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.

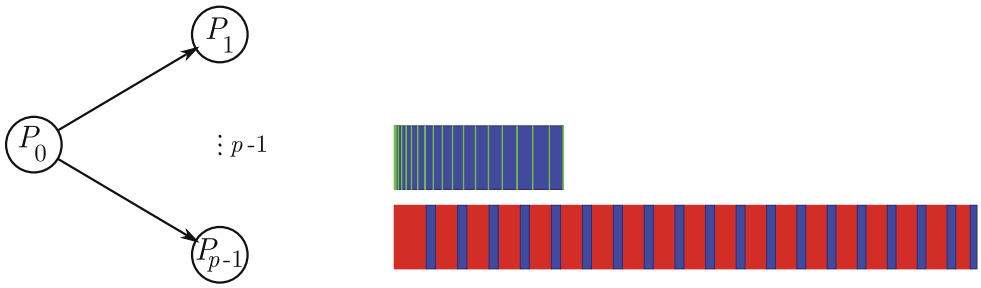
3 Modeling Communications

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).

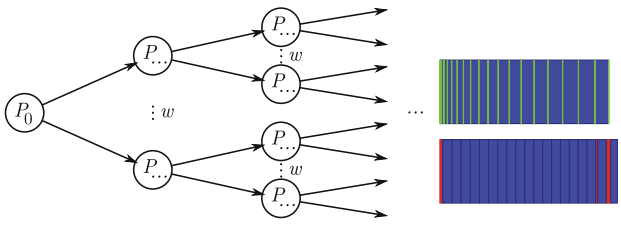
Limited Bandwidth and Asynchronous Collective Communications. We observed experimental results to be very similar to those of the model, as long as the ratio between computations and communications remains large enough (*nfront* relatively large compared to *nproc*). Strong scaling, i.e., increasing *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.

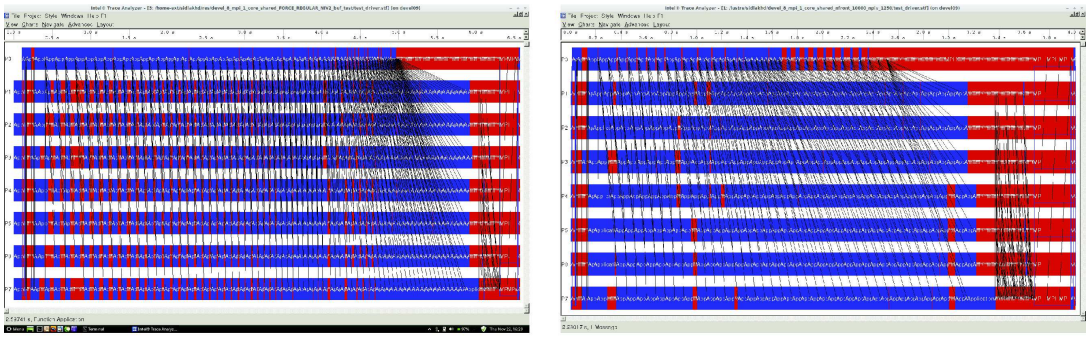
Fig. 7. Influence of the IBcast communication pattern with a limited bandwidth per proc (γ=1.2 Gb/s, α=10 GFlops/s) on LL algorithm with $nfront = 10000$, $npan = 32$, $nproc = 32$ and $npiv$ chosen to balance work (idle times in red) (Color figure online).

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)