Sparse Matrix Structure for Dynamic Parallelisation Efficiency

Markus Ast[1], Cristina Barrado[2], José Cela[2], Rolf Fischer[1], Jesús Labarta[2], Óscar Laborda[2], Hartmut Manz[1], and Uwe Schulz[1]

[1] INTES Ingenieurgesellschaft für technische Software mbH, Stuttgart, Germany
[2] Universitat Politècnica de Catalunya, Barcelona, Spain

Abstract. The simulated models and requirements of engineering programs like computational fluids dynamics and structural mechanics grow more rapidly than single processor performance. Automatic parallelisation seem to be the obvious approach for huge and historic packages like PERMAS. The approach is based on dynamic scheduling, which is more flexible than domain decomposition, is totally transparent to the end-user and shows good speedups because it is able to extract parallelism where others are not. In this paper we show the need of some preparatory steps on the big input matrices for good performance. We present a new approach for blocking that saves storage and decreases the computation critical path. Also a data distribution step is proposed that drives the dynamic scheduler decisions such that an efficient parallelisation can be achieved even on slow multiprocessor networks. A final and important step is the interleaving of the array blocks that are distributed to different processors. This step is essential to expose the parallelism to the scheduler.

1 Introduction

Although the increase of single processors performance, the requirements of engineering programs (like computational fluids dynamics and structural mechanics) for bigger and bigger models grows more rapidly. Simulations tend to require more accuracy, specify finer meshes, or increase the number of simulations. Standard models to deal with are around 1 million degrees of freedom (DoF) and up to 10 millions DoF for industrial benchmarks. For this reason, computational resources (like main memory limits or CPU time) are still a limiting factor in engineering. Out-of-core capabilities are essential to solve such problems.

Since scalability of single CPU becomes more and more difficult, the solution can not rely on computers speed alone. Parallelisation of the algorithms seem to be the obvious approach. The usage of parallel languages like HPF or new environments like Java can be a good strategy for new software [4.]. However, for huge and historic packages where rewriting would be too costly, the parallelism has to be integrated with incremental steps. Domain decomposition has been the a popular way to introduce parallelism in engineering packages. In this approach, the structure is divided into several meshes that can be solved in parallel and a last stage merges the results. Solvers based on domain decomposition show good speedups [10.] but they need more effort in the assembly phase. Also the domain decomposition done for an architecture configuration usually is not appropriate for another. An alternative parallelisation strategy is applied in the PERMAS system. It is more flexible than domain decomposition because it can exploit domain grain parallelism but also finer grain parallelism [9.]. Moreover PERMAS parallelism is achieved automatically, thus, it is transparent to the programmer. In this way the whole code is parallelised (i.e. non-linear simulations or contact analysis), while others just have a parallel solver. Also PERMAS guarantees that the numerical results are independent of the number of processors used on their computation.

In this paper we evaluate how reordering and data distribution can improve the performance of PERMAS parallelisation. While the classical reordering step is applied to improve the matrix fill-in, new steps can be introduced before the actual computations in order to increase parallelism. We present the three new steps that PERMAS applies after the classical reordering step: blocking, data distribution and interleaving. The paper evaluates different heuristics and shows that the achieved speed-up is up to a 5.3 on a 8-processor SGI Origin 2000. As far as we know, other commercial out-of-core packages [7.] are only able to achieve 1.42 speedup on a 4-processor CRAY-YMP for a big problem and 3.26 speedup on 8-processor CRAY C90 for a small problem. Even the speedups of in-core parallel commercial systems [1.] are around 1.8 for small problems (from 4 to 180 thousands DoF). The paper is organized as follows. Section 2 presents the storage and parallelisation strategies adopted by PERMAS. Sections 3 and 4 detail the three preparatory steps (blocking, distribution, interleaving) and presents the measures of some simulations. Final remarks and conclusion are in section 5.

2 PERMAS Global Structure

The general purpose Finite Element (FE) system PERMAS is a commercial software with 20 years of history. Real problems of structural mechanics and fluid dynamics are the actual input data. These real problems are defined with an extremely large matrix (up to 10 millions degrees of freedom), called **hypermatrix**.

Storage. PERMAS stores the hypermatrix in a three levels structure. In the highest level, L3, we have the hypermatrix structure. Each element is either a pointer to a second level submatrix or null if all the elements of the submatrix are zero. Since the hypermatrix is symmetric, only the upper triangular of L3 is actually stored. In the second level, L2, we have again either indirections to L1 or null pointers. This two levels suppose only about a 5% of the total storage. In the last level, known as L1, we have the actual data. PERMAS maps the non-zero L1 blocks as dense arrays using the file system and handles their input/output to disk.

Parallelisation. The parallelisation strategy can be found in [2.] and here we just summarize the principal aspects. The PERMAS main module, which follows a loop-nested structure to traverse the L3 and L2 levels of the hypermatrix, generates a task for each numerical computation done over the L1 matrices. Each task is inserted on-the-fly in the Task Graph (TG). When the TG is larger than a threshold, the main module passes the control to an additional module, the **Parallel Task Manager** (PTM). The PTM contains a dynamic scheduler and sends the ready tasks to slave processors (executors) using MPI. The executors do the numerical computations using standard BLAS calls. This strategy shows several advantages. Parallelisation is done automatic and transparent to the programmer because the program structure is the same for sequential and for parallel versions. Previous (sequential) PERMAS programs can be parallelised by just changing the BLAS calls with new PTM calls. The approach exploits a finer grain parallelisation than Domain Decomposition and thus, makes possible a better load balance. It is more flexible because a same executable works for different hardware configurations (number of processors) without recompilation. Finally, the numeric results are exactly the same for sequential or parallel (with any number of processors) because the operation dependences and the execution order do not change.

Preparatory Steps. It is well known that the parallel factorization can be improved with a preparatory re**ordering** step which permutes the nodes of the FE mesh. Besides the classical reordering, that PERMAS does using a combined technique of minimum degree and nested dissection [3.6.8.], it does three more preparatory steps: blocking, data distribution and interleaving: The **blocking** step consists on dividing the hypermatrix into its three level storage hierarchy. The intuitive way to do it is to superpose a grid on top of the hypermatrix twice: a fine grid defines the blocks of level L1 and a larger grid defines those of level L2. Section 3 presents an alternative algorithm for the L1 grid. New data structures are built after blocking to represent the matrix and the elimination tree at the L1 level. We call **Plane Array (PA)** to the matrix that represent L1 structure of A. Each element PA(i,j) represents a L1 block. A zero value means that the block is not really allocated. We named **Plane Elimination Tree** (**PET**) to the elimination tree of PA. These coarser-grain data structures are only needed on preparation. During execution the PTM exploits a task-level parallelism which is of a finer grain than the elimination tree parallelism [9.]. Next step is the **distribution**. It decides the initial assignment of the L1 blocks to processors. A good distribution is a compromise between a good load balance and a reduction of the communications. Section 4 presents the tight relation of the distribution and the PERMAS dynamic scheduling. It also evaluates several distribution alternatives and shows the need of the last preparatory step, interleaving.

3 Blocking: Fixed-Sized vs. Variable-Sized

The main objective of the hypermatrix blocking is the minimisation of the required storage, but, as we will show, this is not the only issue. Fig 1. shows the hypermatrix skyline of a motivating example, where the grey area represents the non zero elements after the reordering pass. Fig 1.a presents the classical blocking strategy of PERMAS, lets call it fixed-sized. The hypermatrix is divided into square blocks by superposing a grid on top of the it.

Fixed-sized blocking is a simple and clear strategy. Here, some tuning on the size of the blocks can help to minimise the storage requirements and the I/O overhead. This is an input parameter that usually ranges from 30x30 to 128x128 elements per block. There is a compromise between small sized blocking, which reduces the L1 stored zeros and big sized blocking which reduces the number of blocks and thus minimises the I/O overhead. Fixed-sized blocking becomes a problem during parallel execution because of data dependences. The computations of L1 blocks (tasks) are subjected to the precedences of the PET. These precedences inhibit the dispatching of new computations. When the precedences are due to true dependences then they must be preserved. But precedences can be artificially created by blocking. These artificial dependences are not important on a sequential execution, but on a parallel execution they suppose longer critical paths and an increase of the computation time. These artificial dependences can disappear using variable-sized blocking.

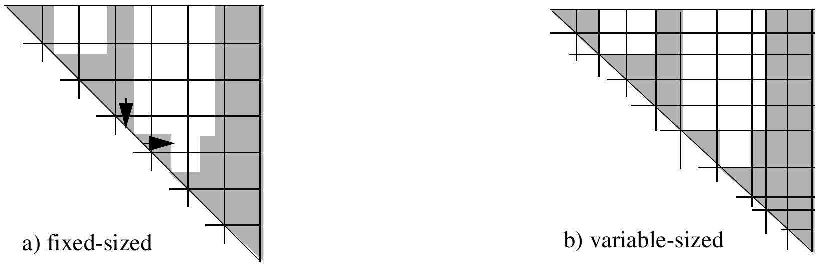


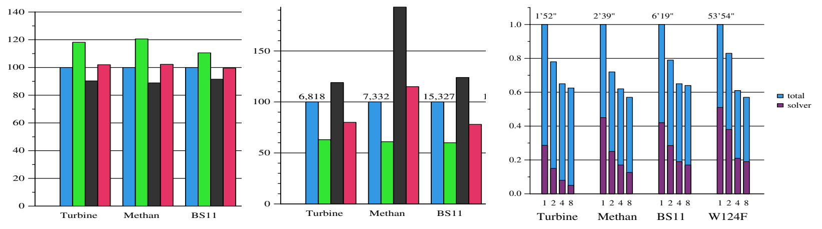
Fig 1.Blocking alternatives

For example, let us consider the fourth diagonal block of Fig 1.a, PA(4,4). At the element level, there is a decoupling point that divides the block in two parts, let us call them $\mathrm{Up}_{4,4}$ and $\mathrm{Low}_{4,4}$. The computations of the elements of the two parts can be done in parallel. Moreover, all the elements of blockPA(4,5) can be also computed in parallel with the $\mathrm{Up}_{4,4}$. Nevertheless, since the parallelism grain is the L1 level, the elements of the two parts of PA(4,4) belong to the same task and execute sequentialy. Moreover, the transitive closure of the dependences from PA(3,4) to PA(4,4) and from PA(4,4) to PA(4,5), creates the artificial dependence from PA(3,4) to PA(4,5). The variable-sized blocking proposed is illustrated in Fig 1.b. It finds the decoupling points of the hypermatrix and uses them as the vertices of the superposed virtual grid. The resulting blocks have different sizes and are not square. The variable-sized blocking decreases the number of dependences and moreover it saves disk area. On the other side, it increases the number of blocks. Fig 2. presents the results of simulating the solver part of 4 commercial benchmarks which characteristics are shown in Table 1. Numbers are given for two fixed-sized and two variable-sized blocking: Plot *fixed(16Kw)* stands for blocking into square blocks of 16Kwords (128Kbytes). Values are normalized to this first block-

Table 1. Benchmark description

Bench	Problem	DoF	total/solver time (m' s")	Mem (Mb)	I/O blocks
Turbine	Eigenvalue	66,456	1' 52" / 32"	90	9,278
Methan	ship structure	48,162	2' 39" / 1' 12"	135	8,978
BS11	rotating piece	111,057	6' 19" / 2' 38"	180	53,058
W124F	car	1,310,616	53' 54" / 27' 33"	810	209,940

ing. The plot *fixed(32Kw)* use the same strategy with double sized blocks. The two other plots, *variable(16Kw)* and *variable(32Kw)*, show the results of the variable-sized blocking when the L1 sizes are upper limited to 16Kwords or 32Kwords respectively. Fig 2.a shows the disk space needed to store the hypermatrices of the 4 benchmarks. Disk requirements are greater when blocks are bigger, because they include more zero-stored elements, while small and variable blocking covers better the shape of the hypermatrix with less space. Fig 2.b shows the number of blocks. This gives a measure of the dynamic overheads. When more (smaller) tasks are generated, more scheduling time and more input/output are expected. The Fig 2.c shows the expected execution time based on the critical path of the TG. The weight of all the tasks are considered to be equal to 1 when the block size was 16Kwords and equal to 2 for 32Kwords blocks (same for fixed than for variable sized -as the worst case-).



a) matrix size b) number of blocks c) critical path

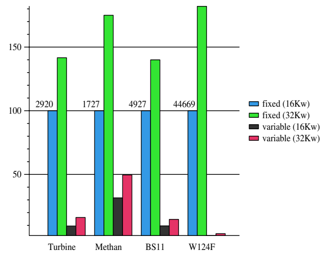
Fig 2.Fixed vs. variable sized blocking

Looking at the simulation results we conclude that variable-sized blocking can save up to 10% of the disk storage. The new storage is more fragmented and introduces a 20% more overhead on the TGM and on I/O requests. Finally it reduces the critical path length on around 90%, thus, much more parallelism is exposed. Variable-sized blocking is now integrated in the PERMAS system as an option. CPU-time improvements are shown on most applications (i.e. 20% to 40% less execution time for Turbine with 16Kw block size).

4 Data Distribution and Interleaving

The data distribution main objective is to improve load balancing while minimising communications. Several algorithms [4.11.12.], based on recursive traverse of the elimination tree, have being proposed for column-based and submatrix based approaches (i.e. subtree recursive mapping of columns). They show benefits for the static parallelised solver. In this section we show the PERMAS approach. It does a preparatory step where the L1 blocks are assigned to virtual processors. Then, the dynamic scheduler [5.] uses this informations as a suggestion, but subjected to the availability of the actual processors.

Four different data distributions are tested for a Cholesky factorization on an 8 processor architecture. Fig 3.a shows their parallel speed-ups relative to the sequential execution and Fig 3.b shows their number of messages. We choose a 10Mbps (slow) Ethernet network as the worse case for message passing, in order to show that an efficient parallelisation is only possible with a good data distribution.The random, row-random and group-random distributions use an easy cyclic distribution with increasing coarse levels. The random plot stands for a L1 block level distribution, while the row-random stands for a distribution done at the row level and group-random distributes groups of 10 rows. Since there are always dependences from the diagonal block to the rest of the blocks on the same row, the row-random distribution converts them to local and the number of messages decreases. For the architecture simulated this is still not enough to make the parallel execution faster then the sequential. The group-random distribution decrements more the number of inter-processor communications but still the speed-up is null. The last heuristic, named balanced, uses the PET to distribute



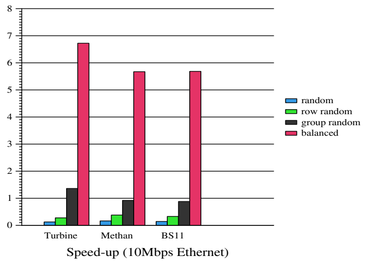
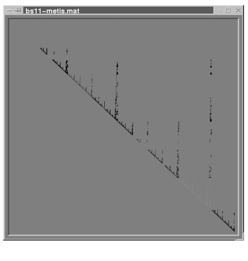


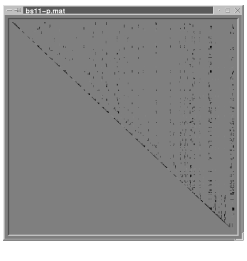
Fig 3.Data distribution simulations for a 8 multiprocessor

rows. Fig 3.b shows that this reduces the number of data communication again, now in a factor form 5 to 9. This reduction makes the difference in terms of speed-up, which is raised up to 6 for the simulated architecture. The balanced data distribution works as follows. Initially it assigns all the PET nodes to one processor. Then it enters in an iterative loop that decides to reassign a subtree from the most heavily loaded processor to the less loaded processor. The computational weight of the subtree is considered when deciding the PET cutting point. The loops iterates until a 5% threshold on processors balance is achieved.

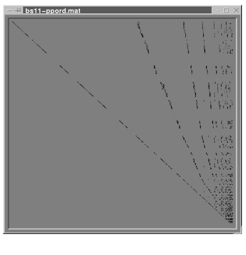
The hypermatrix of Fig 4.a shows with 8 colours the results of the balanced distribution. The colours are clearly defined because joint consecutive rows. This block ordering is now a problem for the dynamic scheduler. The probability of having a Task Graph with tasks distributed to different processors is very low. The solution interleaving, this is, to find an equivalent reordering of the PA such that blocks distributed to a same processor are not consecutive.



a) data distribution



b) interleaving by rows



c) interleaving by blocks

Fig 4.Example of interleaving on BS11 for 8 processors

The mixture of colours of Fig 4.b an Fig 4.c shows this graphically. Such new reordering can expose the parallelism to the PTM from the beginning because the operands of the tasks on the dynamic TG are distributed over all the processors. Fig 4.b is achieved with a post-ordering at the block-row level. This schema showed very good speedups on the simulations but was no introduced in PERMAS environment because it introduced much storage in the L2 level. Fig 4.c shows the final heuristic integrated in PERMAS which uses a coarser post-ordering heuristic (10 rows).

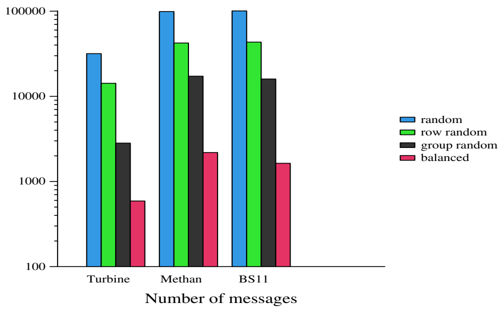


Fig 5.Elapsed time (SGI Origin 2000)

Finally, Fig 5. shows the performance of PERMAS parallelisation after applying the preparatory steps using 2, 4 and 8 slave processors. The total application execution time and the solver execution time are shown. Time savings of up to 20% and 40% of the total application time are achieved for 2 and 4 processors respectively. With 8 processors an additional gain of 5% saved time shows that more effort has still to be done in scalability. The main benefits are obtained on the parallelisation of the solver, but still the rest of the application has improvements from 10% to 15%. The solver speedup, that ranges from 2.4 to 5.3, is much better than speedups reported for Abaqus [1.] or MSC/ Nastran [7.], which are less than 2 for large problems. This is an impressive performance if we consider the important amount of I/O overhead of the PERMAS out-of-core applications, specially in the backward and forward substitutions.

5 Conclusions and Future Work

This paper shows the need of several preparatory steps on sparse matrix structure for obtaining good performance of the automatic parallelisation of PERMAS. We propose a variable sized blocking of the hypermatrix and show how this blocking alternative saves storage and speeds the parallel execution. Also, a data distribution step is proposed and considered together with the dynamic scheduler that shows promising speedups even on slow multiprocessor networks. Finally, the interleaving step, done with a post-ordering algorithm, shows to be essential to expose dynamically the available parallelism. All these steps are integrated into the core of the PERMAS system. The speedups measured for real executions are much better than other commercial out-of-core FE systems. The benefits are achieved mostly for the solver part of the application, but PERMAS parallelisation approach also benefits the rest of the application. We are now working on the extension of the parallelisation to other parallel paradigms (multi threading). We also plan to investigate additional parallelisation granularity (medium and coarse grain), and the parallelisation of all the application (matrix assembly operations, preparatory steps).

Acknowledgments. This work has been partially supported by the Ministry of Education of Spain under contract TIC98-0511, by the CEPBA and by the European Commission under ESPRIT contract n.22740 (PARMAT project).

References

1. Abaqus product performance. http://www.abaqus.com/products/p_performace58.htm
2. M. Ast, R. Fischer, J. Labarta and H. Manz. "Run-Time Parallelization of Large FEM Analyses with PERMAS". NASA'97 National Symposium. 1997.
3. T. Bui and C.Jones "A heuristic for reducing fill in sparse matrix factorization". 6th SIAM Conf. Parallel Processing for Scientific Computing, pp.445-452, 1993.
4. S. Fink , S. Baden and S. Kohn. "Efficient Run-Time Support for Irregular Block-Structured Applications". Journal of Parallel and Distributed Computing 50, pp.61-82. 1998.
5. T. Johnson. "A concurrent Dynamic Task Graph". International Conference on Parallel Processing, 1993.
6. G. Karypis and V. Kumar. "A fast and highly quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing. 1995.
7. L. Komzsik, "Parallel Processing in MSC/Nastran'. 1993 MSC World Users Conference, Virginia, 1993. http://www.macsch.com
8. V. Kumar et al. "Introduction to parallel Computing. Design and analysis of algorithms. The Benjamin/Cumminngs Pub. 1994.
9. J. Liu. "Computational models and task scheduling for parallel sparse Cholesky factorization". Parallel Computing 3, pp.327-342, 1986.
10. Marc product description. http://www.marc.com/Product/MARC
11. R. Schreiber. "Scalability of sparse direct solvers". Graph theory and sparse matrix computations, The IMA volumes in mathematics and its applications, vol. 56, pp.191-209, 1993.
12. S. Venugopal, V. Naik. "Effects of partitioning and scheduling sparse matrix factorization on communications and load balance". Supercomputing'91, pp.866-875, 1991.