A Preliminary Out-of-core Extension of a Parallel Multifrontal Solver

Emmanuel Agullo[1], Abdou Guermouche[2], and Jean-Yves L'Excellent[3]

[1] LIP-ENS Lyon, France
[2] LaBRI, Bordeaux, France *
[3] INRIA and LIP-ENS Lyon, France

Abstract. The memory usage of sparse direct solvers can be the bottleneck to solve large-scale problems. This paper describes a first implementation of an *out-of-core* extension to a parallel multifrontal solver (MUMPS). We show that larger problems can be solved on limited-memory machines with reasonable performance, and we illustrate the behaviour of our parallel *out-of-core* factorization. Then we use simulations to discuss how our algorithms can be modified to solve much larger problems.

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). For large sparse problems, direct approaches often require a large amount of memory, that can be larger than the memory available on the target platform (cluster, high performance computer, ...). In order to solve increasingly large problems, *out-of-core* approaches are then necessary, where disk is used to store data that cannot fit in physical main memory.
Although several authors have worked on sequential or shared-memory *out-of-core* solvers [1, 9, 17], sparse *out-of-core* direct solvers for distributed-memory machines are less common. In this work, we aim at extending a parallel multifrontal solver (MUMPS, for MUltifrontal Massively Parallel Solver, see [3]), in order to enable the solution of larger problems, thanks to *out-of-core* approaches. Recent contributions by [13] and [14] for uniprocessor approaches pointed out that multifrontal methods may not fit well an *out-of-core* context because large dense matrices have to be processed, that can represent a bottleneck for memory; therefore, they prefer left-looking approaches (or switching left-looking approaches). However, in a parallel context, increasing the number of processors can help keeping such large frontal matrices in-core. Note also that another type of approach is based on virtual memory and system paging, that can be controlled by low level mechanisms [8] in relation with the application and provide better performance than default LRU mechanisms. However, such approaches are very closely related to the operating system and are not adapted when designing portable codes.

* This work was done during an INRIA post-doctoral position at ENSEEIHT-IRIT, Toulouse, France

This paper is organized as follows. After a quick description of the memory management in multifrontal methods (Section 2), we present in Section 3 an approach to store the sparse factors L and U to disk. We will observe that this approach allows us to treat larger problems with a given memory, or the same problem with less memory. In Section 3.4, both a synchronous approach (writing factors to disk as soon as they are computed) and an asynchronous approach (where factors are copied to a buffer and written to disk only when the buffer is full) are analyzed, and compared to the in-core approach on a platform with a large amount of memory. Finally, in order to process much larger problems, we present in Section 4 simulation results where we suppose that the active memory of the solver is also stored on the disk and study how the overall memory can further be reduced. This study is the basis to identify the bottlenecks of our approach when confronted to arbitrarily large problems.

2 Memory management in a parallel multifrontal method

In multifrontal methods, the task dependencies are represented by a so-called assembly tree [6, 10], that is processed from bottom to top during the factorization. At each node of the tree is associated a so-called *frontal matrix*, or *front*, and a task consisting in the partial factorization of the frontal matrix. The partial factorization produces a Schur complement, or *contribution block*, which will be used to update the frontal matrix of the parent node (see [2], for example, for more details). This leads to three areas of storage, one for the factors, one for the contribution blocks, and another one for the current frontal matrix [2]. The active memory (as opposed to the memory for the factors) then corresponds to the sum of the contribution blocks memory (or stack memory) and the memory for the current active matrix. During the factorization process, the memory required for the factors always grows while the stack memory that contains the contribution blocks varies: when the partial factorization of a frontal matrix is performed, a contribution block is stacked which increases the size of the stack; on the other hand, when the frontal matrix of a parent is formed and assembled, the contribution blocks of the children nodes can be discarded and the size of the stack decreases[4].
From the parallel point of view, the parallel multifrontal method as implemented in MUMPS uses a combination of static and dynamic scheduling approaches. Indeed, a first partial mapping is done statically (see [4]) to map some of the tasks to the processors. Then, for parallel tasks corresponding to large frontal matrices of the assembly tree, a master task is in charge of the elimination of the so-called fully summed rows, while dynamic scheduling decisions are used to select the processors in charge of updating the rest of the frontal matrix (see Figure 1). Those decisions are taken to balance workload, possibly under memory constraints (see [5]).

3 Out-of-core multifrontal approach

3.1 Preliminary study

In the multifrontal method, the factors produced during the factorization step are not reused before the solution step. It then seems natural to first focus on writing *them* to disk.

[4] In parallel, the contribution blocks management may differ from a pure stack mechanism.

Thus, we present a preliminary study which aims at evaluating by how much the in-core memory can be reduced by writing the factors to disk during the factorization. To do so, we simulated an *out-of-core* treatment of the factors: we free the corresponding memory as soon as each factor is computed. Of course the solution step cannot be performed as factors are definitively lost, but freeing them allowed to analyze real-life problems on a wider range of processors (in this initial study).
We measure the size of the new peak of memory (which actually corresponds to the *active memory* peak) and compare it to the one we would have with an *in-core* factorization (*i.e.* the *total memory peak*). In a distributed memory environment, we are interested in the maximum peak obtained over all the processors as this value represents the memory bottleneck.
For a small number of processors, we observe that the active memory is much smaller than the total memory. In other words, if factors are written to disk as soon as they are computed, only the active memory remains *in-core* and the memory requirements decrease significantly (up to 80 % in the sequential case).

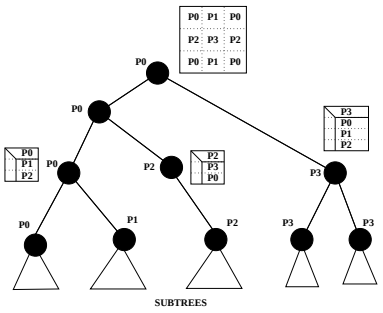


Fig. 1. Example of the distribution of an assembly tree over four processors.

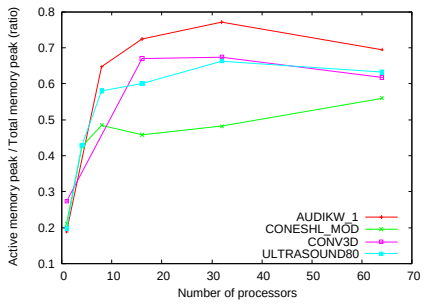


Fig. 2. Ratio of active and total memory peak on different number of processors for several large problems (`METIS` is used as the reordering technique).

On the other hand, when the number of processors increases, the peak of the active memory decreases more slowly than the total memory as shown in Figure 2. For example, on 64 processors, the active memory peak reaches between 50 and 70 percent of the peak of total memory. In conclusion, on platforms with small numbers of processors, an *out-of-core* treatment of the factors will allow us to process significantly bigger problems; the implementation of such a mechanism is the object of Section 3.2. Nevertheless, either in order to further reduce memory requirements on platforms with only a few processors or to have significant memory savings on many processors, we may have to treat both the factors and the active memory with an *out-of-core* scheme. This will be studied in Section 4.

3.2 Out-of-core management of the factors

The performance of I/O mechanisms are essential and impact directly the performance of the whole application. Neither MPI-IO [16] (because files are not shared by processors

in our case) nor FG [7] (our I/O threads do not interfere with each other) match our purpose. Both `AIO`, an asynchronous I/O mechanism optimized at the kernel level, and the recent Fortran 2003 asynchronous I/O layer were not available on our target platform (see Section 3.3). We finally used the standard C I/O routines *fread/fwrite* and *read/write* (or *pread/pwrite* when available) which are known to be efficient low-level kernels.

In the synchronous I/O scheme, the factors are directly written with a synchronous scheme using the standard I/O subroutines (either *fread/fwrite* or *read/write*). In the asynchronous I/O scheme, we associate with each MPI process of our application an I/O thread in charge of all the I/O operations. This allows us to overlap the time needed by I/O operations with computations. The I/O thread is designed over the standard `POSIX` thread library (pthread library). The communication and the synchronization between the computational thread and the I/O thread are designed using semaphore mechanisms. The communication scheme between the two threads is described in Figure 3. Each time an I/O operation has

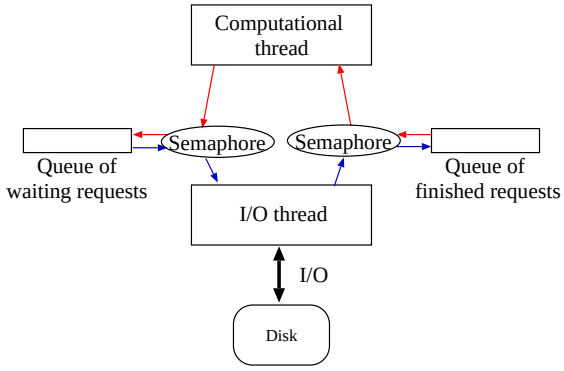


Fig. 3. Thread communication scheme.

to be performed, the computational thread posts an I/O request and inserts it into the *queue of waiting requests*. Concerning the I/O thread, it treats the I/O requests in the *queue of waiting requests* using a FIFO strategy. Once an I/O request is finished, it is inserted in the *queue of finished requests* by the I/O thread. The computation thread can then remove it from this queue when checking for the completion of the request.

Together with the two I/O mechanisms described above, we designed a buffered I/O scheme (that can be either synchronous or asynchronous). This approach relies on the fact that we want to free the memory occupied by the factors as soon as possible without necessarily waiting for the completion of the corresponding I/O. Thus, and in order to avoid a complex memory management in a first approach, we added a buffer where factors are copied before they are written to disk. The buffer is divided into two parts so that while an asynchronous I/O operation is occurring on one part, factors that are being computed can be stored in the other part (double buffer mechanism allowing the overlap of I/O operations with computation).

3.3 Experimental environment

In order to study the impact of the proposed mechanisms, we now experiment with them on several problems (see Table 1) extracted from either the PARASOL collection[5] or coming from other sources. The tests have been performed on the IBM SP system of IDRIS[6] composed of several nodes of either 4 processors at 1.7 GHz or 32 processors at 1.3 GHz. On this machine, we have used from 1 to 128 processors with the following memory constraints: we can access 1.3 GB per processor when asking for more than 128 processors, 3.5 GB per processor for 17-64 processors, 4 GB for 2-16 processors, and 16 GB on 1 processor.

Matrix	Order	NZ	Type	$nnz(L\|U) \times 10^6$	Description
AUDIKW_1	943695	39297771	SYM	1368.6	Automotive crankshaft model (PARASOL)
CONESHL_mod	1262212	43007782	SYM	790.8	provided by SAMTECH; cone with shell and solid element connected by linear constraints with Lagrange multiplier technique
CONV3D64	836550	12548250	UNS	2693.9	provided by CEA-CESTA; generated using AQUILON (http://www.enscpb.fr/master/aquilon)
ULTRASOUND80	531441	330761161	UNS	981.4	Propagation of 3D ultrasound waves, provided by M. Sosonkina, larger than ULTRASOUND3

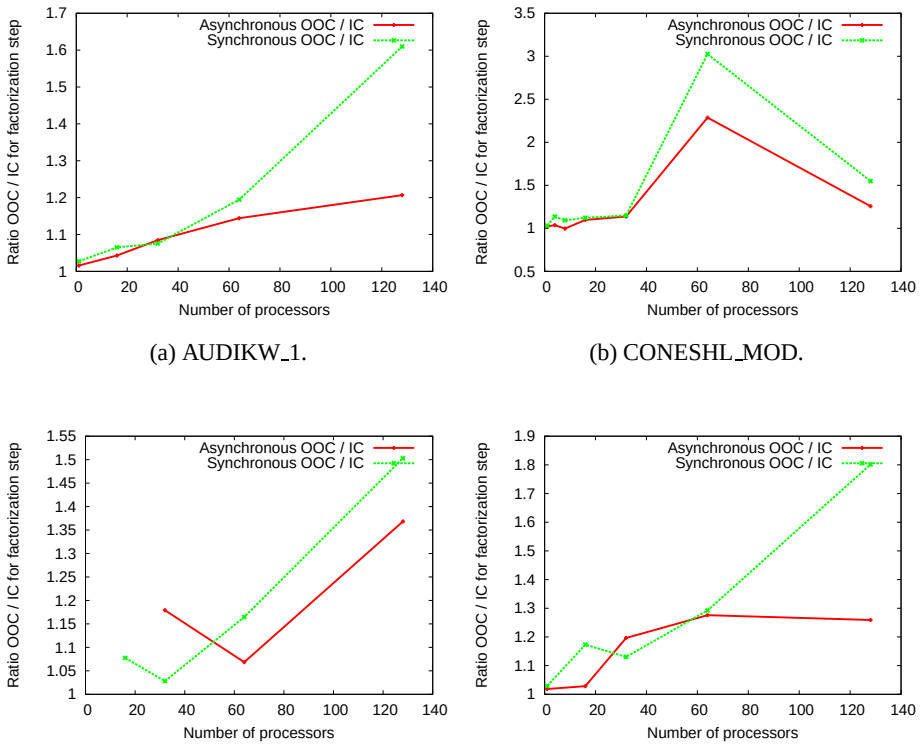
Table 1. Test problems.

By default, we used the METIS package [12] to reorder the matrices and thus limit the number of operations and fill-in arising in the subsequent sparse factorization. The results presented in the following sections have been obtained using the dynamic scheduling strategy proposed in [5].
The I/O system used is the `IBM GPFS` [15] filesystem. With this filesystem it was not possible to write files on disks local to the processors and some performance degradation was observed when several processors write/read an amount of data simultaneously to/from the filesystem: we observed a speed-down between 5 and 50 from 2 to 64 processors when each processor writes a block of 800 MBytes. Finally, it is important to note that we chose to run on this platform because it allows us to run large problems *in-core* and thus compare *out-of-core* and *in-core* approaches (even if the behaviour of the filesystem is not optimal for performance).

3.4 Experiments

First, we have been able to observe that for a small number of processors we use significantly less memory with the *out-of-core* approach: the total memory peak is replaced by the active memory peak, with the improvement ratios of Figure 2. Thus the factorization can be achieved on limited-memory machines.
We now focus on performance issues and report in Figure 4 a comparative study of the *in-core* case, the synchronous *out-of-core* scheme and the asynchronous buffered scheme, when varying the number of processors.
Note that for the buffered case, the size of the I/O buffer is set to twice the size of the largest factor block (to have a double buffer mechanism). As we can see, the performance

[5] `http://www.parallab.uib.no/parasol`
[6] Institut du Développement et des Ressources en Informatique Scientifique



(a) AUDIKW_1. (b) CONESHL_MOD.

(c) CONV3D64. (d) ULTRASOUND80.

Fig. 4. Execution times (normalized with respect to the *in-core* case) of the synchronous and asynchronous I/O schemes.

of the *out-of-core* schemes is indeed close to the *in-core* performance for the sequential case (note that we were not successful in running the CONV3D64 matrix on 1 processor even with the *out-of-core* scheme because the active memory requires more than 16 GB). The *out-of-core* schemes are at most 20% slower than the *in-core* case while they need an amount of memory that can be 80 percent smaller as shown in Figure 2 for one processor. Concerning the parallel case, we observe that with the increase of the number of processors, the gap between the *in-core* and the *out-of-core* cases increases. The main reason is the performance degradation of the I/O with the number of processors that we mentioned at the end of Section 3.3. In order to avoid this problem, we have experimented with the smallest of our large test problems on a machine with local disks. In this case, we do not have such a performance degradation, as shown in Figure 5; on the contrary, the *out-of-core* schemes perform as well or even better than the *in-core* one (cache effects resulting from freeing the factors from main memory and using always the same memory area for active frontal matrices). Finally, concerning the comparison of the *out-of-core* schemes, we can see that the asynchronous buffered approach performs better than the synchronous one. However, it has to be noted that even in the synchronous scheme, the system allocates

data in memory that also allows to perform *I/O* asynchronously, in a way that is hidden to the application. Otherwise, the performance of the synchronous approach would be much worse.

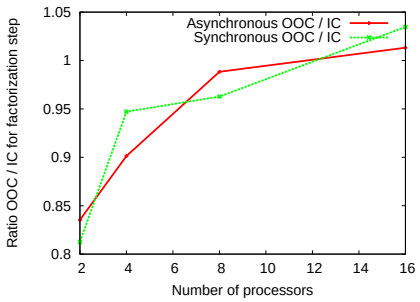


Fig. 5. Performance of the *out-of-core* factorization on a machine (`CRAY XD1` system at `CERFACS`) with local disks for the CONESHL_MOD matrix on different number of processors.

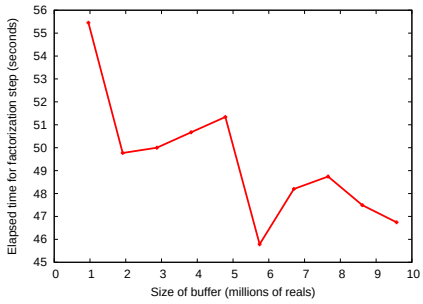


Fig. 6. Performance of the *out-of-core* factorization on 32 processors for the CONESHL_MOD matrix with respect to the size of the I/O buffer.

We artificially decreased the size of the I/O buffer on the matrix CONESHL_MOD on 32 processors (default size was 9.5 million reals for this matrix). We can see from Figure 6 that the factorization time decreases when the size of the buffer increases. Indeed, in our strategy, the nodes that cannot fit into the buffer are written synchronously to disk, slowing down the factorization. (Note that in all cases the size of the buffers ensures a sufficient granularity for the performance of I/O.)

Concerning the solution phase, the size of the memory will generally not be large enough to hold all the factors. Thus, factors have to be read from disk, and the I/O involved increase significantly the time for solution. Note that we use a basic demand-driven scheme, relying on the synchronous low-level I/O mechanisms from Section 3.2. We have observed that the performance of the *out-of-core* solution step is often more than 10 times slower than the *in-core* case. Although disk contention might be an issue on our main target platform in the parallel case, the performance of the solution phase should not be neglected; it becomes critical in an *out-of-core* context and prefetching techniques in close relation with scheduling issues have to be studied. This is the the object of current work by the MUMPS group in the context of the PhD of Mila Slavova.

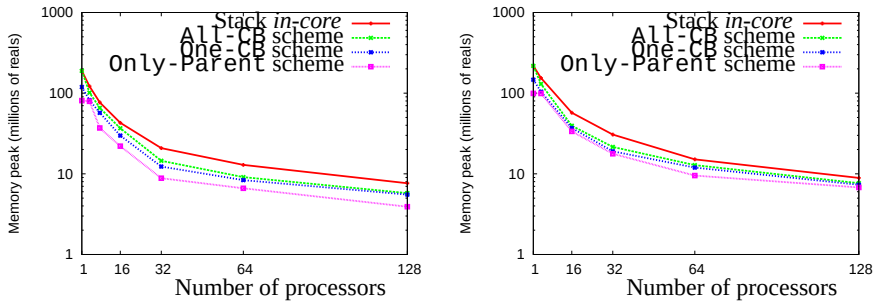
4 Simulation of an *out-of-core* stack memory management

In Section 3, we presented a first *out-of-core* approach for the parallel multifrontal factorization, consisting in writing factors to disk as soon as possible. The results obtained have shown the potential of the approach and how larger problems can be treated. However this approach also has certain limitations and the stack memory now becomes the limiting factor. Therefore, the next step is to manage the stack of contribution blocks with an

out-of-core scheme, where a contribution block may be written to disk as soon as it is produced, and read from disk when needed (either with a prefetching mechanism or with a demand-driven scheme).
With the objective to assess the potential of such an approach, we perform in this section simulations with various scenarios for the stack management:

– `All-CB` ***out-of-core*** **stack memory.** In this scheme, we suppose that during the assembly step of an active frontal matrix, all the contribution blocks corresponding to its children have been prefetched in memory. Thus, the assembly step is processed as in the *in-core* case.
– `One-CB` ***out-of-core*** **stack memory.** In this scheme, we suppose that during the assembly step of an active frontal matrix, only one contribution block corresponding to one of its children is loaded in memory, while the others stay on disk. Thus we interleave the assembly steps with I/O operations.
– `Only-Parent` ***out-of-core*** **stack memory.** In this scheme, we suppose that during the assembly step of an active frontal matrix, no contribution block is loaded in memory. Thus, the assembly step is done in an *out-of-core* way. Note that the implementation of such a strategy will not be efficient at all since the assembly steps are not very costly and there is no way to overlap I/O operations with computations. This strategy corresponds to an ideal scenario concerning the size of the in-core memory.

Note that for the three scenarios, we suppose that a contribution block is written to disk as soon as it is computed. In addition, we assume that all the active frontal matrices remain in memory until the end of their factorization.
Results and discussion. Although we experimented with several matrices, we only illustrate in Figure 7 the memory behaviour using the different *out-of-core* memory management strategies and *in-core* case for two test problems on different numbers of processors.



(a) CONESHL_MOD. (b) ULTRASOUND80.

Fig. 7. Memory behaviour with different memory management strategies on different numbers of processors for two large problems (`METIS` is used as reordering technique).

As expected, we see that the strategies for managing the stack *out-of-core* provide a reduced memory requirement. We also observe that the `Only-Parent` *out-of-core* stack

memory management is the one that best decreases the memory needed by the factorization. Although this strategy might not be good for performance, it is here to provide some insight on the best we can do with our assumptions and with the current version of the code. One interesting phenomenon we observed is that the *out-of-core* stack memory management strategies give better results with symmetric matrices (see Figure 7(a)) than with unsymmetric ones (see Figure 7(b)). For unsymmetric matrices and on large numbers of processors, the bottleneck is very often due to the treatment of master tasks (holding the variables that need to be factored when the frontal matrix is parallelized) that are bigger for unsymmetric matrices (see [3]). Since we prefer to keep these tasks in core, a variant of the splitting algorithm of [3] could be applied in a parallel context, to limit the size of those tasks. In addition, we have observed that with our assumption that an active frontal matrix (or part of it if it is distributed over several processors) has to stay in memory while being factored, it would be beneficial to reduce as much as possible the number of simultaneous active tasks on a processor. This can be done by modifying the scheduling strategies currently existing in the parallel multifrontal method.
These results illustrate that the `One-CB` approach could be a good way to design an *out-of-core* stack memory management strategy with reasonable performance. With the modifications discussed above to further decrease the memory peaks, it seems that the intrinsic limits of the sequential multifrontal method become much less critical thanks to parallelism.

5 Future work

We presented in this paper a first implementation of an *out-of-core* extension of the parallel multifrontal solver `MUMPS`. The selected approach was to drop factors from memory as soon as they are computed and to overlap the I/O operations as much as possible with computations. We illustrated the good behaviour of this approach on a small number of processors and its limitations on larger ones, while first experiments on machines with local I/O showed no significant I/O overhead during the factorization. Nevertheless we noticed that low-level *I/O* mechanisms have to be designed with care as the system is not tuned to *I/O*-intensive and large memory applications.
One key point that must be studied is the design of efficient *out-of-core* stack memory management schemes based on the results presented in Section 4. In this context, the contribution blocks can be considered as read-once/write-once data accessed with a near-to-stack mechanism (for the parallel case the accesses are more irregular). With asynchronous I/O, prefetching algorithms have to be designed. In addition, the number of contribution blocks (for the parallel case) that a processor has in memory is closely related to the scheduling decisions made; both the static and dynamic aspects of scheduling could limit the I/O volume that each processor has to perform and drive some dynamic decisions with the data that are available in memory (for example, give a priority to tasks that depend on/consume contribution blocks already in memory).
In order to treat larger problems where both the factors and the stack memory are *out-of-core*, we have to determine more accurately which type of tasks are responsible for the peak of memory and then to limit their size and/or the number of such tasks that are active at the same time. We have already identified some critical cases in Section 4 and should now modify our algorithms when memory usage becomes a strong priority. Furthermore, adapting the techniques described in [11] could further reduce the stack memory requirements.

We believe that in a parallel context, this study shows that there is still room before reaching intrinsic memory limits of multifrontal methods. Although it is true that large frontal matrices can be problematic in sequential (need for an out-of-core assembly and factorization), this is less the case in a parallel environment.

Acknowledgements

We are grateful to P. R. Amestoy, I. S. Duff and S. Pralet for their remarks on a preliminary version of this paper.

References

1. The BCSLIB Mathematical/Statistical Library. http://www.boeing.com/phantom/bcslib/.
2. P. R. Amestoy and I. S. Duff. Memory management issues in sparse multifrontal methods on multiprocessors. *Int. J. of Supercomputer Applics.*, 7:64–82, 1993.
3. 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.
4. P. R. Amestoy, I. S. Duff, and C. Vömel. Task scheduling in an asynchronous distributed memory multifrontal solver. *SIAM Journal on Matrix Analysis and Applications*, 26(2):544–565, 2005.
5. P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 2005. To appear.
6. C. Ashcraft, R. G. Grimes, J. G. Lewis, B. W. Peyton, and H. D. Simon. Progress in sparse matrix methods for large linear systems on vector computers. *Int. Journal of Supercomputer Applications*, 1(4):10–30, 1987.
7. T. H. Cormen, E. Riccio Davidson, and Siddhartha Chatterjee. Asynchronous buffered computation design and engineering framework generator (abcdefg). In *19th International Parallel and Distributed Processing Symposium (IPDPS'05)*, 2005.
8. Olivier Cozette, Abdou Guermouche, and Gil Utard. Adaptive paging for a multifrontal solver. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 267–276. ACM Press, 2004.
9. F. Dobrian and A. Pothen. Oblio: a sparse direct solver library for serial and parallel computations. Technical report, Old Dominion University, 2000.
10. I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.
11. A. Guermouche and J.-Y. L'Excellent. Constructing memory-minimizing schedules for multifrontal methods. *ACM Transactions on Mathematical Software*, 2005. To appear.
12. G. Karypis and V. Kumar. 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.
13. E. Rothberg and R. Schreiber. Efficient methods for out-of-core sparse Cholesky factorization. *SIAM Journal on Scientific Computing*, 21(1):129–144, 1999.
14. Vladimir Rotkin and Sivan Toledo. The design and implementation of a new out-of-core sparse Cholesky factorization method. *ACM Trans. Math. Softw.*, 30(1):19–46, 2004.
15. F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proc. of the First Conference on File and Storage Technologies*, 2002.
16. R. Takhur, W. Gropp, and E. Lusk. On implementing mpi-io portably and with high performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, 1999.
17. S. Toledo. Taucs: A library of sparse linear solvers, version 2.2, 2003. Available online at http://www.tau.ac.il/~stoledo/taucs/.