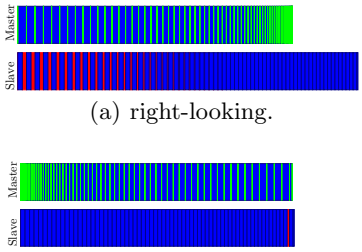


Fig. 1. Partial factorization of a front of order *nfront*, with *npiv* variables to eliminate by panels of *npan* rows, and $ncb = nfront - npiv$ rows to be updated.



(a) right-looking.

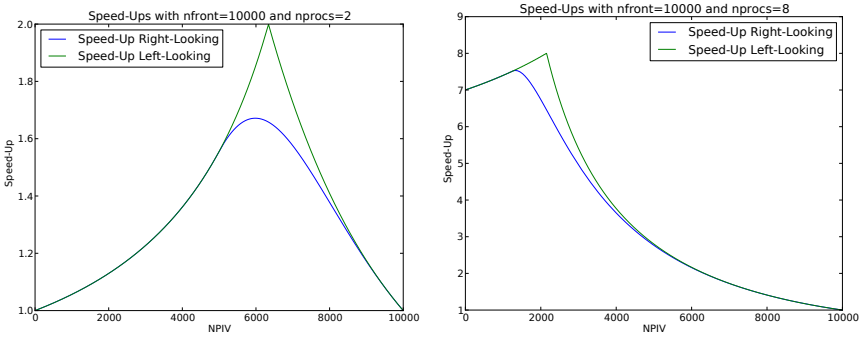
(b) left-looking.

Fig. 2. Gantt chart of the RL and LL algorithms. Factorization in green, updates in blue and idle times in red.

Right-Looking and Left-Looking algorithms. To better characterize the main properties of our algorithms, let us consider an ideal situation where the number of operations on the master is equal to that of each slave. **Fig. 2** 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 slaves 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 slave (all slaves theoretically behave the same way). **Fig. 2(a)** clearly illustrates the weakness of the RL approach. Given that *npiv* balances the total amount of work (flops) between master and slaves, one would expect all processes to finish at the same time. However, the slaves finish much later because they have idle phases that sum up to the gap between master and slaves completion times. As the consumption of factored panels is critical on the slaves, 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. 2(b)**.

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. 3 (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: slaves

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 slave. 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 slaves 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. 3. 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. 3 (b)**, the maximum speed-ups of RL and LL tend to get closer. LL reaches its maximum speed-up when all processes (master and slave) get the same amount of computations *Flops equilibrium* (*eqFlops*), so that neither the master nor the slaves are bottlenecks to each other. On the other hand, RL reaches its maximum speed-up when all processes (master and slave) are roughly assigned the same number of rows *Rows equilibrium* (*eqRows*). This latter approximation relies on the fact that this keeps slaves always busy, leading to a speed-up at least equal to $nproc - 1$.

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 [13]. 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 slaves 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 slaves, we slightly modify the *eqFlops* ideal $\frac{npiv}{nfront}$ ratio (for LL) so that $\frac{flops_{master}}{GFlops\ rate_{master}} = \frac{flops_{slave}}{GFlops\ rate_{slave}}$.

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, the flexibility of multifrontal methods can be used to split the initial

front into a chain of fronts that we can choose to have an optimal ratio $\frac{npiv}{nfront}$ (possibly except the last one). Simple models of such chains were discussed in [10] and have been revisited in [11]. **Fig. 4** reports 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 *npiv* (compared to **Fig. 3**) 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.

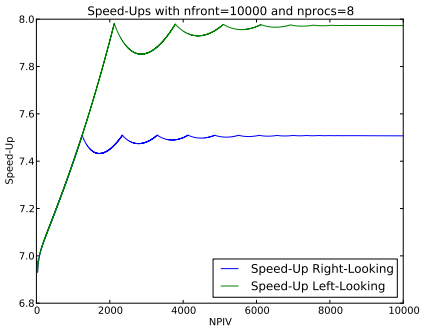


Fig. 4. Simulated generalized 1D factorization (*nfront* =10000,*nproc* =8) with varying *npiv*. LL (resp. RL) uses *eqFlops* (resp. *eqRows*).

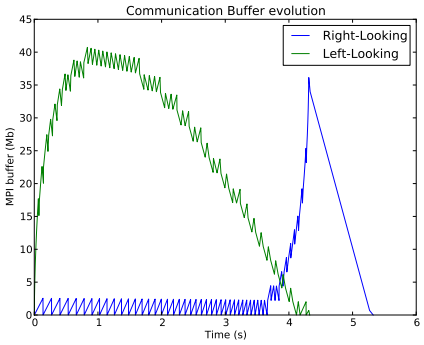


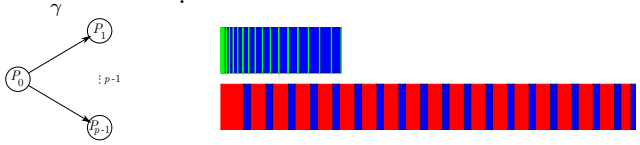
Fig. 5. Amount of data sent but not ready to be received using RL an LL algorithms with *eqFlops* (*nproc* =8, *nfront* =10000, *npiv* =2155).

3 Modeling communications

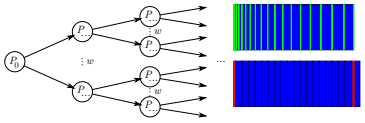
Memory for communication. Assuming that sends are performed as soon as possible, **Fig. 5** represents the evolution of the memory utilization in the send buffer for LL and RL factorizations, both with *eqFlops*. This send buffer is the place in memory where panels computed by the master are temporarily stored (contiguously) and sent using non-blocking primitives; when the slaves start receiving, send buffer can often be freed. Most of the time, the buffer in the RL variant only contains one panel, immediately consumed by the slaves; 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 slaves, whereas the LL does. This study also shows that, in order to control buffer memory, messages should *not* be sent as soon as possible.

Limited bandwidth and asynchronous collective communications. When increasing the number of processors, the communication of the panels from master to slaves becomes critical. Many efficient broadcast implementations exist for MPI [14], 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 simple w-ary broadcast tree (**Fig. 6(b)**). The Gantt charts of **Fig. 6** show the impact of the communication patterns with limited bandwidth per process. With the baseline communication algorithm, the slaves 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. 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 $\frac{nfront \times npan \times (nproc-1)}{\gamma}$.



(a) Baseline algorithm (loop of Isends).



(b) Tree-based algorithm.

Fig. 6. 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).

4 Preliminary experimental results

We generalized the asynchronous factorization algorithms available in the MUMPS solver [2] 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*) as well as a Xeon-based SGI Altix ICE 8200 with 2×4 core nodes (*hyperion*)[5]. 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

[5] We acknowledge the use of computing resources from CALMIP (project 2013-0989) and IDRIS (project x2013065063).

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* led to bad results for RL; this is why we use *eqRows* in that table.

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 (here a 3D finite-difference Laplacian problem on a 128^3 grid) in a limited-memory environment [10], more slaves have to be mapped on each front of the assembly tree. On 128 MPI processes of *hyperion*, we observed a time reduction from 805 to 505 seconds thanks to *IBcast* (see [11] for further results).

Table 2. Influence of *IBcast* on *hyperion* with *nfront* =*npiv* =64000.

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

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 [2], 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 [12] are also a promising approach to further improve the cost

of broadcasting factorized panels. Comparison with HPL [6] would also be interesting.

References

1. P. R. Amestoy, A. Buttari, I. S. Duff, A. Guermouche, J.-Y. L'Excellent, and B. Uçar. The multifrontal method. In D. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1209–1216. Springer, 2011.
2. P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.
3. C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23(2):187–198, Feb. 2011.
4. G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. 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.
5. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.*, 35(1):38–53, 2009.
6. J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming*, 5(3):173–184, 1996.
7. F. Desprez, J. J. Dongarra, and B. Tourancheau. 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.
8. G. H. Golub and C. F. Van Loan. *Matrix Computations. 2nd ed.* Johns Hopkins Press, Baltimore, MD., 1989.
9. L. Grigori, J. Demmel, and H. Xiang. CALU: A communication optimal LU factorization algorithm. *SIAM J. Matrix Analysis Applications*, 32(4):1317–1350, 2011.
10. F.-H. Rouet. *Memory and performance issues in parallel multifrontal factorizations and triangular solutions with sparse right-hand sides.* PhD thesis, Institut National Polytechnique de Toulouse, Oct. 2012.
11. W. M. Sid-Lakhdar. *Scaling multifrontal methods for the resolution of large sparse linear systems on hybrid shared-distributed memory architectures.* Ph.D. dissertation, ENS Lyon, 2014. In preparation.
12. E. Solomonik, A. Bhatele, and J. Demmel. 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 '11, pages 77:1–77:11, New York, NY, USA, Nov. 2011. ACM.
13. S. Toledo. Locality of reference in lu decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, 1997.
14. D. M. Wadsworht and Z. Chen. Performance of MPI broadcast algorithms. In *Proceedings of 22nd International Parallel and Distributed Processing Symposium (IPDPS'08)*, pages 1–7, 2008.

[6] http://www.netlib.org/benchmark/hpl/