A Study on Load Imbalance in Parallel Hypermatrix Multiplication using OpenMP [⋆]

José R. Herrero, Juan J. Navarro

Computer Architecture Dept., Univ. Politècnica de Catalunya
Barcelona, (Spain)
{josepr,juanjo}@ac.upc.edu,

Abstract. In this paper we present our work on the the parallelization of a matrix multiplication code based on the hypermatrix data structure. We have used OpenMP for the parallelization. We have added OpenMP directives to a few loops and experimented with several features available with OpenMP in the Intel Fortran Compiler: scheduling algorithms, chunk sizes and nested parallelism. We found that the load imbalance introduced by the hypermatrix structure could not be solved by any of those OpenMP features.

1 Introduction

We have used a Hypermatrix data structure [1, 2] in sequential linear algebra codes. We could obtain efficient implementations in both sparse and dense codes. Now, we are interested in the parallelization of our dense codes. This data structure, however, presents difficulties when work has to be distributed amongst several processors. Namely, the difficulty to balance the load evenly. In this paper we present the work we have done to produce a hypermatrix multiplication based on OpenMP directives. We wanted to know whether some of the features available in OpenMP could surmount the intrinsic difficulties of parallel codes based on the Hypermatrix data structure. We have chosen matrix multiplication because it is highly parallelizable. Also, it is a very important operation since it appears as a basic kernel in many scientific applications. For this reason it has been studied extensively [3–5].

1.1 OpenMP

OpenMP [6] provides a set of directives and environment variables to express and control parallelism in the execution of a program. The user can choose the scheduling algorithm. When a *static* scheduling algorithm is used, the distribution of iterations to threads is done before the execution of any of them. When a *dynamic* scheduling algorithm is used, the next piece of work for a thread is assigned when it is needed. It is taken from the remaining operations due. There

[⋆] This work was supported by the Ministerio de Ciencia y Tecnología of Spain (TIN2004-07739-C02-01)

is a default value for the number of iterations assigned to each processor which can be changed by the user explicitly. We refer to the *chunk* size. The default for the static scheduling is to split the work in as many parts as the number of threads defined. The default for the dynamic scheduling is to take one iteration each time.

Several nested loops can be parallelized. OpenMP permits this fact with a feature known as *nested parallelism*. When nested parallelism is activated, parallel constructs can be used within other parallel constructs.

In this paper we have used OpenMP for the parallelization of a matrix multiplication code based on the hypermatrix data structure.

1.2 Hypermatrix data structure

Our application uses a data structure based on a hypermatrix (HM) scheme [1, 2], in which a matrix is partitioned recursively into blocks of different sizes. A commercial package known as PERMAS uses the hypermatrix structure for solving very large systems of equations [7]. It can solve very large systems out-of-core and can work in parallel. This approach is also related to a variety of recursive/nonlinear data layouts which have been explored elsewhere for both regular [8–11] and irregular [12] applications.

The HM structure consists of *N* levels of submatrices. In order to have a simple HM data structure which is easy to traverse we have chosen to have blocks at each level which are multiples of the lower levels. The top *N-1* levels hold pointer matrices which point to the next lower level submatrices. Only the last (bottom) level holds data matrices. Data matrices are stored as dense matrices and operated on as such. Hypermatrices can be seen as a generalization of quadtrees. The latter partition each matrix precisely into four submatrices [13].

Null pointers in pointer matrices indicate that the corresponding submatrix does not have any non-zero elements and is therefore unnecessary. This is useful when matrices are sparse. Figure 1 shows a sparse matrix and a simple example of a corresponding hypermatrix with 2 levels of pointers.

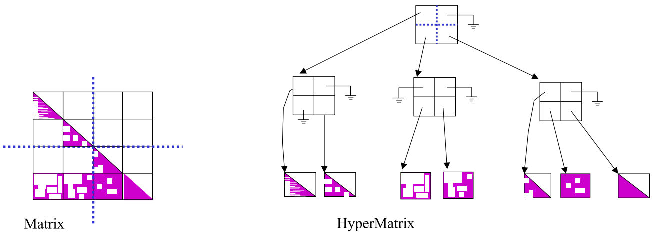


Fig. 1. A sparse matrix and a corresponding hypermatrix.

In the past, we have been working on the sparse Cholesky factorization based on the hypermatrix data structure. We created efficient routines which operate on small matrices. By small we mean matrices which fit in cache. We grouped such routines in a library called the Small Matrix Library (SML). Information about the creation of the SML can be found in [14]. Further details on the application of SML to sparse hypermatrix Cholesky can be found in [15].

The hypermatrix data structure, however, can also be used for dense matrix computations. Recently, we have applied a similar approach to work on dense matrices. When working on dense matrices in-core, two levels of pointers are enough. To work efficiently on dense matrices we have extended our SML with routines which work with larger sizes than the ones used for the sparse codes. On MIPS, ALPHA and Itanium2 platforms we could obtain very efficient codes for the matrix multiplication.

Now, we are interested in the efficient execution on multiprocessor machines. The hypermatrix data structure, however, presents some difficulties when parallel code is developed. Namely, the partitioning of the matrix is done when the data structure is set. Each pointer in the upper pointer matrix level maps a part of the matrix. If the dimension of such matrix is not a multiple of the number of processors used then the load is not distributed evenly amongst them.

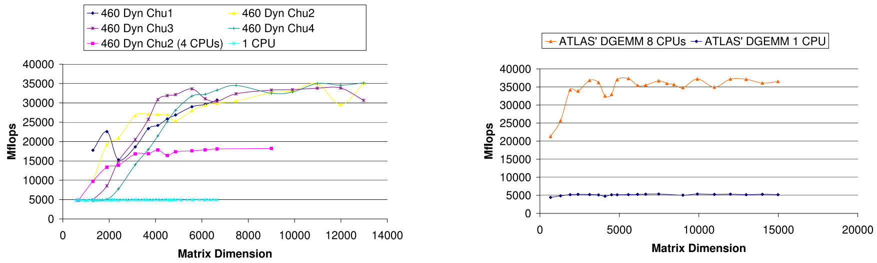
We have started with the study of the hypermatrix multiplication operation, which is very regular and has a high potential for parallelism. We have added OpenMP directives to a few loops and experimented with several features available with OpenMP in the Intel Fortran Compiler: scheduling algorithms, chunk sizes and nested parallelism. We anticipate than none of these features was completely successful for the efficient parallelization of our code.

2 Parallel dense hypermatrix multiplication

The target machine was a 8-way SMP with Intel Itanium2 processors running at 1.5 GHz. The theoretical peak of this machine is 48 Gflops. The Itanium2 has three levels of cache. In the first level it has separate instruction and data caches with 16 Kbytes each. Then, it also has a 256 Kbytes L2 cache and an off-chip L3 cache with possible sizes ranging from 1.5 up to 9 MB.

We have experimented with four and eight processors. In this section we will discuss the results obtained. Our preliminary study on four CPUs provides a speed-up of 3.7 for medium to large matrices. The best combination was that where the two outermost loops were parallelized using nested parallelism and a dynamic scheduling algorithm was used where the chunk size equaled 2. Figure 2a shows the performance of our hypermatrix multiplication code on four processors for the $C = C - A * B^T$ operation for both the sequential and parallel versions of our code. We have used an upper block size of size 460×460, i.e. each upper level pointer maps a block of such size. We then tried the same approach on eight processors. The same figure 2a shows the performance obtained on eight processors with a dynamic scheduling strategy, with the two outermost loops parallelized using nested parallelism. Several chunk sizes were used.

We observe that for small matrix dimensions small chunk sizes provide better results. However, as the matrix gets bigger, larger chunk values can be more effective. This is due to the reduction in the overhead which occurs when one thread searches for new work. Giving a thread more work at once reduces the



a) b)

Fig. 2. a) Two parallel loops with dynamic scheduling and several chunk sizes on 8, 4 and 1 processors. **b)** Performance of ATLAS' DGEMM.

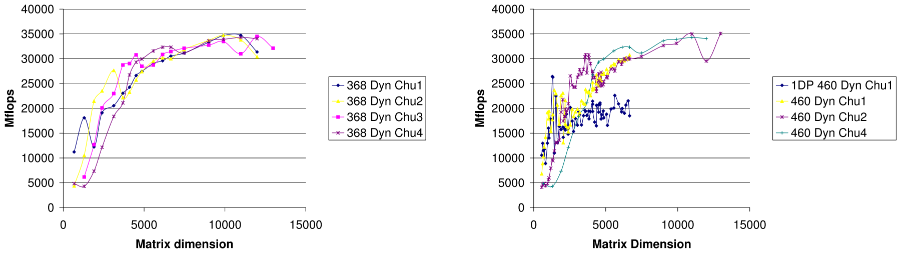
number of times this needs to be done. Also, the memory hierarchy can be better used since contiguous blocks corresponding to consecutive iterations can be reused in the cache. It is important to note that a certain chunk value is effective only when it keeps a good load balancing. Since the loops we have parallelized are the outer loops, they correspond to the upper level pointer matrix. The chunk size times the number of processors should divide the upper level matrix dimension evenly. Otherwise, load imbalance occurs and the performance drops.

We wanted to compare our results to those of ATLAS [3]. Figure 2b shows the performance of the sequential and parallel (on eight processors) versions of ATLAS matrix multiplication routine DGEMM. Their code, starting with the sequential version, outperforms ours. We must note, however, that on this machine ATLAS uses a hand-tuned kernel. The interesting point here comes from the fact that their code achieves high speed-ups sooner than our code. For some large matrix dimensions the speed-up we obtain is similar to theirs (around 7.0). However, for smaller matrices our speed-up is considerably lower. This is due to the load imbalance mentioned above. We have revisited our code and tried several variants aiming to improve its performance, specially when working on smaller matrices.

2.1 Reducing the block size

By default we have been using an upper block size of 460×460, i.e. each upper level pointer maps a block of such size. However, we have also reduced the size of the block to 368×368. Figure 3a shows the performance obtained with dynamic scheduling and nested parallelism for this block size. Results are similar to those obtained with our default block size of 460×460.

For the upper levels we have also tried other multiples of the lower levels close to the value $\sqrt{C/2}$, where C is the cache size [16]. Figure 5b shows the performance obtained with several sizes. To simplify the comparison, the maximum value obtained for all chunk sizes for a given block size are presented. Results are similar for all of them. We must note that the smaller block size 276×276 provides the worst performance for larger matrices. This size does not use the memory hierarchy so effectively. Also, there is more overhead in the parallelization.



a) b)

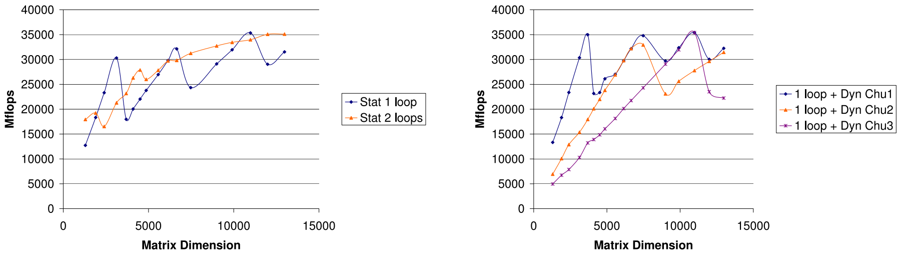
Fig. 3. a) Two parallel loops with dynamic scheduling: smaller blocks. **b)** Three loops parallelized (one in the level of pointers to data).

2.2 Parallel loop in level of pointers to data

We have tried another code which parallelizes a third loop in addition to the outermost two loops. This loop is the outermost loop in the level of pointers to data. Figure 3b compares its results to those shown in figure 2a. This code only gets better performance for a few small matrices. For larger matrices, this code does not offer any advantages. The granularity of this third loop is too small and the overhead of the parallelization outweighs any possible advantages.

2.3 Static scheduling

Figure 4a shows the results obtained with a static scheduling. Results with and without nested parallelism are shown. When only the outermost loop is parallelized we get a saw shape curve. The peaks correspond to sizes which get a perfect partitioning of the hypermatrix, i.e. with a number of pointers in the upper matrix which is a multiple of the number of processors. The use of nested parallelism introduces some overhead. However, it improves the performance for matrix sizes which are not multiples of the number of CPUs. The performance obtained in both cases is in general worse than that presented in figure 2.



a) b)

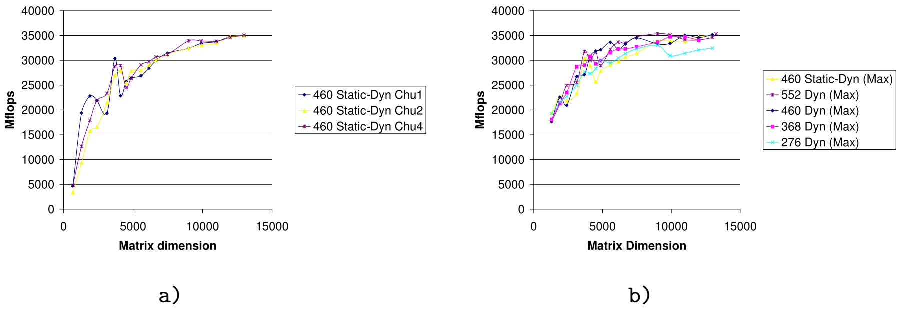
Fig. 4. a) One and two parallel loops with static scheduling. **b)** Parallel outer loop with dynamic scheduling and several chunk sizes.

2.4 Dynamic scheduling with only 1 Parallel loop

Figure 4b shows the results obtained when only the outermost loop is parallelized. A dynamic scheduling is used in this case. Again, we get a saw shaped curve. It is quite obvious that larger chunk sizes suffer from load imbalance more often.

2.5 Combining Static and Dynamic scheduling algorithms

We have scheduled the outermost loop using a static scheduling and the second outermost loop using a dynamic scheduling. Figure 5a shows the performance obtained. The performance obtained is similar to the one obtained when both loops are scheduled using a dynamic scheduling algorithm as shown in figure 5b.



a)

b)

Fig. 5. a) Static scheduling of outermost loop and dynamic scheduling of next inner loop. **b)** Maximum values obtained for each block size and scheduling algorithm.

2.6 Perfect matrix partitioning

Figure 6 shows the results obtained when the matrix has been partitioned in a number of parts which is multiple of the number of threads. All partitions have the same size: each upper level pointer maps blocks of 460 × 460. Thus, a dimension of 3680 produces a hypermatrix with eight pointers in the upper level. The number of pointers in the upper level corresponding to the other three matrix dimensions in the figure are 16, 24 and 32 respectively. All of them are examples where the load can be easily balanced amongst the processors.

These results allow us to study the overhead of each strategy. We use *nest* to identify the use of nested parallelism. A value of 0 means nested parallelism is not allowed while a value of 1 means the opposite. Label *sch* corresponds to the scheduling algorithm used. A value of 0 denotes static scheduling. A value of 1 is used to indicate dynamic scheduling. We use *chu* to specify the chunk size. A value of 0 is used to signify the default value for a given scheduling strategy.

On these perfectly partitioned hypermatrices we can observe that, when the matrices are small, the only way to get good speed-ups is via simple strategies:

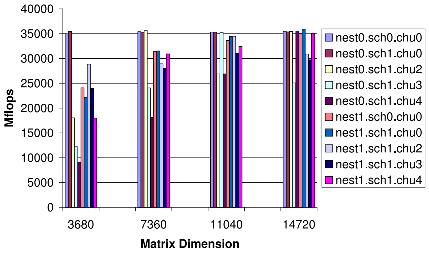


Fig. 6. Experiments with different OpenMP features when the hypermatrix is partitioned for perfect load balancing.

parallelizing only the outermost loop with either static or dynamic scheduling. As the matrices get large there are more strategies which provide good speedups. However, in any case it is important to use a chunk size which allows for a good load balancing. The result of dividing the dimension of the upper level pointer matrix by the number of threads must be a multiple of the chunk size.

In a few occasions, a chunk size larger than the default for the dynamic scheduling strategy (which defaults to 1) can improve slightly the performance of our matrix multiplication. This is due to the reduction of the overhead which occurs when one thread takes several iterations at once instead of taking one iteration each time. Also, better use of the memory hierarchy results when one thread reckons several contiguous blocks corresponding to consecutive iterations.

Nested parallelism is not really effective in such situations. It cannot provide any advantages. Instead, it introduces an additional overhead with the creation of the inner parallel construct.

3 Conclusions

We conclude that the best way to parallelize our application is by means of an adequate partitioning of the matrix. If this is possible, a simple scheduling strategy where just the outermost loop is parallelized turns out to be the best solution. Both static and dynamic scheduling algorithms work well and perform in a similar manner.

When data cannot be partitioned adequately we can take advantage of nested parallelism. Despite its overhead, it offers the advantage of being able to open new parallel sections which can employ otherwise idle processors. The resulting performance curves are smoother than the saw shaped curves which result from those cases where only the outer loop was parallelized.

We have conducted experiments with eight processors and found some load imbalance in those cases where the dimension of the matrix in the upper pointer level is low and is not multiple of the number of processors used. Thus, smaller matrices suffer from load imbalance as the number of processors grow. This can limit the effectivity of parallel codes based on the hypermatrix scheme. In the future, we plan to replace the hypermatrix data structure in our algorithms which deal with dense matrices. We plan to use a plain storage of the data submatrices which can be accessed with a simple indexing scheme. We believe that in this way we can still use our routines which deal with small submatrices and, at the same time, can split the work amongst all processors more effectively.

References

1. Fuchs, G., Roy, J., Schrem, E.: Hypermatrix solution of large sets of symmetric positive-definite linear equations. Comp. Meth. Appl. Mech. Eng. **1** (1972) 197–216
2. Noor, A., Voigt, S.: Hypermatrix scheme for the STAR–100 computer. Comp. & Struct. **5** (1975) 287–296
3. Whaley, R.C., Dongarra, J.J.: Automatically tuned linear algebra software. In: Supercomputing '98, IEEE Computer Society (1998) 211–217
4. Choi, J., Dongarra, J., Pozo, R., Walker, D.: ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers. In: Proc. Fourth Symposium on the Frontiers of Massively Parallel Computation, ACM Press (1992) 120–127
5. Chatterjee, S., Lebeck, A.R., Patnala, P.K., Thottethodi, M.: Recursive array layouts and fast parallel matrix multiplication. In: Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures, ACM Press (1999) 222–231
6. OpenMP: (URL) http://www.openmp.org.
7. Ast, M., Fischer, R., Manz, H., Schulz, U.: PERMAS: User's reference manual, INTES publication no. 450, rev.d (1997)
8. Chatterjee, S., Jain, V.V., Lebeck, A.R., Mundhra, S., Thottethodi, M.: Nonlinear array layouts for hierarchical memory systems. In: Proceedings of the 13th international conference on Supercomputing, ACM Press (1999) 444–453
9. Frens, J.D., Wise, D.S.: Auto-blocking matrix multiplication, or tracking BLAS3 performance from source code. Proc. 6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Program., SIGPLAN Not. **32** (1997) 206–216
10. Valsalam, V., Skjellum, A.: A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. Concurrency and Computation: Practice and Experience **14** (2002) 805–839
11. Wise, D.S.: Ahnentafel indexing into Morton-ordered arrays, or matrix locality for free. In: Euro-Par 2000,LNCS1900. (2000) 774–783
12. Mellor-Crummey, J., Whalley, D., Kennedy, K.: Improving memory hierarchy performance for irregular applications. In: Proceedings of the 13th international conference on Supercomputing, ACM Press (1999) 425–433
13. Wise, D.S.: Representing matrices as quadtrees for parallel processors. Information Processing Letters **20** (1985) 195–199
14. Herrero, J.R., Navarro, J.J.: Automatic benchmarking and optimization of codes: an experience with numerical kernels. In: Proceedings of the 2003 International Conference on Software Engineering Research and Practice, CSREA Press (2003) 701–706
15. Herrero, J.R., Navarro, J.J.: Improving Performance of Hypermatrix Cholesky Factorization. In: Euro-Par 2003,LNCS2790, Springer-Verlag (2003) 461–469
16. Lam, M., Rothberg, E., Wolf, M.: The cache performance and optimizations of blocked algorithms. In: Proceedings of ASPLOS'91. (1991) 67–74