Applied Mathematics, 2015, 6, 1276-1281
Published Online July 2015 in SciRes. http://www.scirp.org/journal/am
http://dx.doi.org/10.4236/am.2015.68121

Scientific Research Publishing

Intel® Math Kernel Library PARDISO* for Intel® Xeon Phi™ Manycore Coprocessor

Alexander Kalinkin, Anton Anders, Roman Anders

Intel Corporation, Software and Services Group (SSG), Novosibirsk, Russia
Email: alexander.a.kalinkin@intel.com, anton.anders@intel.com, roman.anders@intel.com

Received 16 June 2015; accepted 18 July 2015; published 22 July 2015

Abstract

The paper describes an efficient direct method to solve an equation $Ax = b$, where A is a sparse matrix, on the Intel® Xeon Phi™ coprocessor. The main challenge for such a system is how to engage all available threads (about 240) and how to reduce OpenMP* synchronization overhead, which is very expensive for hundreds of threads. The method consists of decomposing A into a product of lower-triangular, diagonal, and upper triangular matrices followed by solves of the resulting three subsystems. The main idea is based on the hybrid parallel algorithm used in the Intel® Math Kernel Library Parallel Direct Sparse Solver for Clusters [1]. Our implementation exploits a static scheduling algorithm during the factorization step to reduce OpenMP synchronization overhead. To effectively engage all available threads, a three-level approach of parallelization is used. Furthermore, we demonstrate that our implementation can perform up to 100 times better on factorization step and up to 65 times better in terms of overall performance on the 240 threads of the Intel® Xeon Phi™ coprocessor.

Keywords

Multifrontal Method, Direct Method, Sparse Linear System, HPC, OpenMP*, Intel® MKL, Intel® Xeon Phi™ Coprocessor

1. Introduction

This paper describes a direct method for solving the equation $Ax = b$ with sparse matrix A on Intel® Xeon Phi™ coprocessors. The objective is to utilize effectively all available threads. Additionally, it is very important to reduce OpenMP* [2] synchronization overhead because it has a significant impact on the overall performance on systems with a large number of threads. The main idea of the method is to decompose matrix A into a product of lower-triangular L, diagonal D, and upper triangular U matrices and then to solve the subsystems obtained. To achieve good scalability on a large number of threads a "multi-frontal" approach is used. The multi-frontal ap-

How to cite this paper: Kalinkin, A., Anders, A. and Anders, R. (2015) Intel® Math Kernel Library PARDISO* for Intel® Xeon Phi™ Manycore Coprocessor. *Applied Mathematics*, **6**, 1276-1281. http://dx.doi.org/10.4236/am.2015.68121

proach first proposed in Duff [3] and further expanded by Liu [4]. The algorithm includes several stages: reordering, factor, solve. The initial matrix is subject to a reordering procedure described in Karypis [5] [6] in order to represent it in the form of a dependency tree. A symbolic factorization is then done where the total number of nonzero elements is computed in the form of *LDU* factors. Then actual factorization of the permuted matrix in the *LDU* form is performed as described in Amestoy [7]. In this paper we primarily focus on implementation of the factorization and solve steps. We present an OpenMP implementation of the *LDU* decomposition and solve of the triangular systems obtained based on the hybrid parallel algorithm used in Intel® Math Kernel Library Parallel Direct Sparse Solver for Clusters [1]. This approach demonstrates good performance and scalability on a large number of MPI processes [8]. We describe several approaches to parallelization of *LDU* decomposition using OpenMP. In particular, we use a three-level approach of parallelization to address issues related to the large number of threads (about 240) and expensive synchronization.

2. Algorithm Description

Direct sparse solver typically rely on three stages/steps: reordering, factorization and solve. The reordering step includes METIS [9] functionality to produce fill-in reducing ordering for sparse matrices and symbolic factorization to detect nonzero pattern of the factored matrix, dependency tree, and scheduling for work distribution between threads (see **Figure 1**). The optimization of the first stage is achieved by a loop parallelized via OpenMP and by using nested dissection parallelization from METIS [9].

In this chapter, we optimize algorithms for the last two stages of the direct solver. For most matrices investigated the second stage brings noticeable improvement for the overall performance. So we focus our investigation on this part of the solver.

2.1. Factorization Step

Consider sparse matrix *A*. We begin by performing a permutation of initial matrix to reduce the number of nonzero elements in the matrix *L* in *LDU* decomposition. Then neighboring columns with identical nonzero structure are formed into a super-node [3]—we can view the matrix as a set of super-nodes composed of sub-matrices. Since the initial matrix is sparse, it is possible to factor certain super-nodes independently. Taking into accaunt super-node approach structure of factorized matrix all operation implemented via MKL BLAS and LAPACK functionality [1]. During the symbolic factorization step, we determine the order in which we can perform the numerical factorization and then complete it in parallel. The simplified implementation of this approach can be expressed in **Algorithm 1**.

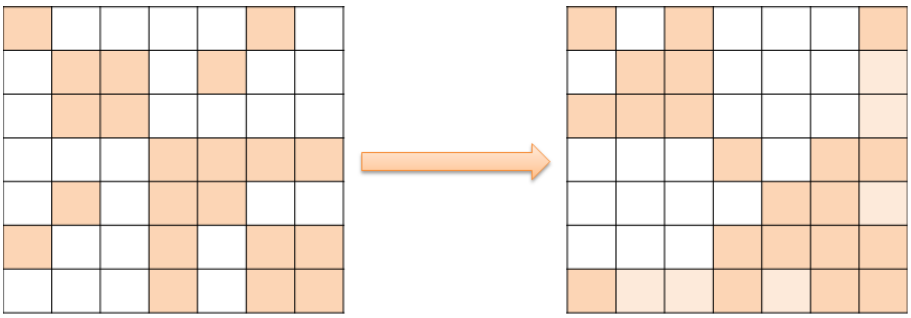


Figure 1. Nonzero pattern of the original matrix (left) and of the same matrix after reordering (right).

Algorithm 1. Simple parallel factored.

```
1  Each thread takes a separate super-node;
2  If all super-nodes on which current depends on are processed then
3  |   factor this super-node via MKL BLAS & LAPACK functionality;
4  else
5  ∟  wait until all needed super-nodes are factored;
6  Take the next super-node and go to 2.
```

This approach shows good performance on a small number of threads, but when their number increases, the performance is reduced due to multiple OpenMP synchronization points.

During the reordering step we compute a dependency tree which allows us to factor the matrix in parallel. **Figure 2** shows a sample matrix and its dependency tree. One can see that it is possible to factor the 1st, 2nd, 4th and 5th blocks independently. Once this is done, blocks 3 and 6 can be processed, and block 7 is factored last. More details about constructing the dependency tree can be found in [10]. From [11], we know that this approach scales well up to 32 MPI processes, while additional MPI processes do not provide significant performance improvements.

We can use the dependency tree to perform the factorization as described in **Algorithm 2**.

Because we store the initial matrix in super-nodal format, during factorization step we deal with dense matrices. The dense matrices can be large, so using BLAS level 3 [1] functionality to compute the matrix-matrix product and to perform the Cholesky decomposition for diagonal blocks helps improve performance. For this purpose, we use four threads that are available on each Intel Xeon Phi core.

Now we formulate a three-level approach to parallelization based on the three algorithms considered earlier. All OpenMP threads are divided into independent groups which are responsible for parallelization according to the dependency tree and all synchronizations take place only inside each group of threads. This will be the first level of parallelization. Then inside each group we can apply **Algorithm 1**. This will be the second level of parallelization. Finally, inside each group we unite two or four (depending on the maximum number of available threads) threads into one group to perform BLAS level 3 functionality in parallel. This will be the third level of parallelization. This three-level approach allows us to utilize all threads effectively. In particular, this distribution of threads between the different groups reduces the OpenMP synchronization overhead that historically negatively impacted performance.

2.2. Solution Step

We now consider the last stage of direct sparse solver, namely the solving of the systems with lower-triangular L, diagonal D and upper triangular U matrices. Similar to the implementation of parallel factorization algorithm, we use two-level parallelization for this case. As before, we distribute all of the leaf nodes of the dependency tree between threads and compute the elements corresponding to these nodes. Then we use respective threads from the child nodes to compute unknowns that correspond to their parent node in the tree. As a result more and more threads are involved in computations of a node as we proceed closer to the top of the tree. This composes the second level of parallelization in our algorithm. To utilize effectively threads collaboratively working on nodes up the tree we apply parallelization algorithm that is similar to that used when we start computations for the whole tree. The scheme of the algorithm on the example of the lower triangular matrix L is shown in **Figure 3**.

The idea described above comes from the left-looking 1D directed acyclic graph approach. First, we distribute the nodes of the tree among super-nodes. Next each thread starts to handle its own super-node.

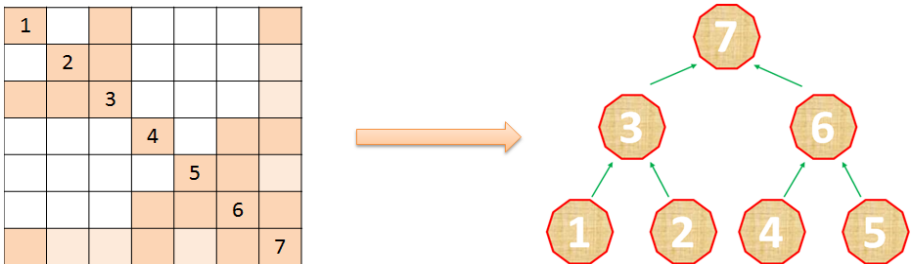


Figure 2. Dependency tree sample.

Algorithm 2. *LDU* decomposition based on the dependency tree.

```
1  for level = bottom_level, top_level do
2  |  Calcuate update of current node by calcluated nodes
3  |  Calculate LDUT decomposition of each node of the current level with a separate thread;
4  end
```

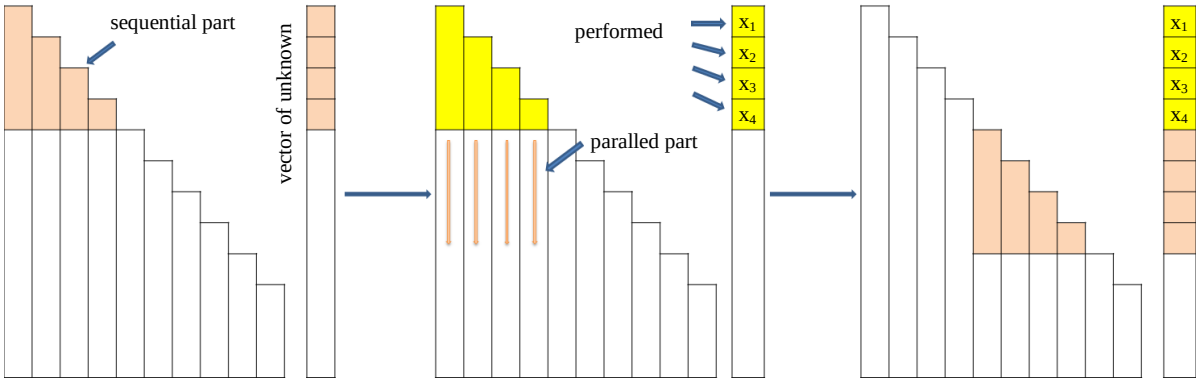


Figure 3. Parallel implementation of solution step.

To prevent multiple synchronizations near the top of the tree we apply a right-looking approach to the root. Thus, to compute unknowns corresponding to any of the nodes of the tree we first update these values with the computed values at the children level.

3. Experimental Results

The platform used for the experiments in this paper is an Intel Xeon Phi coprocessor. The system is equipped with 16 GB GDDR5 memory and includes a 61-core coprocessor running at 1.23 GHz. In this work, we used 60 cores to test the solver, leaving the remaining core to run the operating system and other software.

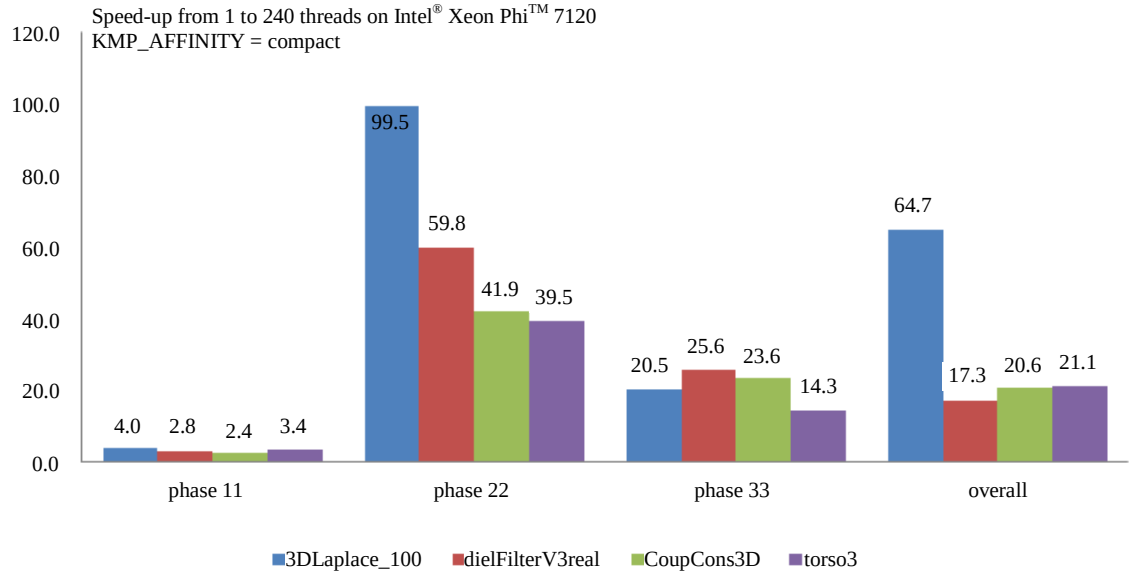
Sparse matrices used in our performance evaluation are taken from The University of Florida Sparse Matrix Collection [12].

On the **Figure 4** one can see the scalability of our implementation from 1 to 240 threads with KMP_AFFINITY = compact setting. The number of columns and nonzero elements for each matrix is as follows: 3DLaplace_100 $N = 10^6$, $NNZ = 39 \times 10^5$; dielFilterV3real $N = 11 \times 10^5$, $NNZ = 45 \times 10^6$; CoupCons3D $N = 4 \times 10^5$, $NNZ = 22 \times 10^6$; and torso3 $N = 25 \times 10^4$, $NNZ = 44 \times 10^5$. The Chart shows that our implementation can achieve up to 100 times better performance during the factorization step using the full coprocessor and up to 65 times improvement in the overall performance. Currently, during the solution step we can effectively utilize about 60 cores (4 threads each). Adding additional threads does not yield any improvement.

Figure 5 demonstrates the impact of every stage to the overall performance. One could see that on one thread factorization has the most impact on performance (about 90%). As the number of threads increases, the reordering step begins to take more than 50% of the total time. As was shown on **Figure 4**, the first stage has low scalability. For example, in spite of scalability for matrix dielFilterV3real on factorization and solving steps being good (60 and 25 times, respectively), the overall performance is low (only about 17 times improvement) because the reordering step takes about 74% of the total computational time (see **Figure 5**). Therefore the next step for the improvements is to optimize reordering and symbolic factorization.

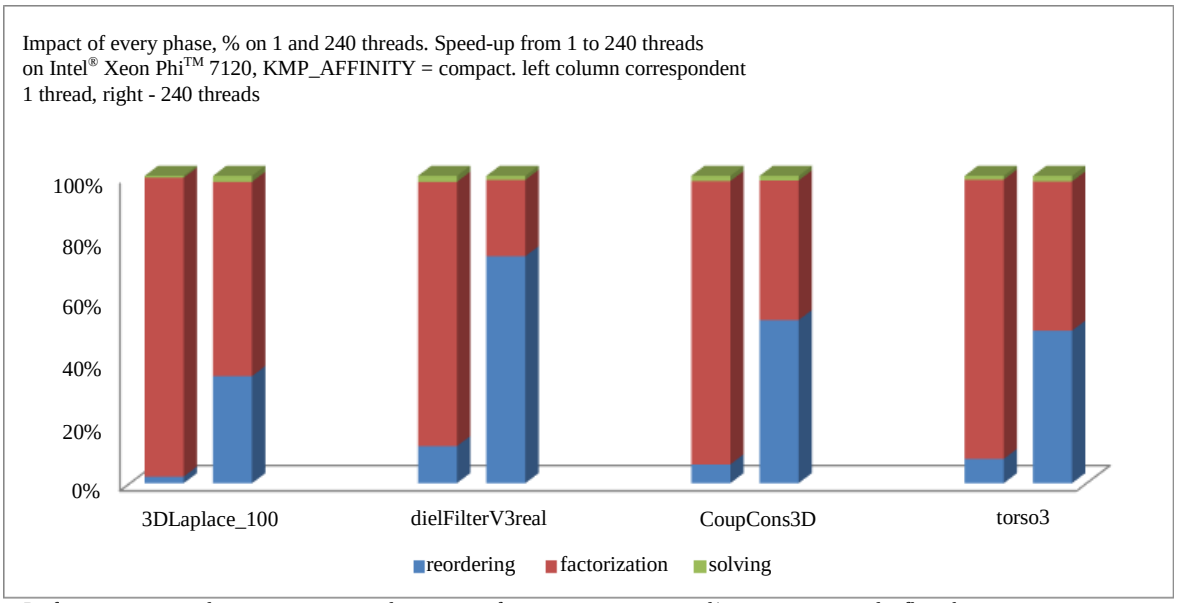
4. Conclusions

In this paper, we presented an efficient implementation of the Intel MKL PARDISO for the Intel Xeon Phi coprocessor. The implementation uses a three-level parallelization approach that allows for a more optimal utilization of all of the available cores. The first level is based on the sparsity of the initial matrix and, as a result, on the sparsity of the factorized one. The second level is related to the dependency tree which can be calculated at the reordering step using METIS [9]. The third level uses four threads available on each core allowing the use of parallel BLAS level 3 functionality for the matrix-matrix multiplication and Cholesky decomposition for the diagonal blocks. We used static work distribution among OpenMP threads to reduce expensive synchronizations. The required distribution can be calculated at symbolic factorization on the reordering step. We showed that our implementation has good scalability for different matrices. It gives up to 100 times performance improvements factorization step and up to 65 times in overall performance. We also proposed an approach to parallelize the solution step. There are still performance issues related to the reordering step which can take more than 50% of the total time for a large number of threads. This is a topic for research.



Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, refer to
http://www.intel.com/content/www/us/en/benchmarks/resources-benchmark-limitations/html
Refer to our Optimization Notice for more information regarding performance and optimization choices in Intel software products at: http://software.intel.com/en-ru/articles/optimization-notice/
*Other brands and names are the property of their respective owners

Figure 4. Speedup of Intel MKL PARDISO for 240 threads.



Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate Performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, refer to
http://www.intel.com/content/www/us/en/benchmarks/resources-benchmark-limitations/html
Refer to our Optimization Notice for more information regarding performance and optimization choices in Intel software products at: http://software.intel.com/en-ru/articles/optimization-notice/
*Other brands and names are the property of their respective owners

Figure 5. Percentage impact of each solver stage for 1 and 240 threads.

The Intel Math Kernel Library now includes several new features such as the Schur complement of a sparse matrix [13] and a cluster version of the direct sparse solver [11]. At the same time, Intel MKL continues to provide high performance functionality on modern Intel processors.

References

[1] Intel Math Kernel Library (Intel MKL). https://software.intel.com/en-us/intel-mkl

[2] OpenMP. http://openmp.org/wp/

[3] Duff, I.S. and Reid, J.K. (1983) The Multifrontal Solution of Indefinite Sparse Symmetric Linear. *ACM Transactions on Mathematical Software*, **9**, 302-325. http://dx.doi.org/10.1145/356044.356047

[4] Liu, W.H.D. (1992) The Multifronal Method for Sparse Matrix Solution: Theory and Practice. *SIAM Review*, **34**, 82-109. http://dx.doi.org/10.1137/1034004

[5] Karypis, G. and Kumar, V. (1996) Parallel Multilevel Graph Partitioning. *Processing of* 10*th International Parallel Symposium*, 314-319.

[6] Karypis, G. and Kumar, V. (1998) A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering. *Journal of Parallel and Distributed Computing*, **48**, 71-85. http://dx.doi.org/10.1006/jpdc.1997.1403

[7] Amestoy, P.R., Duff, I.S., Pralet, S. and Voemel, C. (2003) Adapting a Parallel Sparse Direct Solver to SMP Architectures. *Parallel Computing*, **29**, 1645-1668. http://dx.doi.org/10.1016/j.parco.2003.05.010

[8] Kalinkin, A. (2013) Sparse Linear Algebra Support in Intel Math Kernel Library, Sparse Linear Algebra Solvers for High Performance Computing Workshop, Scarman House, University of Warwick, 8-9 July 2013.

[9] METIS. http://glaros.dtc.umn.edu/gkhome/views/metis

[10] Kalinkin, A. (2013) Intel Direct Sparse Solver for Clusters, a Research Project for Solving Large Sparse Systems of Linear Algebraic Equations on Clusters. *Sparse Days Meeting* 2013 *at CERFACS*, Toulouse, 17-18 June 2013.

[11] Kalinkin, A., Anders, A. and Anders, R. (2014) Intel Math Kernel Library Parallel Direct Sparse Solver for Clusters, EAGE Workshop on High Performance Computing for Upstream, September 2014.

[12] Davis, T.A. and Hu, Y. (2011) The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, **38**, 1:1-1:25. http://www.cise.ufl.edu/research/sparse/matrices

[13] Kalinkin, A., Anders, A. and Anders, R. (2015) Schur Complement Computations in Intel® Math Kernel Library PARDISO. *Applied Mathematics*, **6**, 304-311. http://dx.doi.org/10.4236/am.2015.62028