Some Experiments and Issues to Exploit Multicore Parallelism in a Distributed-Memory Parallel Sparse Direct Solver

Indranil Chowdhury, Jean-Yves L'Excellent

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Some Experiments and Issues to Exploit Multicore Parallelism in a Distributed-Memory Parallel Sparse Direct Solver

Indranil Chowdhury — Jean-Yves L'Excellent

N° 7411

October 2010

Domaine 3

ISSN 0249-6399 ISRN INRIA/RR--7411--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

Some Experiments and Issues to Exploit Multicore Parallelism in a Distributed-Memory Parallel Sparse Direct Solver*

Indranil Chowdhury† , Jean-Yves L'Excellent‡

Domaine : Réseaux, systèmes et services, calcul distribué
Équipe-Projet GRAAL

Rapport de recherche n° 7411 — October 2010 — 32 pages

Abstract: MUMPS is a parallel sparse direct solver, using message passing (MPI) for parallelism. In this report we experiment how thread parallelism can help taking advantage of recent multicore architectures. The work done consists in testing multithreaded BLAS libraries and inserting OpenMP directives in the routines revealed to be costly by profiling, with the objective to avoid any deep restructuring or rewriting of the code. We report on various aspects of this work, present some of the benefits and difficulties, and show that 4 threads per MPI process is generally a good compromise. We then discuss various issues that appear to be critical in a mixed MPI-OpenMP environment.

Key-words: sparse matrices, direct solver, MPI, OpenMP, multicore, multithread

* This work was partially supported by the ANR SOLSTICE project, ANR-06-CIS6-010, and by a French-Israeli "Multicomputing" project. It relies on the MUMPS software package (see http://graal.ens-lyon.fr/MUMPS or http://mumps.enseeiht.fr)

† The work of this author was done while employed by INRIA and working at Université de Lyon, Laboratoire LIP (UMR 5668, CNRS, ENS Lyon, INRIA, UCBL), ENS Lyon, 46 allée d'Italie, 69364 Lyon Cedex 7, France.

‡ INRIA, Université de Lyon, Laboratoire LIP (UMR 5668, CNRS, ENS Lyon, INRIA, UCBL), ENS Lyon, 46 allée d'Italie, 69364 Lyon Cedex 7, France.

This text is also available as a research report of the Laboratoire de l'Informatique du Parallélisme http://www.ens-lyon.fr/LIP.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Premières expériences pour l'exploitation d'un parallélisme de type multi-cœurs dans un solveur creux direct pour machines à mémoire distribuée

Résumé : MUMPS est un logiciel de résolution de systèmes linéaires creux de grande taille, basé sur une méthode directe de factorisation. Il utilise un parallélisme de type passage de messages basé sur MPI (Message Passing Interface). Dans ce rapport, nous montrons comment un parallélisme de type multithread peut permettre de mieux tirer parti des architectures multi-cœurs récentes. Le travail effectué consiste à utiliser des librairies BLAS multithreadées et à insérer des directives OpenMP dans les routines s'avérant être les plus coûteuses, avec comme objectif de fond d'éviter toute restructuration profonde ou réécriture significative du code. Nous présentons divers aspects de ce travail, les gains et les difficultés, et montrons qu'utiliser 4 threads par processus MPI est souvent un bon compromis. Nous insistons pour finir sur les aspects qui nous semblent critiques pour une bonne exploitation d'un parallélisme mixte MPI-OpenMP.

Mots-clés : matrices creuses, solveur direct, MPI, OpenMP, multicœur, multithread

1 Introduction

MUMPS (**MU**ltifrontal Massively **P**arallel sparse direct **S**olver) [3, 5] is a parallelized sparse matrix solver which is widely used by many research and commercial groups around the world. This software was originally developed using MPI (Message Passing Interface) for distributed memory machines, although the initial developments actually relied on a shared-memory code [2]. It uses the BLAS and ScaLAPACK [9] libraries for dense matrix computations and is written in Fortran 90 and C. In recent years, with the advent of multicore machines, interest has shifted towards multithreading existing software in order to maximize utilization of all the available cores. Some researchers have reported development of multithreaded direct solvers - for example Anshul Gupta in the IBM WSMP software [15] and Jonathan Hogg in the HSL software [16]. This work is focused towards the multithreading of existing MUMPS software without a deep restructuring of the existing code, in other words, by adding an OpenMP layer on top of the already existing MPI layer. While a pure MPI implementation can still be used to run parallel processes on the available cores, we will see that a purely threaded or a mixed MPI-OpenMP strategy yields to significant gains in both memory usage and speed.

This work is part of the SOLSTICE project which comprises a team of academic institutions and industries interested in or heavily using sparse solvers. It is also part of a French-Israeli Multicomputing project implying ENSEEIHT-IRIT, ENS Lyon and Tel Aviv University, whose aim is to improve the scalability of state-of-the-art computational fluid dynamics calculations by the use of state-of-the-art numerical linear algebra approaches.

MUMPS is used to solve large scale systems of the form

$$\mathbf{A}x = b, \tag{1}$$

where $\mathbf{A}$ is a $n \times n$ symmetric positive-definite, symmetric indefinite or unsymmetric sparse matrix. b can be single or multiple right hand sides. The above forms the core of computations in many engineering applications ranging from finite-elements to modified nodal analysis. There are three typical methods for solving these systems : (1) iterative methods such as multigrid solvers, (2) direct methods (such as MUMPS) and (3) hybrid methods that combine both direct and iterative techniques. Direct methods consist of three phases - analysis, factorization and solve. MUMPS relies on a multifrontal approach [11, 12]; the analysis phase uses ordering packages like AMD, METIS, PORD and SCOTCH to build the assembly tree for the subsequent factorization and solve phases. In the factorization phase, we seek an $\mathbf{LU}$ factorization of the matrix $\mathbf{A}$ when it is unsymmetric, and an $\mathbf{LDL}^T$ factorization when it is symmetric. The solution phase consists of sparse triangular solves to obtain the solution.

In this report, we are primarily concerned with the multithreading of the factorization and the solve phases since most sparse solver applications require single analysis phase and multiple factorization and solve phases. OpenMP directives are used for multithreading as opposed to POSIX threads, because of the ease of usage of OpenMP in structured codes. This report is organized as follows. After describing our metholdology and experimental setup in Section 2, we show in Section 3 which parts of the code are worth multithreading. Section 4 provides more experimental results of multithreading for different cases: symmetric positive definite, symmetric indefinite, and unsymmetric matrices. The solve phase is also considered. Then, various issues are discussed in Section 5: thread affinity, memory consumption, mix of MPI and OpenMP, minimum granularity to avoid speed-downs in some regions, and control of the number of threads in multithreaded BLAS libraries. We finally provide some concluding remarks and ideas for future work in Section 6, and refer the reader to the appendix for advises on experimenting this OpenMP version of MUMPS and description of the MUMPS routines cited in this report. Some experiments of using MUMPS with saddle-point problems are also given in the appendix.

2 Methodology and experimental setup

	Borderline Opteron	Dunja Nehalem
Location	LaBRI, Bordeaux	LIP, ENS Lyon
Processor	AMD Opteron 2218	Intel Xeon Nehalem E5520
Frequency	2.6 GHz	2.27 GHz
One node	4 dual-core processors	2 quadri-core processors
Memory per node	32 GB PC2-5300 DDR2-667	32 GB
Cache	2 MB (L2)	8 MB (L3)
Compiler	Intel v.10	Intel v.11

Table 1: Test machines used in this study.

The experiments were conducted on 8 cores of two different platforms - Opteron (machine named Borderline) and Nehalem (machine named Dunja). The characteristics of those two machines are given in Table 1. Intel compilers were used for all the experiments reported here: version 10 was available on Borderline/Opteron, whereas version 11 was available on Dunja/Nehalem. The Intel compilers were used as opposed to GNU because with the versions used, we had better results and less surprises with them than with GNU when using OpenMP. With those compilers, thread affinity (which describes how likely a thread is to run on a particular core), can be set thanks to an environment variable. Thread affinity will be discussed later in this report

Symmetric matrices			
Name	Order	nonzeroes	Origin
APACHE2	715176	4 817 870	Univ. Florida collection
THREAD	29736	4 444 880	PARASOL collection
BOXCAV	544932	3 661 960	SOLSTICE project
FDTD	343000	3 704 680	11-point finite difference (70x70x70 grid)
HALTERE	1288825	10476775	SOLSTICE project
Unsymmetric matrices			
Name	Order	nonzeroes	Origin
DIMES2 (complex)	55640	13 929 296	Sparse direct multipole solver [14]
AMAT30	134335	975205	Multicomputing project
AMAT60	965215	7458355	Multicomputing project
AMAT90	3140695	24795505	Multicomputing project
A1M	738235	4756918	part of AMAT60 excluding the rows and columns corresponding to the bottom-right zero block

Table 2: Test matrices used in this study. Matrices are ordered with METIS. HALTERE is complex but treated as if it was real (imaginary part ignored).

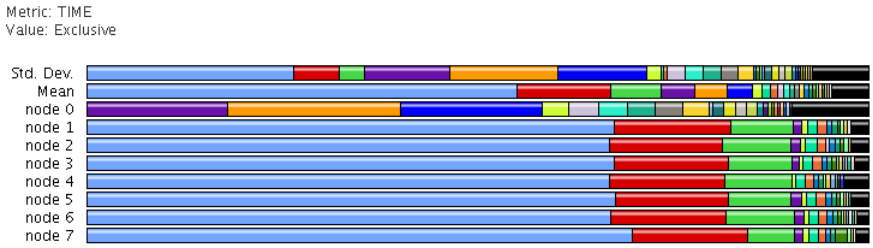

Figure 1: Multinodal Profiles generated by the TAU Paraprof tool.

(Section 5.1). Various symmetric and unsymmetric matrices were tested mainly from the SOLSTICE group of the GridTLSE website [19], Israeli project and Tim Davis collection [10]. The matrices used in this report are listed in Table 2. The bottlenecks to the code were identified by performing single node experiments on Borderline/Opteron and Dunja/Nehalem, with an open-source profiler called TAU[1] [18]. This tool was used to profile the performance of the code in a mixed MPI/OpenMP environment. Some features of TAU are mentioned here. Firstly, TAU has various functionalities for a number of performance or monitoring aspects that may be of interest in such code tuning and optimization. Some of them are – a) accurate profiling of all the parts of MUMPS to understand the intensive portions of the code, b) performance measures and optimizing the number of threads for each profiled OpenMP region (which can actually be a subportion of some routine), c) profiling hybrid OpenMP / MPI performance and d) call paths and dependencies. Memory profiling (leak detection), event tracing (compatible with the commercial tool Vampir[2]) and studying cache-hit rates can also be done, but at the expense of significantly increasing the runtime and diskspace. Textual and graphical profiles of all the subroutines can be generated after MUMPS execution once it is compiled using the TAU wrappers. In parallel, both thread-by-thread and averaged statistics are available. An example of a textual report is given in Table 3. Graphical visualization is also possible, an example of a multinodal profile generated by the TAU Paraprof tool is shown in Figure 1. In this figure, the surface of each colored bar represents the cost of a given subroutine.

3 Multithreading costly portions of MUMPS

The following assumes some familiarity with the multifrontal method and with MUMPS. Therefore, some basic explanations are available in Section C

[1] Available from `www.cs.uoregon.edu/research/tau`.
[2] now called Intel trace analyzer.

Time percentage	Exclusive msec	Inclusive msec	Num Calls	Num Subrs	Inclusive usec/Call	RoutineName
100.0	26,258	55,164	1	8	55164302	MAIN
52.4	0.217	28,905	3	24	9635259	`mumps`
49.8	7	27,457	1	55	27457075	`factodriver`
38.1	0.07	21,007	1	8	21007612	`facb`
38.1	105	21,000	1	335974	21000004	`facpar`
33.3	36	18,366	29430	112617	624	`factoniv1`
24.3	13,426	13,426	309	0	43452	`facp`
11.2	6,158	6,184	1	299	6184717	`factosendarrowheads`
8.2	4,510	4,510	1028	0	4388	`facq`
2.9	1,482	1,575	29430	118161	54	`facass`
2.3	1	1,244	1	58	1244343	`analdriver`
1.7	537	946	1	15	946437	`analf`
1.6	49	862	29430	159788	29	`facstack`
1.2	658	658	309	0	2130	`copyCB`
0.6	345	345	1	0	345723	`analgnew`
0.6	314	314	26520	0	12	`facm`

Table 3: Example of TAU profile when running MUMPS on an unsymmetric matrix.

of the appendix, together with a short description of the routines cited in this report. For more detailed explanations about the multifrontal or MUMPS, the reader should refer to the literature, for example papers [4, 3].

Profiling revealed that the following portions of MUMPS were generally the most costly ones and could be scaled by multithreading:

1. The BLAS operations during factorization. MUMPS uses Level 1, 2 and 3 BLAS operations during both factorization and solve. These operations consume most of the time, and can be multithreaded using threaded BLAS libraries such as GOTO, ACML and MKL (see Section 3.1. For example, in Table 3, `facp`, `facq`, and `facm` refer to the routines containing the most expensive BLAS operations from the factorization in the case of unsymmetric matrices.

2. Assembly operations in MUMPS (`facass`), where contribution blocks of children nodes of the assembly tree are assembled at the parent level. In this phase the initializations to zero of the frontal matrices were also costly and could be multithreaded in a straightforward way yielding considerable gain in the unsymmetric case (`facass`). In the symmetric case, because of the difficulty of parallelizing efficiently small triangular loops, the parallelization is slightly less efficient. The assembly of children contributions to parent nodes were also parallelized using OpenMP.

3. Copying contribution blocks during stacking operations (`facstack` and `copyCB` operations). These were multithreaded using OpenMP directives.

4. In some symmetric cases pivot search operations (`facildlt`) were found to be quite costly. These operations were also multithreaded using OpenMP directives.

In the next subsections the scaling of the multithreaded parts is discussed.

3.1 BLAS operations

The natural and simplest approach here consists in using multithreaded BLAS libraries. The well-known libraries include GOTOBLAS[3], probably the fastest, MKLBLAS[4] from Intel, ACMLBLAS[5] from AMD and ATLAS[6] BLAS. The number of threads is fixed during configuration in ATLAS and could not be varied at runtime using the environment variable OMP_NUM_THREADS. The other three libraries allow modifying the number of threads at runtime and were experimented with. Although the manuals suggest that both Level 3 and Level 2 BLAS operations are multithreaded, Level 3 BLAS functions scaled better because of the higher flops to memory accesses ratio. The routines `facsldlt` and `factldlt` contains Level 3 BLAS in symmetric cases, while `facp` and `facq` contain Level3 BLAS in unsymmetric cases (again, please refer to Section C of the appendix for more details on those routines). The Level 2 BLAS (`facmldlt` routine used in the symmetric case to factorize a block of columns and `facm` routine used in the unsymmetric case to factorize a block of rows) scaled poorly with the number of threads. The likely reason is bad data locality during these operations and a larger amount of memory accesses compared to the number of flops. In the unsymmetric and symmetric positive definite cases, `facm` and `facmldlt` consume a small percentage of the total time (as will be shown later) and hence we obtained comparatively better scaling than in the symmetric indefinite case, where the cost of `facmldlt` is larger. One reason is that in the unsymmetric case, the row storage is more cache friendly to factorize a block of rows, whereas the row storage is less efficient to factorize a block of columns. Furthermore (see C, the difference between the symmetric and symmetric positive cases come from the fact that in the symmetric positive definite case, much less work is performed in `facmldlt` because the non-fully summed rows will be updated later using Level 3 BLAS, in `factldlt`.

[3] www.cs.utexas.edu/users/flame/goto
[4] software.intel.com/en-us/intel-mkl/
[5] www.amd.com/acml
[6] math-atlas.sourceforge.net/

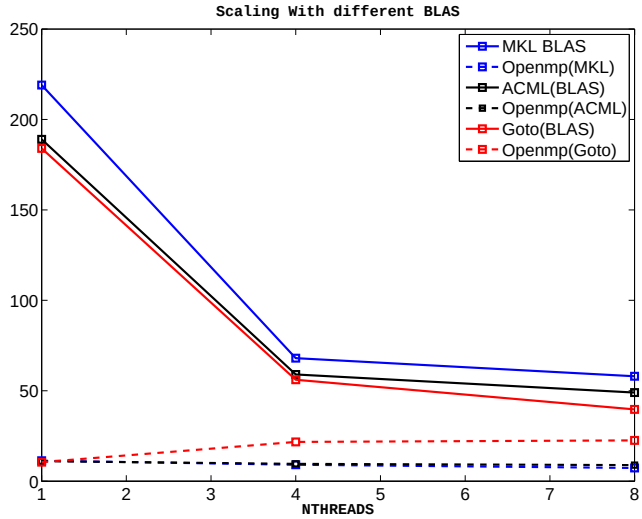

Figure 2: Time spent (seconds) in BLAS calls and in OpenMP regions, as a function of the BLAS library used, Opteron.

The GOTO library was configured using flag USE_OPENMP=1, in order to allow for compatibility with OpenMP. Although we observed that GOTO is the fastest among the BLAS libraries tested, it could not be used since even with USE_OPENMP=1 it still appeared to conflict with the other OpenMP regions, so that the net performance of MUMPS with high numbers of threads turned out to be better with other libraries. It seems that GOTO creates threads and keeps some threads active after the main thread returns to the calling application – perhaps this is the cause of interference with the threads created in the other OpenMP regions leading to the slowdown. Figure 2 shows this effect: with GOTO BLAS, the time spent in BLAS routines is smaller, but the performance of the other OpenMP regions in MUMPS deteriorates a lot. ACML perturbed the OpenMP regions only a little while MKL was found to be the most compatible with all the OpenMP regions. Therefore we used MKL for all our experiments.

3.2 Assembly operations

The *assembly operations* in MUMPS are performed in the routine `facass`, where the costly parts involve initialization of the parent block to zero followed by adding contributions from the children. In both operations, parallelization using OpenMP directives helped. These operations mostly involve memory accesses (together with add operations). The scaling for the testcase *AMAT30* ($N = 134335$, $NZ = 975205$, unsymmetric) is shown in Figure 3. We observe that the initialization to 0 shows slightly inferior speed-up with

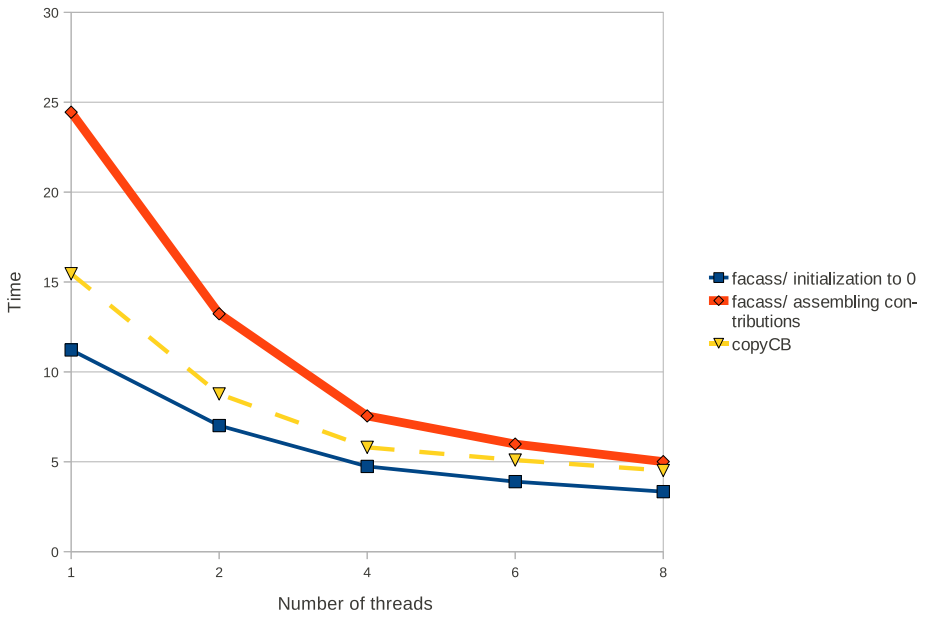

Figure 3: Assembly time as a function of the number of threads (testcase *AMAT30*), Opteron. The figure shows separately initialization to 0 and assembly of contributions blocks (both are part of the assembly routine facass). Time for copying contribution blocks (copyCB) is also given for comparison.

respect to the assembly of contribution blocks. However, this testcase shows good scaling with number of threads. On some other matrices the scaling tends to saturate after 4 threads. This is generally due to small front sizes and small contribution block sizes (less than 300), where we generally do not obtain any scaling. More details of scaling for each routine will be shown later.

3.3 Copying contribution blocks

These copies (routine `copyCB`) operations consist of memory accesses and are performed during stack operations (routine `facstack`); they are amenable to multithreading (as can already be observed in Figure 3). Figure 4 shows that the scaling strongly depends on the contribution block sizes. In this figure, each point gives the average time per call for a range of front sizes. For example, the value on the y-axis corresponding to 1500 on the x-axis represents the average time spent in the routine when the call is done with a contribution block in the range 1400 and 1600. Our experience is that the potential for scaling of `copyCB` and `facass` are similar: similar to assembly

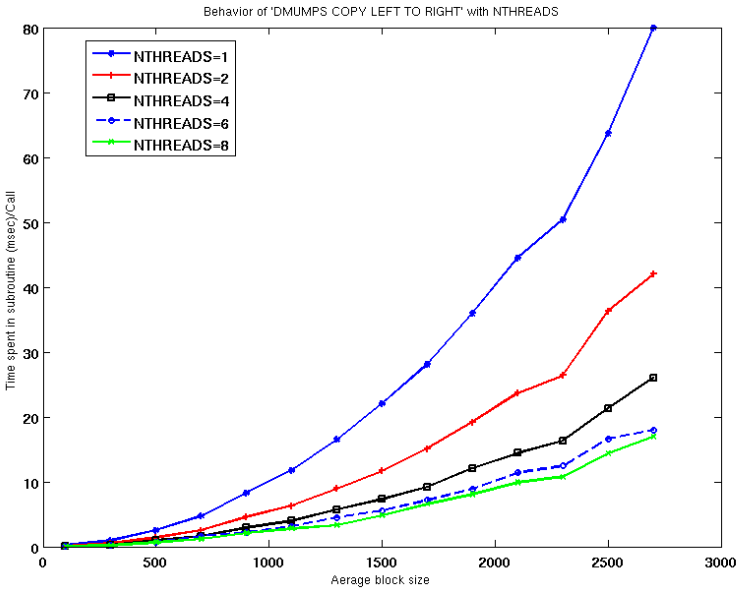

Figure 4: Behaviour of copyCB as a function of the number of threads. Each point on the x-axis refers to an interval of width 200. Testcase *DIMES*, Opteron.

operations, large blocks are needed to obtain reasonable speed-ups in these memory-bound operations. Unfortunately, the number of calls to `copyCB` or `facass` with small blocks is much larger than the number of calls with large blocks, so that overall, on the range of matrices tested, the bad scalability with small blocks can be a bottleneck when increasing the number of threads.

3.4 Pivot Search operations

In some symmetric indefinite testcases, the pivot search operations in the `facildlt` routine were found to be costly. In the unsymmetric cases the pivot search (routine `faci`) is on rows whose elements are contiguous in memory, while in symmetric cases pivot search is mainly done within columns with a stride equal to the front size. The non-contiguous memory accesses in symmetric cases are the likely reason for this increased cost. Still, the pivot search operations from `facildlt` were multithreaded using OpenMP

	1 thread	2 threads	4 threads
Loop RMAX	3.35	2.33	2.07
Loop TMAX	1.65	1.11	0.84

Table 4: Performance (time in seconds) of the two main loops of the symmetric pivot search operations (routine facildlt), testcase *THREAD*, Opteron.

	1 thread	2 threads
Loop RMAX(without IF statements)	0.789	7.965
Loop RMAX(with IF statements)	0.835	0.789
Loop TMAX(without IF statements)	0.033	0.035
Loop TMAX(with IF statements)	0.037	.032

Table 5: Stabilizing the facildlt OpenMP operations using IF statements, testcase *BOXCAV*, Opteron. Times in seconds.

reduction statements the results of which are shown in Table 4. In the testcase *THREAD* there was a reasonable gain in the two main pivot search loops, we call them RMAX and TMAX, however for most testcases there is a speed-down in these regions with the OpenMP directives. This is mostly caused within the regions of smaller loop sizes (< 300) or granularities which speed down distinctively as the number of threads is increased. We can stabilize this effect by using IF statements as a part of the OpenMP directive, which stops multithreading according to the IF statement. An example is shown in Table 5 where there is a disastrous slowdown without regulating the OpenMP region according to block sizes, and this effect is stabilized with addition of the IF statements. This stabilizing effect by adding the IF statement was observed both on Opteron and Nehalem machines.

4 Results of Multithreading and Case Studies

There are three different matrix modes that MUMPS takes as input, these are - symmetric positive definite, symmetric indefinite and unsymmetric. In the first two cases an LDL^T factorization is sought, whereas the unsymmetric cases are treated using an LU factorization. In this section, the results of multithreading for these different modes are discussed, and we then focus on the solve phase in Section 4.4. All experiments in this section were done with 'KMP_AFFINITY=none' (effects of thread affinity are discussed later). We remind that the METIS ordering is used, together with MKL BLAS and Intel compilers.

Routine	1 thread	2 threads	4 threads	8 threads
`factldlt`	21.9	11.62	7.61	5.28
`facsldlt`	2.4	1.28	0.81	0.54
`facmldlt`	0.12	0.12	0.12	0.13
`ldltassniv12`	0.73	0.70	0.67	0.64
`copyCB`	0.61	0.56	0.63	0.73
FactorTime	31.8	20.14	16.07	13.66

Table 6: Time (seconds) spent in the most costly routines as a function of the number of threads. Testcase *APACHE2*, Nehalem. The TAU profiler was used, inducing a small performance penalty.

4.1 Symmetric Positive Definite cases

Testcase APACHE2 (N=715176, NZ=2766523) from the University of Florida collection was used. The TAU profiles of the various routines with varying number of threads on Nehalem is shown in Table 6, this is a single node machine with 8 cores. The costliest, `factldlt` and `facsldlt` are the BLAS 3 routines which scale reasonably well with the number of cores. The speed-up is around 4 for these routines. However, there is little gain in the other multithreaded parts such as `facm` (Level 2 BLAS), `facass` and `copyCB`; the reason being the small granularity of tasks in these functions with many memory accesses. Overall in the purely threaded case there is an overall gain of around 3 times. In this testcase the multithreaded regions work on smaller contribution blocks and hence we don't see much gain. Later we will show that most of the gains come from the bigger blocks (block size > 200) and will try to stabilize the slowdowns in smaller blocks (see section 5.4).

Since the gain is smaller with increasing number of threads we should obtain better gains with a mixed MPI/OpenMP strategy than a purely threaded solver. This behaviour is shown in Table 7. For symmetric positive definite matrices we see that the pure MPI version (8 MPI) of MUMPS is faster (here almost twice faster) than the pure threaded version (8 threads). However, using 4 threads per MPI process pushes the gains a little further with an overall speed-up of 4.28 (the problem is relatively small). For a single right-hand side, the solve time decreases with increasing number of MPI processes. This confirms the difficulty of exploiting multiple threads in BLAS 2 operations in the threaded version, compared to the natural exploitation of tree parallelism with MPI.

We now provide some elements of comparison with the HSL MA87 code, a version of which was provided to us by the authors. HSL MA87 is a DAG-based solver for sparse symmetric positive definite matrices [16] written by Jonathan Hogg especially designed to efficiently exploit multicore architectures. The scaling of this solver versus MUMPS is shown in Table 8, where it can be seen that HSL MA87 scales almost perfectly with the

	8 MPI 1 thread	4 MPI 2 threads	2 MPI 4 threads	1 MPI 8 threads
FactorTime	7.68	7.40	**6.68**	13.48
SolveTime (1 RHS)	0.13	0.27	0.37	0.58

Table 7: Time (seconds) for the factor and solve phases with mixed MPI and OpenMP on 8 cores, testcase *APACHE2*, Nehalem, compiling without TAU. Serial version (1 MPI, 1 thread) takes 28.62 seconds.

number of threads, with a best case factorization time of 4.62 seconds. As we have shown before in Table 7, the best case performance of MUMPS is 6.68 seconds for this case and although this is inferior to HSL performance we are quite close. It is important to note here that MUMPS can handle general matrices on distributed machines and it may be difficult to obtain similar scaling as HSL MA87 in a general solver with the current OpenMP approach.

	1 thread	2 threads	4 threads	8 threads
FactorTime (HSL MA87)	28.04	14.49	7.57	4.62
SolveTime (HSl MA87)	0.609	0.69	0.699	0.724

Table 8: Time (seconds) for the factor and solve phases of the HSL MA87 code. Testcase *APACHE2*, Nehalem; refer to Table 7 for the performance of MUMPS on 8 cores.

The scaling and performance gains for a bigger testcase, the *FDTD* matrix($N = 343000$, $NZ = 3704680$) are shown in Table 9 and Table 10. In this case, using 8 threads for one MPI process gives the best speed-up of almost 5.5. In comparison, HSL MA87 gives a speed-up of 6.9, corresponding to a performance of 164.16 seconds with 1 thread and 23.93 seconds with 8 threads. The solve time for 1 RHS for the HSL code on 8 threads was 1.02 second, while for MUMPS it was 0.57 second so that it can be interpreted that MUMPS solve performance is better than MA87 for the bigger cases. Note that there is a difference in FactorTime between Table 9 and Table 10. For this testcase, this is due to the overhead when TAU is used for compiling MUMPS.

4.2 Symmetric indefinite case

MUMPS can handle symmetric indefinite matrices, where a factorization of LDL^T is sought with both 1x1 and 2x2 pivots in D. Many practical applications yield matrices of this form, for example most of the SOLSTICE matrices belong to this category. The scaling of a Solstice matrix, *Haltere* ($N = 1288825$, $NZ = 10476775$) is shown in Table 11, where the scaling tends to saturate after 4 threads. Therefore we expect that using 1 MPI

Routine	1 thread	2 threads	4 threads	8 threads
`factldlt`	134.84	69.09	36.91	21.36
`facsldlt`	19.54	9.53	5.37	3.33
`facmldlt`	0.31	0.38	0.37	0.73
`ldltassniv12`	1.68	1.49	1.29	1.16
`copyCB`	1.44	1.19	0.97	0.895
FactorTime	167.56	89.63	51.2	33.64

Table 9: Time (seconds) spent in costly routines as a function of the number of threads. Testcase *FDTD*, Nehalem. The TAU profiler was used inducing a small performance penalty.

	1 MPI 8 threads	2 MPI 4 threads	4 MPI 2 threads	8 MPI 1 threads
FactorTime	**29.76**	42	39.33	47.87
SolveTime	0.55	0.4085	0.50	0.3

Table 10: Time (seconds) spent in the factor and solve phases with mixed MPI and OpenMP on 8 cores. Testcase *FDTD*, Nehalem, compiled without TAU. Serial version (1 MPI, 1 thread) takes 164 seconds.

process per 4 threads should perform better than a pure OpenMP version on 8 threads. This is shown in Table 12 where it can be seen that 4 threads and 2 MPI gives the best performance similar to *APACHE2* matrix. The best case speedup over the serial version is 4.9 times. The scaling of the BLAS2 routine `facmldlt` used to be very poor and was seriously limiting the scalability of the solver (see results before this section) but the results in this section are with a new version of `facmldlt` which has been rewritten to better exploit locality of data. The new version does not rely on BLAS and was parallelized directly with OpenMP, providing slightly better results. The same testcase processed as an unsymmetric matrix using the unsymmetric version of the solver is shown in Table 13. The main difference is that in the unsymmetric case `facm` takes a much smaller percentage of time than with the symmetric indefinite solver. With the latest version of MUMPS the symmetric code actually scales better than the unsymmetric one. The scaling of `facass` and `copyCB` on Nehalem are still poor but were found to be better on Opteron, as shown in Table 14. On Opteron, these parts scale by almost a factor of 2 till 4 threads and then saturate so that we should use a maximum of 4 threads per MPI process to get a good performance.

4.3 Unsymmetric cases

We provided some first remarks on the compared unsymmetric and symmetric behaviour of the solver in the previous subsection. Here, larger un-

Routine	1 thread	2 threads	4 threads	8 threads
factldlt	70.05	36.53	20.39	12.226
facsldlt	28.38	14.78	8.06	5.302
facmldlt	1.007	0.71	0.42	0.35
ldltassniv12	2.056	1.766	1.35	1.447
copyCB	1.59	1.42	1.14	1.23
FactorTime	132.0	77.2	48.5	35.9

Table 11: Time (seconds) spent in the most costly routines as a function of the number of threads. Testcase *Haltere*, Nehalem. The TAU profiler was used inducing a small performance penalty.

	1 MPI 8 threads	2 MPI 4 threads	4 MPI 2 threads	8 MPI 1 thread
FactorTime	30.36	**25.75**	28.8	30.75
SolveTime (1 RHS)	1.18	0.9084	0.5429	0.4544

Table 12: Time spent in the factor and solve phases with mixed MPI and OpenMP on 8 cores. Testcase *Haltere*, Nehalem, compiling without TAU. Serial version (1 MPI, 1 thread) takes 126 seconds.

Routine	1 thread	2 threads	4 threads	8 threads
facp	147.83	76.6	40.075	21.99
facq	36.14	19.8	10.61	7.65
facm	0.21	0.21	0.22	0.23
facass	9.21	7.695	6.718	6.545
copyCB	2.988	2.545ec	2.446	2.28
FactorTime	211.5	120.02	75.68	55.11

Table 13: Time (seconds) spent in the most costly routines as a function of the number of threads. Testcase *Haltere_unsym*, processed as an unsymmetric problem, Nehalem.

Routine	1 thread	2 threads	4 threads	8 threads
facp	315	162	86	50.1
facq	75.7	39.72	21.5	13.57
facm	0.73	0.75	0.79	0.8
facass	28.35	23.42	20.56	21.02
copyCB	9.867	6.98	4.776	6.02
FactorTime	577	261	162	119

Table 14: Time (seconds) spent in the most costly routines as a function of the number of threads. Testcase *Haltere*, processed as an unsymmetric problem, Opteron.

Routine	1 thread	2 threads	4 threads	8 threads
`facp`	119.95	61.97	32.165	17.66
`facq`	29.15	15.73	8.24	5.611
`facm`	0.35	0.36	0.36	0.38
`facass`	7	5.98	5.277	5.49
`copyCB`	2.32	2.20	2.06	1.92
FactorTime	166.17	93.93	55.68	39.02

Table 15: Time (seconds) spent in the most costly routines as a function of the number of threads. Testcase *A1M*, Nehalem. The TAU profiler was used.

	1 MPI 8 threads	2 MPI 4 threads	4 MPI 2 threads	8 MPI 1 thread
FactorTime	35.72	**34.98**	35.89	47.18
SolveTime (1 RHS)	0.6977	0.5283	0.4533	0.39

Table 16: Time (seconds) spent in the factor and solve phases with mixed MPI and OpenMP on 8 cores. Testcase *A1M*, Nehalem, compiling without TAU. Serial version (1 MPI, 1 thread) takes 162 seconds.

	1 MPI 8 threads	2 MPI 4 threads	4 MPI 2 threads	8 MPI 1 thread
FactorTime	107.332	**90.409**	96.19	127.467

Table 17: Time (seconds) spent in the factor phase with mixed MPI and OpenMP on 8 cores. Testcase *A1M*, Opteron. Serial version (1 MPI, 1 thread) takes 424.8 seconds.

symmetric testcases were benchmarked using the matrices from the French-Israeli Multicomputing project. They are generated from the discretization of 3-dimensional Navier-Stokes operator and yield saddle-point matrix structure. We present here two cases - *AMAT60* ($N = 965215$, $NZ = 7458355$) which is a saddle-point matrix with a large zero block at the end; and *A1M* which is a submatrix of *AMAT60* with no zeros on diagonal. The scaling of testcase *A1M* ($N = 738235$, $NZ = 4756918$) is shown in Tables 15, 16 and 17, where the scaling results are similar to the unsymmetric case discussed in the previous subsection. The best case speed-up is obtained with 4 threads and 2 MPI processes, which is 4.63 on Nehalem and 4.70 on Opteron. The scaling of the bigger case is shown in Table 18, which shows considerable gains in all routines except `facm` up to 8 threads. This indicates that the routine `facm` should probably be rewritten with more attention paid to locality, similar to what was done for `facmldlt` (see first paragraph of Section 4.2. Results of the mixed strategy are given in Table 19 and show the best case performance with 8 threads (an overall speedup of 6.22 over the serial version). This is similar to the result of the *FDTD* testcase which also showed best results with 8 threads, implying that for very large cases 8 threads per MPI process currently give better performance than 4 threads per MPI process.

Routine	1 thread	2 threads	4 threads	8 threads
`facp`	2876	1424	723	379
`facq`	780	385.89	195	115.4
`facm`	59.2	56	57.6	57.4
`facass`	85.9	55.5	41.1	37.9
`copyCB`	33.1	22.5	17.1	15.5
FactorTime	3884.4	2013.6	1068.0	624.2

Table 18: Time (seconds) spent in the most costly routines as a function of the number of threads. Testcase *AMAT60*, Opteron. The TAU profiler was used.

	1 MPI 8 threads	4 MPI 2 threads	2 MPI 4 threads	8 MPI 1 thread
FactorTime	**624**	807	768	NA
SolveTime (1 RHS)	8.26	4.17	6.08	NA

Table 19: Time (seconds) spent in the factor and solve phases with mixed MPI and OpenMP on 8 cores. Testcase *AMAT60*, Opteron. Serial version (1MPI, 1 thread) takes 3884 seconds. NA means that the result is not available because of insufficient memory.

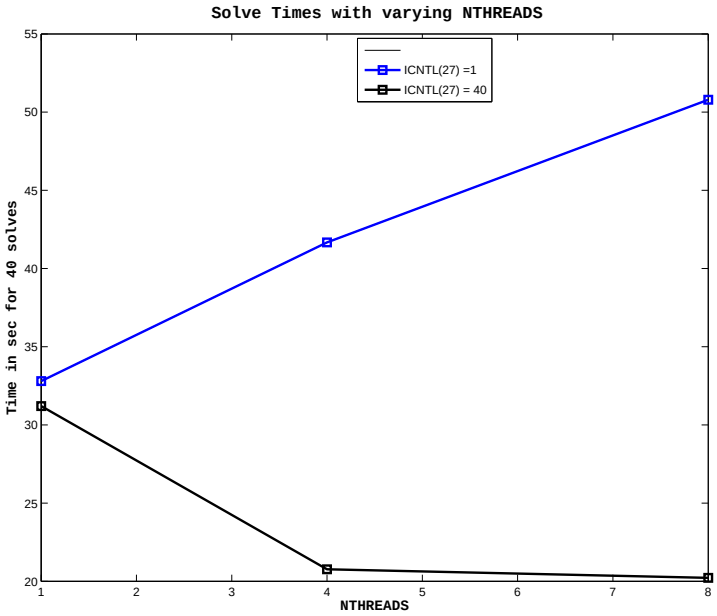

Figure 5: Time spent in the solve phase to solve 40 right-hand side vectors as a function of the number of threads for two different blocking factors (ICNTL(27)). Testcase *A1M*, Opteron.

4.4 Solve Phase

In direct solvers, factorization time is the dominant cost whereas solve time is not in case of a small number of right hand sides. But this cost becomes dominant in the cases when the number of right-hand sides is big, or when an application iterates on successive solves without refactoring the matrix. Some regions of the solve phase use BLAS operations, while some small loops in the solve phase were multithreaded. Single right-hand side uses BLAS2, while multiple right-hand sides use BLAS3. Therefore we expect to see scaling as we increase the number of right-hand sides and the blocking factor during the solve phase. This blocking factor, ICNTL(27), corresponds to the number of columns of right-hand side treated simultaneously. Notice that in the current version of the code, this blocking factor cannot be very large due to memory constraints, even if we have a very large number of right-hand sides simultaneously. The scaling of the solve phase is shown in Figure 5, line "SolveTime", where we indeed see slowdowns with lower ICNTL(27) for a constant number of solves. With ICNTL(27) bigger than 4, the scaling behaviour improves although it tends to saturate with more than 4 threads (similar to the factorization phase).

5 Issues and discussion on Multithreading

Tuning a mixed MPI/OpenMP solver is a challenging task [17] because of various compiler, platform and architecture issues. In our approach, the OpenMP directives form a high level abstraction which does not provide transparency as to how the compiler and operating system handle the multithreaded tasks. They cannot yet be trusted to provide the best solution for all the issues involving multithreading. Different authors report different findings involving the use of OpenMP and POSIX threads [8] for handling multithreading - such as thread affinity and thread activity. The leading issues one should keep in mind are the following - data locality, thread activity and thread affinity. In the following subsections we report on the important issues during multithreading MUMPS.

5.1 Thread Affinity

Thread affinity describes how likely a thread is to run on a particular core. By setting the thread affinity, we can bind a thread to a particular core which may be beneficial to the performance of numerical kernels, particularly in cases where the NUMA factor[7] is large. To our knowledge there are two ways to set thread affinity - one is through the use of libnuma, and the other is through the environment variable KMP_AFFINITY, when INTEL compilers are used. We have used the latter method for experimenting with the effect of thread affinity. There can be three values of this variable - none (default), compact and scatter. Intuitively speaking the default mode should perform the best since other modes interfere with the operating system's ability to schedule tasks. On Opteron no effect of setting the thread affinity was observed. For smaller testcases we observed a small effect of this variable on Nehalem, as depicted in Table 20. For small datasets the compact mode seems to perform the best in OpenMP regions, and consequently there is a small difference in the net factorization time. For example, in *APACHE2* factorization takes around 1.5 seconds more in default mode and this difference was constant between several runs. However for larger cases this effect was not noticeable. From this we can conclude that setting the thread affinity has a stabilizing effect when smaller datasets with low granularity are processed in the multithreaded regions. It does not seem to affect scalability by a great deal. The thread affinity effect in our experiments was visible only on Nehalem architectures, not on Opteron. One reason of this might be that in our approach we do not do any specific effort on locality aspects, for example, we do not force the same thread to work on the same data during different tasks of the computation. We do not have

[7] In NUMA architectures, the NUMA factor shows the difference in latency for accessing dada from a local memory location as opposed to a non-local one.

	2 threads			4 threads			8 threads		
	compact	none	scatter	compact	none	scatter	compact	none	scatter
facass	527	634	594	490	521	510	560	518	492
copyCB	480	701	665	465	673	625	687	898	677
faci	106	162	163	99	147	141	146	244	141

Table 20: Effect of KMP_AFFINITY on the time (milliseconds) spent in assembly (facass), memory copies (copyCB), and pivot search operations (facildlt). Nehalem platform, testcase *APACHE2*, using the symmetric indefinite version of the solver.

any control either on the status of the various levels of cache after calls to multithreads BLAS.

5.2 Memory consumption

Using a larger number of threads per node decreases the memory requirements. This is because larger number of MPI processes require some duplication of data during different parts of the factorization phase, and because the current memory scalability with the number of MPI processes has some potential for improvements (see last Chapter of [1]). The shared memory environments do not require data duplication and hence when memory is limited using larger number of threads per MPI process is advisable. For example, the example problem *AMAT60* in Table 18 consumes a total of 17.98 GB on one MPI process, 23.1 GB on two MPI processes and 23.21 GB on four MPI processes. The *APACHE2* testcase takes 1596 MBytes on one MPI process, 1772 MBytes on two MPI processes, 1968 MBytes on four MPI processes and 2416 MBytes eight MPI processes.

5.3 Mixed MPI/OpenMP Strategy

As was shown in the experiments a mixed MPI/OpenMP strategy helps to gain speed as well. Intuitively, MPI implementation is faster because of natural data locality but suffers from large overhead of communication costs. OpenMP can help reduce the number of communications so using small number of threads is better. However as was shown in the experiments, in MUMPS there is almost linear speedup in the BLAS operations (which constitute the dominant costs) for smaller number of threads and they saturate as the number is increased due to physical limitations in the memory bandwidth and shared cache. So for the MUMPS solver, MPI processes should be increased when the saturation sets in order to maintain optimum performance. For eight cores, 4 threads per MPI process currently seems to be the optimum number for most cases.

5.4 Regulating multithreaded regions

As was shown before, sometimes we may want to avoid forking into multiple threads due to slowdown in small block sizes. OpenMP has different methods for regulating this - such as setting chunk sizes during scheduling, IF statements that can avoid multithreading, or NUMTHREADS statements to fix the number of threads for some regions. For small chunks of tasks multithreading may be avoided using IF statements, giving the same serial performance. For example, the `faci` operations corresponding to the pivot search (see Table 4) required an IF statement to stop multithreading in smaller blocks. However and to our surprise, it was found that we actually have a slight degradation from the serial performance even with these statements as shown in Table 21, in the smaller blocks. In the testcase *THREAD* (table 4) most of the time is spent in the bigger blocks resulting into a net speedup. The reason such performance degradation is not understood since intuitively the compiler should follow the same method as in serial execution for the smaller blocks. For getting performance gains within the OpenMP regions that are memory-bound, the granularity of tasks has to be important: Table 22 shows the performance for different block sizes in the `copyCB` operations. Since most of the time is spent in the in smaller blocks the overall gain is smaller in this testcase. Furthermore, in the symmetric case, these `copyCB` operations are performed on triangular matrices (so called *triangular loops*, which are harder to parallelize. Some improvements may consist in forcing smaller chunks, or in selecting a dynamic "GUIDED" scheduler, or in trying to use the COLLAPSE keyword from the OpenMP 3.0 standard, or in defining manually the first and last row for each threads in order to balance the work. This has to be studied further. Dynamic scheduling might have a significant cost on the numerous small blocks and small chunks may be dangerous: during the `copyCB` operations chunks of memory are copied from source to destination areas which are not contiguous as shown in Figure 6. By using smaller chunks, the work would be better balance but thread interleaving may decrease data locality and become critical, if for example two threads need to access the same cache line at a boundary between two chunks.

In all cases, test matrices with too many small fronts or blocks do not exhibit large gains within the threaded regions. This causes non-uniform scaling in different testcases. In order to improve thread parallelism when a lot of time is spent in small tasks, threads would need to exploit the so called tree parallelism rather than just the parallelism within frontal matrices. This is clearly beyond the objectives of this study consisting in multithreading an MPI solver without deep restructuring or rewriting of the code.

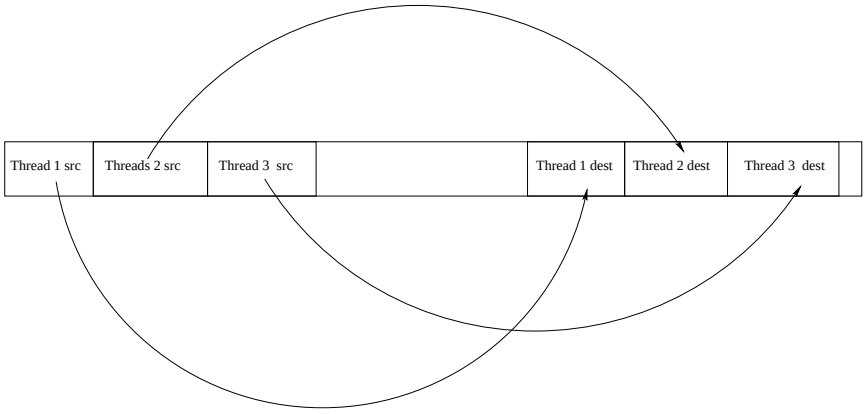

Figure 6: Working areas of threads during CopyCB Operations.

Order of front	1 thread	2 threads with IF	2 threads without IF
0-100	504	767	600
100-200	148	311	131
200-300	188	307	142

Table 21: Effect of an OpenMP IF to avoid multithreading the assembly of contribution blocks smaller than 300. Time spent (milliseconds) in facass operations for different front sizes. Testcase *Haltere*, Nehalem.

Order of contribution	# calls to # copyCB	1 thread	2 threads	4 threads	8 threads
0-500	60056	1823	1431	1204	1508
500-1000	721	1049	887	605	415
1000-1500	140	613	527	358	267
1500-2000	52	432	370	237	186
2000-2500	28	436	370	237	179
2500-3000	15	357	309	193	145
3000-3500	5	155	135	85	62
3500-4000	8	337	295	186	140
4500-5000	3	150	132	84	64
Total	61028	5358	4460	3192	2971

Table 22: Time (milliseconds) spent in copyCB for varying sizes of the contribution blocks. Testcase *Haltere*, Opteron.

5.5 Other issues

There are other issues such as platform issues and thread activities. Different platforms have different architectures and hence demonstrate differing scaling behaviour. For example, machines with better memory bandwidth like Intel Nehalem demonstrate better scaling than previous Xeon machines. Also machines with larger NUMA factors demonstrate poorer scaling with threads and a pure MPI implementation is very competitive. Other authors have shown that keeping threads active all the time also leads to better performance. In MUMPS we follow the fork-join model to execute the threaded regions – this loses some time in activating the sleeping threads once the multithreaded execution starts, as is particularly visible on small frontal matrices. The cache and NUMA penalties are particularly critical on these small frontal matrices at the bottom of the tree, where the threaded approach would benefit from exploiting the tree parallelism (similar to MPI), as said in the previous section.

Another issue is related to the multithreaded BLAS. Whereas multithreaded BLAS libraries are a natural way to parallelize BLAS operations, it is not simple to control the number of threads within each BLAS call in a portable manner. As an example, consider the routine `factldlt`, performing the BLAS3 updates of the Schur of a symmetric frontal matrix. The way `factldlt` is implemented in MUMPS, many DGEMM BLAS calls are done to perform updates of various rectangular blocks from the lower triangle of the Schur complement. Instead of parallelizing inside each BLAS call, it would make sense to parallelize at the upper level, hopefully obtaining better speed-ups than around 4 on 8 threads for this routine. However, all BLAS calls use the number of threads defined by OMP_NUM_THREADS, which prevents us from doing this[8]. This issue will become critical if tree parallelism is exploited by the threads, because BLAS calls should use 1 thread when tree parallelism is used, and more threads higher in the tree. In order to do this in a portable way, one extreme solution would be to avoid multithreaded BLAS altogether and redo the work of parallelizing above a serial BLAS library. This could help improving data locality and trying to keep assigning the same thread to the same data during the processing of a frontal matrix but getting similar parallel BLAS performance as the existing multithreaded BLAS libraries would probably be very hard. However, this issue would have to be decided before widely distributing a first version of a multithreaded MUMPS library.

[8] With specific BLAS libraries, it should be possible to nest OpenMP parallelism and BLAS parallelism. For example mkl_num_threads could be set dynamically to 1 within the routine `factldlt`, allowing exploiting the OMP_NUM_THREADS threads across the BLAS calls.

6 Concluding Remarks and Future Work

In this report, we presented some of our experience in multithreading the multifrontal solver MUMPS, without deeply restructuring the code. Although an exhaustive study is missing, it was shown that multithreading the solver is useful both in terms of speed and memory, as compared to pure MPI implementations. We get almost scalable results for all cases (unsymmetric, symmetric positive definite and symmetric indefinite cases), and a mixed MPI and OpenMP strategy often yields an increased performance. 4 threads per MPI process seem to provide the best performance for most cases, whereas for the bigger, computationally more intensive cases, 8 threads work better. By mixing MPI and thread parallelism we were able to obtain speed-ups around 6 on 8 cores without deeply rewriting the code. Scalability was demonstrated on two different platforms without architecture-specific tuning of the code. Some limitations of thread parallelism have been shown on the small tasks, whose cost cannot be neglected on the range of matrices we have tested. This means that, especially when more threads will have to be used, a plateau can be expected and either tree parallelism has to be exploited at the thread level, or much larger matrices must be targeted. Note that this remark applies to both the factorization and solve phases.

Before that, the factorization and assembly kernels should be studied more deeply to push further the current approach. In particular, the triangular loops of the symmetric code have to be better parallelized, keeping the same thread working on the same rows during various operations would help, and the block sizes must be optimized as a function of the number of threads during the blocked factorizations. Some BLAS 2 factorization kernels (such as `facmldlt`) have started to be rewritten in order to provide better locality but there is still scope for improvements. Also, finding a portable way to exploit parallelism across independent BLAS calls is a critical issue, otherwise serial BLAS has to be used, which means in many cases writing OpenMP code around BLAS calls with smaller sizes.

We also plan to investigate the performance gains on machines with massively parallel nodes and different architectures; on large problems, we expect the relative costs of small tasks to be less critical if the number of threads per MPI task does not grow too much. However, when increasing the number of MPI processes, the amount of work spent in parallelized fronts will become predominant and profiling will be needed to identify the corresponding parts of the code which need to be multithreaded in this new context. For example, the factorization kernels use different routines than the ones from this report when fronts are parallelized. Also, the MPI version heavily relies on the use of *MPI_PACK* (and *MPI_UNPACK*) in the message passing layer. In case these operations appear to be costly, replacing them by multithreaded memory copies might be an option. When

more than one MPI process is used, ScaLAPACK is called at the top of the tree, and the block size and grid shape for ScaLAPACK may need to be adapted to the number of threads used by each MPI process involved in ScaLAPACK computations (note that in that case, multithreaded BLAS *is* required if we do not want to also rewrite ScaLAPACK. This implies that a version of MUMPS relying only on serial BLAS will not be satisfactory for the performance of ScaLAPACK routines).

Acknowledgements

We are grateful to the members of the MUMPS team for many helpful discussions. We also thank Patrick Amestoy and Guillaume Joslin for their comments on a previous draft of this report.

References

[1] E. Agullo. *On the Out-of-core Factorization of Large Sparse Matrices*. PhD thesis, École Normale Supérieure de Lyon, Nov. 2008.

[2] P. R. Amestoy. *Factorization of large sparse matrices based on a multifrontal approach in a multiprocessor environment*. PhD thesis, Institut National Polytechnique de Toulouse, 1991. Available as CERFACS report TH/PA/91/2.

[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 J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods Appl. Mech. Eng.*, 184:501–520, 2000.

[5] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136–156, 2006.

[6] M. Benzi, G. H. Golub, and J. Liesen. Numerical solution of saddle-point problems. *Acta Numerica*, 14:1–137, 2005.

[7] R. Bridson. An ordering method for the direct solution of saddle-pont matrices. Preprint available from http://www.cs.ubc.ca/r̃bridson/kktdirect/.

[8] A. M. Castaldo and R. C. Whaley. Minimizing startup costs for performance-critical threading. In *IPDPS '09: Proceedings of the*

2009 IEEE International Symposium on Parallel&Distributed Processing, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[9] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. Technical Report LAPACK Working Note 95, CS-95-283, University of Tennessee, 1995.

[10] T. A. Davis. University of Florida sparse matrix collection, 2002. http://www.cise.ufl.edu/research/sparse/matrices.

[11] 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.

[12] I. S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear systems. *SIAM Journal on Scientific and Statistical Computing*, 5:633–641, 1984.

[13] Y. Feldman and A. Y. Gelfgat. On pressure-velocity coupled time-integration of incompressible navier-stokes equations using direct inversion of stokes operator or accelerated multigrid technique. *Comput. Struct.*, 87(11-12):710–720, 2009.

[14] D. Gope, I. Chowdhury, and V. Jandhyala. DiMES: multilevel fast direct solver based on multipole expansions for parasitic extraction of massively coupled 3d microelectronic structures. In *42nd Design Automation Conference*, pages 159–162, June 2005.

[15] A. Gupta. A shared- and distributed-memory parallel general sparse direct solver. *Applicable Algebra in Engineering, Communication and Computing*, 18:263–277, 2007. 10.1007/s00200-007-0037-x.

[16] J. Hogg, J. K. Reid, and J. A. Scott. Design of a multicore sparse cholesky factorization using DAGs. Technical Report RAL-TR-2009-027, Rutherford Appleton Laboratory, 2009.

[17] A. Rane and D. Stanzione. Experiences in tuning performance of hybrid mpi/openmp applications on quad-core systems. In *Proceedings of the 10th LCI Int'l Conference on High-Performance Clustured Computing*, 2009.

[18] S. S. Shende and A. D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.

[19] Grid tlse. http://www.gridtlse.org/.

[20] M. Tuma. A note on the LDLT decomposition of matrices from saddle-point problems. *SIAM J. Matrix Anal. Appl.*, 23(4):903–915, 2001.

This appendix contains three sections:

- some guidelines to use several threads with this version of MUMPS, which relies on OpenMP and a multithreaded BLAS library,
- a remark on the saddle-point problems arising from the French-Israeli Multicomputing project,
- a short description of the MUMPS routines cited in this report.

A User guidelines for using multithreaded feature in MUMPS

1. For using multithreaded features in MUMPS the following are currently required - an OpenMP Fortran compiler and a threaded BLAS library (MKL BLAS library is recommended). The only extra flags required both during compilation and linking is -openmp (when Intel ifort is used) or -fopenmp (when GNU compiler is used). The number of threads can be set using the environment variable OMP_NUM_THREADS.
2. The number of threads required for optimum performance depends on the testcase. In general, the multithreaded parts scale up to a certain number of threads before saturating. For example, on 8-core machines saturation sets in after 4 threads. So, the number of threads per MPI process should be set at the 4 for eight-core machines for best performance of most testcases.
3. For very large problems that require large memory using maximum number of threads is beneficial to restrict the total memory usage per node.
4. If Intel compilers are used setting KMP_AFFINITY=compact is recommended.

B Remark on Saddle Point matrices

In the French-Israel Multicomputing project targeting computational fluid dynamics applications, the matrices arise from are a class of saddle point problems, the fast solution of which has been the topic of several research efforts. The sparsity structure of the cases we handled is shown in Figure 7. A class of formulations arising from Navier-Stokes equations typically yield

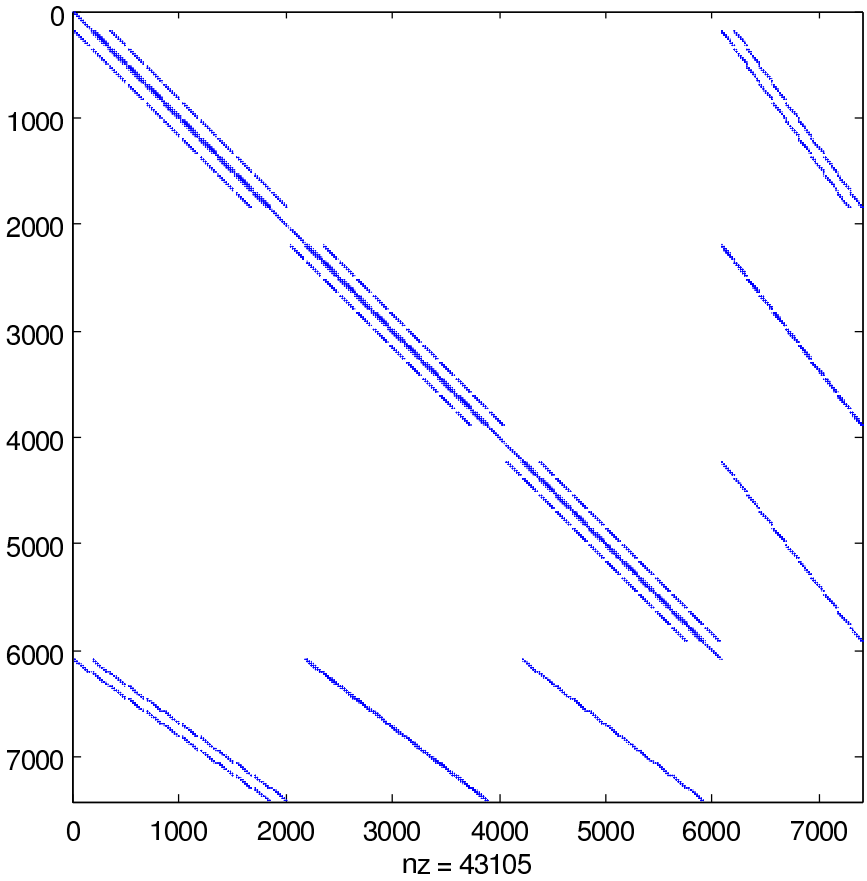

Figure 7: Sparsity pattern of a small Israeli Matrix.

such a structure, where there is a large zero block at the end. The formulation yielding our testcases is described in [13]. A number of direct, iterative and hybrid methods [6] have been proposed to treat such problems. Factorization in direct methods pose a big challenge because of a large number of delayed pivots occurring due to the zero block, for example the number of delayed pivots produced by MUMPS in the *AMAT60* testcase was 319246. Some methods in this area are based on finding a stable permutation without zero pivots on diagonal (see [20, 7]), which essentially are alternate reordering schemes. To get rid of the huge number of delayed pivots we experimented with some of these reordering methods, however a simple hybrid direct factorization of the zero free block followed by an iterative solution of the Schur complement yielded the best solution. The matrix can be represented in the following form:

$$\mathbf{Amat} = \begin{pmatrix} A_{11} & B \\ B^T & 0 \end{pmatrix}$$

Here A_{11} is a zero-free diagonal block consisting of velocity equations in three directions. It can be factorized rather fast using MUMPS and a standard reordering method, compared to the whole **Amat** matrix. Otherwise the three blocks in the independent three directions can be factorized separately to form the LU decomposition of the A_{11} block, and since the order of each sub-matrix is reduced by a factor of 4 the factorization time of each block is small compared to the whole **Amat**. Subsequently, the Schur complement $S = B^T {A_{11}}^{-1} B$ can be solved iteratively. In our case, the Schur complement was found to be well conditioned and we used GMRES. We observed that the number of GMRES iterations remained almost constant with increasing discretization. We do not have a proof of the well conditioning of S, however even if a preconditioner is required this still seems to be better than the direct method given the large reduction of factorization time. S is not formed explicitly (it was found to be dense), only the factors of the three blocks of A_{11} are formed. To compute the matrix-vector products $S.v = w$ during GMRES iterations, first we compute the sparse matrix-vector product $v_1 = B.v$, and then three MUMPS solves are required for computing $v_2 = {A_{11}}^{-1}.v_1$, which requires solves for each of the three blocks. Finally computing $w = B^T.v_2$ forms the required matrix-vector product during each iteration. For the 1 million case (testcase *AMAT60* on 8 cores of the Opteron platform (we use here 8 MPI processes with 1 thread each), the factorization time for the direct method was 3625 seconds and the solve time was 17 seconds. With the same number of nodes, the hybrid approach took 314 seconds for factorization and 45.6 seconds for solve with 38 iterations and a tolerance of 10^{-9}. The increase in solve time is not very large since each iteration consists of 3 MUMPS solves (one for each block). Overall there is a gain of almost a 10x factor for this case. The largest testcase (3 million unknowns) could not be solved on 16 cores of the Opteron platform using

the direct method. Hoever, the hybrid method with 16 MPI processes (1 MPI process with 1 thread on each core) took 2553 seconds for factorization and 347 seconds for solve (GMRES tolerance 10^{-9}, 40 iterations).

C Short description of the routines studied in this report

Before describing each of the routines studied in this report, we remind that MUMPS relies on a multifrontal method, where the computations follow a tree. At each node of that tree a so called *frontal matrix* (or *front*) of the form

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \tag{2}$$

is first assembled (assembly operations) using so called *contribution blocks* from the children. A_{11}, A_{12}, A_{12} are called *fully summed* blocks and are factorized, and a Schur complement is built in place of A_{22}. This Schur complement is the *contribution block* that will be used at the parent node to assemble the frontal matrix of the parent. A is stored by rows and, in case the symmetric solver is used, only its lower triangular part is accessed. During factorization the tree is processed from bottom to top using a topological order, so that two independent branches can be treated independently. This is referred to as *tree parallelism*. Both tree and node parallelism must be used when the number of processes increases. A node is said to be *of type 1* if its frontal matrix is processed by a single MPI process, of *type 2* if it is processed by several MPI processes, and of *type 3* if ScaLAPACK is used.

Since this report focuses on the factorization and solve phases, the routines **analdriver** (main driver), **analgnew** (construction of a matrix graph) and **analf** (ordering and symbolic factorization) which appear in Table 3 and are part of the analysis phase will not be detailed.

mumps Main MUMPS driver

factodriver, **facb**, and **facpar** Nested factorization drivers.

factoniv1 Factorization of a type 1 node (frontal matrix associated to the node is processed by a single MPI process and is not distributed among several MPI processes).

facp In the unsymmetric code, update of the parts A_{21} (TRSM) and A_{22} (GEMM), after the Schur complement has been computed.

faci In the unsymmetric code, pivot search with stability check. Requires accessing to at least one row of A.

facildlt In the symmetric case, pivot search with stability check. Requires accessing to at least one column if A.

facm In the unsymmetric code, BLAS 2 factorization of a block of rows from A_{11} and A_{12}.

facq In the unsymmetric code, BLAS 3 update of the remaining rows of A_{11} and A_{12} after `facm`. after a new block of rows has been factorized by facm.

facmldlt In the symmetric code, BLAS 2 factorization of a block of columns from A_{11} and A_{21}.

facsldlt In the symmetric code, BLAS 3 update of the remaining columns of A_{11} and A_{21}, after `facmldlt`. In the symmetric positive definite solver, A_{21} is not touched at this stage.

factldlt In the symmetric code, BLAS 3 update, by blocks of A_{22}, after A_{11} and A_{21} have been factorized. In the symmetric positive definite solver, A_{21} is first factorized using the A_{11} block (TRSM operation).

facass Assembly of a frontal matrix A at a node k. Requires access to the contribution blocks of the children and extend-add operations with indirections and additions. `facass` also includes the initialization of the entries of a frontal matrix to 0.

ldltassniv12 In the symmetric case, effective assembly operations, called from `facass`.

copyCB Copy of the Schur complement (or contribution block, corresponding to A_{22}) of a frontal matrix from the dense matrix A to a separate location, for later assembly at the parent node.

facstack Stack operation: after a frontal matrix A has been factorized and its Schur complement has moved, factors are made contiguous in memory. For type 1 nodes, this implies copies of A_{21} entries in the unsymmetric case, and copies of entries in both A_{11} and A_{21} in the symmetric case.

facstosendarrowheads modify storage of initial matrix into a so called *arrowhead format* for easier assembly into the frontal matrices at each node of the multifrontal tree. In case several MPI processes are used, this routine also sends the elements of the initial matrix from the MPI process with rank 0 to other MPI processes (assuming the initial matrix is centralized).

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399