Intel Direct Sparse Solver for Clusters, a research project for solving large sparse systems of linear algebraic equations

A. Kalinkin, K. Arturov

Intel Corporation

This research covers the Intel® Direct Sparse Solver for Clusters, the software that implements a direct method for solving the Ax=b equation with sparse symmetric matrix A on a cluster. This method, researched by Intel, is based on Cholesky decomposition. To achieve an efficient work balance on a large number of processes, the so-called "multifrontal" approach to Cholesky decomposition is implemented. This software implements parallelization that is based on nodes of the dependency tree and uses MPI, as well as parallelization inside a node of the tree that used OpenMP directives. The article provides a high-level description of the algorithm to distribute the work between both computational nodes and cores within a single node, as well as between different computational nodes. A series of experiments proves that this implementation causes no growth of the computational time and decreases the amount of memory needed for the computations.

1. Introduction

The paper describes a direct method based on Cholesky decomposition for solving the equation Ax=b with sparse symmetric matrix A. The positive-definite matrix A can be represented in terms of LL^T decomposition, in case of an indefinite matrix the decomposition is LDL^T, where the diagonal matrix D can be amended with extra "penalty" for additional stability of the decomposition. To achieve an efficient work balance on a large number of processes, the so-called "multifrontal" approach to Cholesky decomposition is proposed for the original matrix.

The multifrontal approach was proposed in the papers [1-7]. The decomposition algorithm implementation consists of several stages/ The initial matrix is subject to a reordering procedure [8-11] to represent it in the form of a dependency tree. Then the symbolic factorization takes place where the total number of nonzero elements is computed in LL^T. Then a factorization of the permuted matrix in the LL^T form takes place [2-3, 5].

This work is devoted to Intel® Direct Sparse Solver for Clusters package. This package implements parallelization based on nodes of the dependency tree using MPI as well as parallelization inside the node of tree using OpenMP directives. A variant of MPI-parallelization of Cholesky decomposition can be found in [1, 5]. However, as it will be shown later, such an algorithm becomes poorly scaled both in computational time and in the amount of memory used by each process if the total number of processes increases. To avoid scalability issues, we propose an algorithm, where one "tree" node is distributed among several computational nodes, and data transfers are interleaved with the computations preventing the growth of the computational time and reducing the amount of memory required for each process. This paper describes this algorithm.

The paper is organized as follows: Section 2 provides a brief description of the reordering step and describes parallel algorithms to distribute the work between both computational nodes and cores within a single node. Section 3 briefly describes the algorithm distributing a tree node between different computational nodes. Section 4 demonstrates on a series of experiments that this implementation does not result in the growth of the computational time and decreases the amount of memory needed for a process.

.

2. Main definitions and algorithms

In a general case, the algorithm of Cholesky decomposition can be presented in the following way:

Algorithm 0, Cholesky decomposition

```
L = A
for j = 1,size_of_matrix
 { for j = 1,i
  {L(i,j) = L(i,j)-L(i,k)L(k,j), k = 1,j-1
  if (i==j) L(i,j) = sqrt(L(i,j)
  if (i>j)  L(i,j) = L(i,j)/L(j,j)
  }
}
```

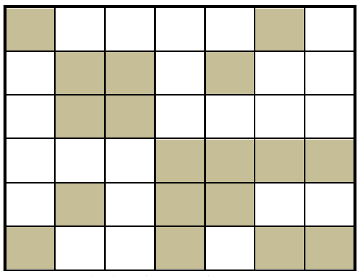
Figure 1. Original matrix non-zero pattern

Here A is a symmetric, positive-define matrix and L is the resulted lower-triangular matrix. If the initial matrix has a lot of zero elements (such kind of a matrix is called sparse), this algorithm can be rewritten in a more efficient way called multifrontal approach.

Suppose we have a sparse symmetric matrix A (Figure 1), where each grey block is in turn a sparse sub-matrix and each white one is a zero matrix. Using reordering algorithm procedures [12], this matrix can be reduced to the pattern as in Figure 2. A reordered matrix is essentially more convenient for computations than the initial one since Cholesky decomposition can start simultaneously from several entry points (for the matrix from Figure 2, 1st, 2nd, 4th and 5th rows of the matrix L can be calculated independently. For the original matrix from Figure 1, two rows only, namely, 1st and 4th, can be calculated independently. So, the reordering can provide an advantage for the algorithm implementation on parallel machines. While proceeding with Cholesky decomposition, the

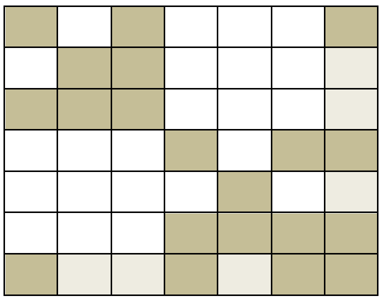
Figure 2. Non-zero pattern of the original matrix after reordering

non-zero pattern of the upper and lower triangular matrices in the final decomposition do not agree with the pattern of the original matrix, that is, additional non-zero blocks may appear (depicted as light-grey squares in the Figure 2). To be precise, a non-zero pattern of the matrix L in Cholesky decomposition is calculated at the symbolic factorization step before the main factorization step (stage). At this stage, we know the structure of the original matrix A after the reordering step and can calculate the non-zero pattern of the matrix L. At the same stage, the original matrix A stored in the sparse format is appended with zeros so that its non-zero pattern matches completely that of the matrix L. Henceforth, we will not distinguish the non-zero patterns of the matrices A and L. Moreover, it should be mentioned here that elements of the matrix L in the rows 3 and 6 can be computed only after the respective elements in the rows 1, 2 and 4, 5 are computed. The elements in the 7th row can be computed at last. This allows us to construct the dependency tree [1-2, 5-6] - a graph where each node corresponds to a single row of the matrix and each graph node can be computed only if its “children” (nodes on which it depends) are computed. The dependency tree for the matrix is given in the Figure 3a (the number in the center of a node shows the respective row number). For our example, an optimal distribution of the nodes between the computational processes is given in the Figure 3b, where the first bottom level of the nodes belongs to the processes with the numbers 0, 1, 2, …, the second level belongs to 0, 2, 4, …, the third belongs to 0, 4, 8, ..., etc.

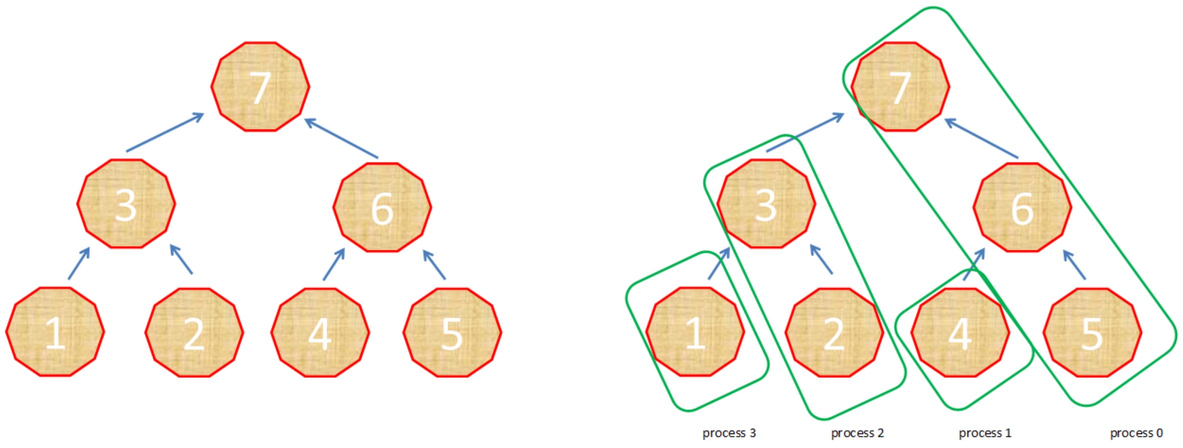

Figure 3a-b. Dependency tree sample distribution among processes

To compute elements of LL^T decomposition within a node Z of the dependency tree, it is necessary to compute the elements of LL^T decomposition in the nodes on which Z depends (its sub-trees). We call an update procedure to bring already computed elements to the node Z (actually, this procedure takes the elements from the sub-trees of Z multiplied by themselves and subtracted from elements of Z, so the main problem here is in a different non-zero pattern of Z and its sub-trees. The 4th line of the Algorithm 0 completes this operation with different L(i,k) being placed into different nodes of the tree, the last step is to calculate LL^T decomposition of elements placed into node Z. Algorithm 1 describes an implementation of the above mentioned piece of the global decomposition in terms of a pseudo-code:

Algorithm 1. Tree-parallelization in Cholesky decomposition

```
for current_level = 1, maximal_level
 {Z = node number that will be updated by this process
  for nodes of the tree with level smaller than the current_level
     {prepare an update of Z by multiplying elements of  LL^T  decomposition elements lying within the current node by themselves}
 send own part of update of Z from each process to process which stores Z
 on process that stores Z compute LLT  elements of Z}
```

Consider in details an implementation of the algorithms called “compute elements of Z” and “prepare an update”.

Each tree node in turn can be represented as a sub-tree or as a square symmetric matrix. To use Algorithm 1, one needs to implement LL^T decomposition of each node on a single computational process. To compute Cholesky decomposition of each node of the tree, a similar algorithm could be implemented on a single computational process with several threads. If the number of threads is equal to the number of nodes of the tree on the bottom level, then LL^T factors for each node from the very bottom level of the tree are calculated by a single thread, on the next level each node can be calculated by 2 threads, on the next one by 4 threads, etc. However, such an algorithm becomes inefficient for the top nodes of the initial tree. Instead, Olaf Schenk suggested some modification of the standard algorithm of Cholesky decomposition for sparse symmetric matrices. The standard LL^T decomposition for sparse symmetric matrices can be described as follows:

Algorithm 2. Classical LL^T decomposition for sparse matrix

```
for row = 1, size_of_matrix
 {for column = 1, row
  {S = A[row, column]
  for all non-zero elements in the row before the column, S = S – A[row, element]*A[element, column]
  // A[row, element]*A[element, column] have been computed
  If column<row A[row, column] = S/A[column, column]
  else A[row, column] = S^(1/2)
  }
}
```

In the paper [7], the following modification of Algorithm 2 for the computers with shared memory is proposed. Each computational thread sequentially selects a row or a set of k rows with similar structure (such set of rows is called supernode) *supr*, for which the following operations are performed.

Algorithm 3. Supernode modifications of Cholesky algorithm

```
for column = 1, supr
 {if column is not computed - wait, otherwise do {A[supr,*] = A[supr,*] – A[column, *]*A[column, column]}
  // A[column, column] could be a square matrix
 }


A[supr, supr] = (A[supr, supr])^(1/2)
// if k ≠1, ½ means Cholesky decomposition of a dense matrix that can be computed with Lapack functions
A[supr, supr+k,..,n] = A[supr, supr+k,..,n] * inv(A[supr, supr)],where inv(B) mean inverse matrix B
```

In his paper, Olaf Schenk applies the algorithm to the entire matrix. In this paper, we apply it to the tree nodes only. Namely this provides an efficient Cholesky decomposition for a dependency tree node of the initial matrix.

The aforementioned Algorithm 3 describes an implementation of the procedure “compute elements of Z” in terms of the Algorithm 1. The procedure “prepare an update” differs only in a sense that each process modifies zero columns with the structure similar to *A[supr,*]* rather than *A[supr,*]* itself. After each process computes its update performing simple summation, we obtain A[supr,*] from which the computed elements of LL^T decomposition have been already deduced.

Thus, we have described the main steps of Algorithm 1. It has a number of drawbacks, however. First, the number of elements of the matrix L in each process differs significantly (for instance in the Figure 3b, the process 0 stores 3 tree nodes, whereas processes 1 and 3 have only one each). As a result, many processes may stay idle expecting the next task to work on. The size of the memory necessary for each process to store elements of the matrix L differs drastically. Following the idea of Algorithm 1, it can be demonstrated that all processes compute an update first, they start transferring data afterwards. This is inefficient since at this time the majority of processes are idle again. To avoid the idle time and distribute elements of LL^T-factors between processes more evenly, we propose an algorithm described in the next Section.

3. Asynchronous execution of processes

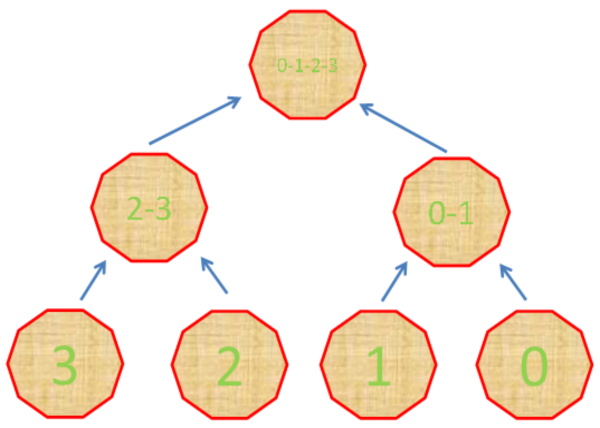

Figure 4. Distribution of nodes among processes

To avoid issues with uneven distribution of the elements of the matrix L, we propose a distribution method as in Figure 4 (digits in the decagons indicate the link of a node with a given process).

Figure 4 demonstrates the same dependence tree as in Figure 3 with the elements from each node of the tree being distributed in different manner between computational processes. At each tree level but the first (bottom) one, the elements of the matrix L are stored on several processes, e.g., at the second level all supernodes are distributed between two processes, at the third one – between four processes, etc. Supernodes from each node of the tree are distributed between n processes as follows: if the total number of supernodes in a certain tree node is m, then the first group of m_1 supernodes belongs to the first process, the next group of m_2 supernodes – to the second process, etc. Here m_1+ m_1+…+ m_n=m. Note that the numbers m_1, m_2,.., m_n may vary that allows one to adjust them in order to provide better performance of the overall algorithm. Then Algorithm 1 is modified so that each process computes its part of the tree node Z. However, this idea does not provide a solution to the problem of keeping processes busy during the computations. Moreover, the problem becomes even bigger since parallel computation of the elements of the matrix L in a single tree node is virtually impossible – almost all supernodes in a single tree node are normally dependent on each other.

Note that each process can be executed by a modern computational node and, therefore, the node can consist of several dozens of individual computational threads. For individual processes to send/receive the data and carry out the computations, we designate one thread in each process to be a “postman”. A “postman” is a thread responsible for data transfer between the processes. Let us consider how Algorithm 2 changes.

3.1 Prepare an update

As it was stated in the Section 2, the Algorithm “prepare an update” is a modification of the Algorithm 3. As was mentioned before, to calculate LL^T factors from node Z of the dependence tree we need to calculate all LL^T factors from its sub-trees and take them into account during computations of LL^T factors of node Z. In general case, however, the node of the tree Z and its sub-tree are stored on different computational processes, so we cannot do it straightforwardly (node Z and its sub-tree have different non-zero pattern, for example). To resolve the issue, the following algorithm is proposed/ Each computational process i allocates matrix Z_i with the same non-zero pattern as the matrix corresponding to the node Z and fills it in with zero elements. Then, all elements from the sub-trees stored on the process i are taken into account in the matrix Z_i as if Z_i is Z (4th line in the Algorithm 0). Further, the computed matrices Z_i are collected on the required process. Considering that we separated a postman thread, there is no need to compute an update first, and send it later, so computations and data transfers can be interleaved.

Algorithm 4. Asynchronous approach in Cholesky decomposition

```
if thread is a postman thread
   {Open a recipient to get updates
   for supr in A_loc
     if supernode supr is computed, send it to the respective process
     else wait until supernode is computed
   }
else
  {create A_loc(all elements in Z) // zero out the elements of node Z with
the same non-zero pattern
   for supr in A_loc
    { if column on the current process
      A_loc[supr,*] = A_loc[supr,*] – A[column, *]*A[column, column]
      // supernode supr is computed
    }
  }
```

It is apparent that having decreased the number of the threads involved in the computations we increased the total time of “update” computations in each process. Nevertheless, the experiments with a big number of threads show that the computational time increases insignificantly. It is also important to note that despite of the increase of the computation time to do the necessary “update”, the transfer of the computed pieces between processes overlaps with these computations. Therefore, the total time spent in the new Algorithm is less compared to the Algorithm 3.

3.2 Compute elements of a tree node

It is much more interesting to consider a more complicated algorithm of computing the elements of the matrix L in a single node of the tree provided that this node is distributed among several processes. Let the elements of the node Z including the elements of the dependent sub-trees (children) that are not processed yet be distributed as it is shown in the Figure 5a, i.e. the first group of supernodes belongs to the first process, and the second group – to the second one, etc. So, each computational process stores only "grey" supernodes.

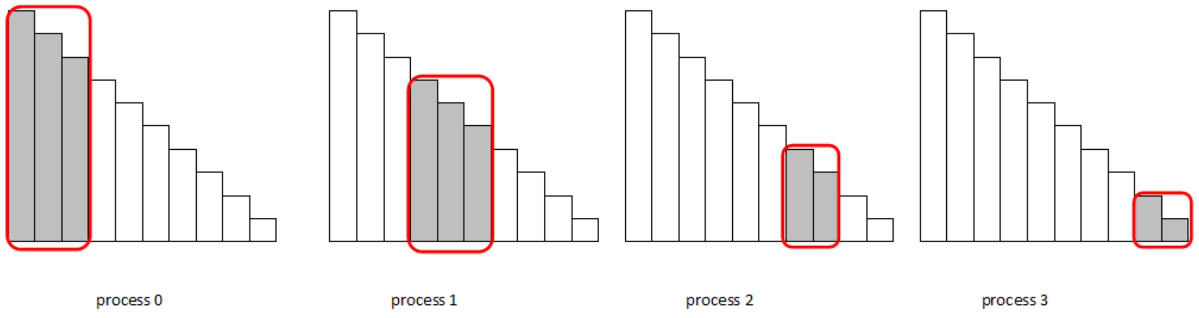

Figure 5a. Supernode distribution among the processes

It can be clearly seen that with such a distribution only the process 0 can start the computations. Thus, the first supernode can only be computed within it. However, after the supernode computation is done, it can be used by the computational threads of the process 0 for other (dependent) supernodes, second it can be sent by the postman thread to other processes, which in turn can use it in their (dependent) supernodes (see the Figure 5b, where blue arrows indicate communications between the processes, the green ones – the update of the supernodes on each process with the supernode received from the process number 0).

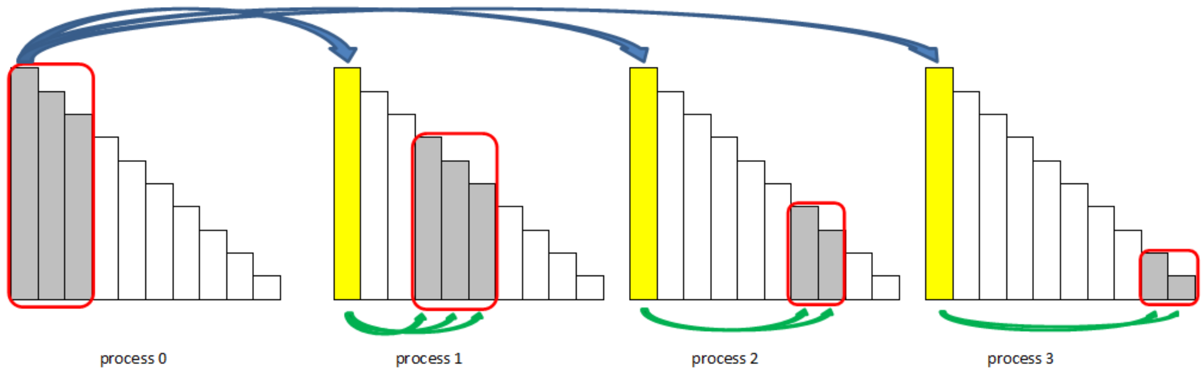

Figure 5b. Computational flow

With this example, it is apparent that if each process has more than 3 computational threads, some threads will simply lack a supernode to process, so making a postman out of one computational thread will have little effect on the overall efficiency if the thread count is big enough. Of course, one supernode can be processed with several threads, but this is not going to be considered in this paper.

4. Numerical experiments

All numerical experiments in this paper were carried on the Infiniband*-linked cluster consisting of 16 computational nodes; each node contains two Intel® Xeon® X5670 processors (12 cores in total) with 48Gb of RAM per node. The variable number of the computational threads within a node is created, i.e. only part of totally available threads is working within the nodes in the most cases.

4.1 Scaling of computational time

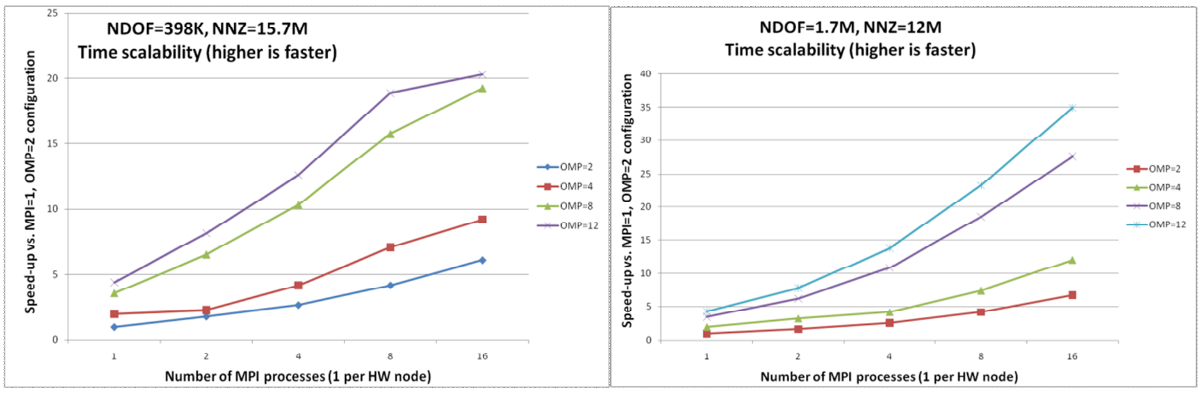

Figure 6a-b. Intel® Direct Sparse Solver for Clusters scalability of time

For this experiment, we selected either 7-diagonal matrix resulted from the approximation of a Helmholtz equation on a uniform grid with a positive coefficient (specific Helmholtz coefficient value is not crucial here since it has no effect on the matrix structure and only the accuracy of the solution obtained depends on it), or the matrix generated from the oil-filtration problem. The number of degrees of freedom (NDOF) for the first matrix is about 398K elements, for the second one is about 1.7M, and the number of nonzero elements (NNZ) in each matrix is 15.7M and 12M, respectively.

Figures 6a-6b show acceleration of Intel® Direct Sparse Solver for Clusters code on a different number of processes compared to the same program launched on 1 MPI process with 2 OpenMP threads on different matrices. The colored lines correspond to a different number of OpenMP threads used in the code. It can be seen from the Figures that the execution time reduces in all cases with the increase of the number of threads and processes. It can be readily seen that sometimes even super-linear acceleration takes place depending on the number of OpenMP threads that can be easily explained from the nature of the algorithm – one thread is used to send & receive data and rather often it falls out of the computations. For example, in the case of 2 threads, one thread is the postman and the other is the computational one, in the case of 4 threads, one thread is the postman and 3 others are computational ones. Based on the Figures, it can be concluded that it is recommended to exploit the computational threads on the node to the maximum and it is not recommended to have several computational processes per one node. From the Figure 6a, it is apparent that the acceleration of the combination 2 MPI x 8 OpenMP is larger than 4 MPI x 4 OpenMP, that in turn, is larger than 8MPI x 2 OpenMP. This perfectly matches the architecture features imposed by the modern computer systems, namely, the growing number of cores (threads) per computational node.

4.2 Memory scaling per node

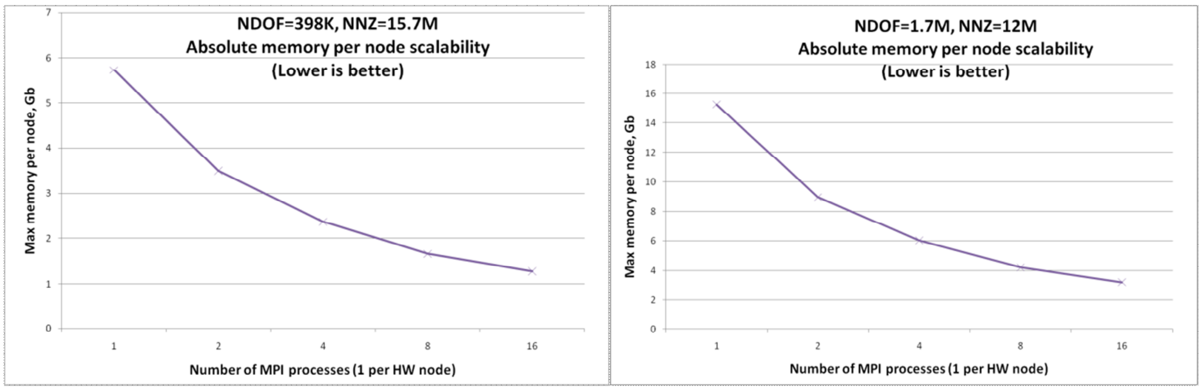

Figure 7a-b. Intel® Direct Sparse Solver for Clusters memory

We use the same matrices as before for the testing purposes. In the Figures 7a-7b, the maximal memory size needed for a computational node depending on the number of nodes is presented. Memory size that Intel® Direct Sparse Solver for Clusters needs for every computational node barely depends on the number of computational threads on it. Therefore, we present the data for the number of threads equal to 12 only. It is apparent that the memory size that every process needs demonstrates invers dependence on the number of processes (for the second matrix, the memory required decreased 5 times for 16 processes vs. 1 process). If the computational cluster has insufficient memory per node, it is still possible to solve the system of linear equations using Intel® Direct Sparse Solver for Clusters package increasing the number of nodes in the cluster. An example of this will be shown in the following

4.3 Solving a huge system of linear equations

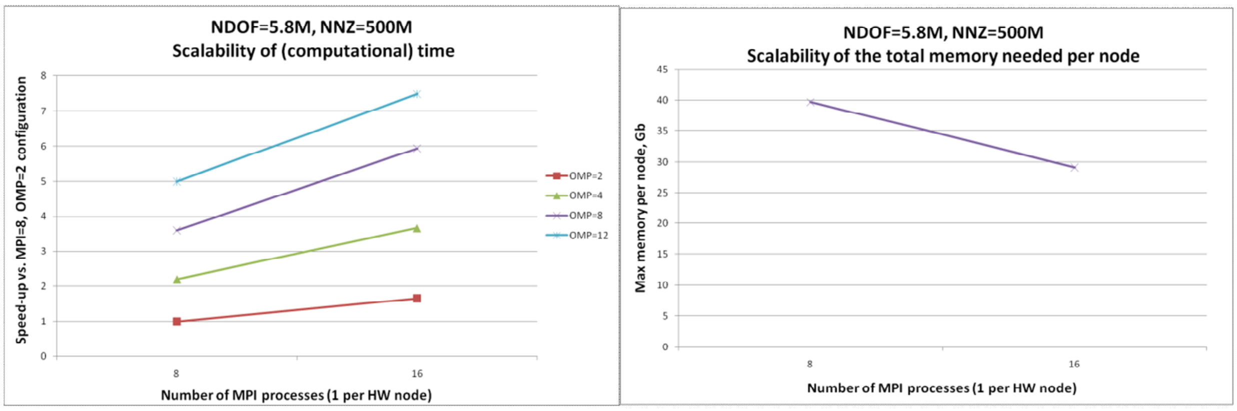

Figure 8a-b. Intel® Direct Sparse Solver for Clusters time and memory scalability, a huge system

In this Subsection, we chose a matrix of size 5.8M with more than half a billion non-zero elements for the experiment. Thanks to Intel® Direct Sparse Solver for Clusters memory scaling, we can solve this system on 8+ MPI processes. Note that almost 40 GB of memory is required per process in case of 8 MPI processes. For 16 processes, 29GB of memory is only required (see Figure 8b). In Figure 8a, the comparison of computational time with the configuration 8 MPI x 2 OpenMP is presented. It is clear that even for such a big matrix size and big number of MPI processes, Intel® Direct Sparse Solver for Clusters shows good scalability both in terms of OpenMP threads and MPI processes.

5. Conclusion

Within the frameworks of multifrontal approach, we proposed an efficient algorithm implementing all stages of Cholesky decomposition inside the node of the dependency tree for all processes on a distributed memory machine. This approach is implemented in Intel® Direct Sparse Solver for Clusters package, and numerical experiments show good scaling in computational time - proportional to the number of computational nodes used and the number of threads within them. Besides, this algorithm reduces the requirement for the memory size used by the algorithm on a single node when the number of processes grows. The experiments made confirm this.

Acknowledgement The authors would like to thank Sergey Gololobov for providing feedback that improved both style and content of the paper.

References

1. P.R.Amestoy, I.S.Duff, C.Vomel, Task scheduling in an asynchronous distributed memory multifrontal solver, SIAM Journal on Matrix Analysis and Applications, Vol 26(2) pp 544--565 (2005)

2. P. Amestoy, I.S. Duff, S. Pralet, C. Voemel, Adapting a parallel sparse direct solver to SMP architectures, Parallel Computing, 29 (11-12), pages 1645-1668.
3. 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.
4. Amestoy, P. R. and Duff, I. S. 1993. Memory management issues in sparse multifrontal methods on multiprocessors. Int. J. Supercomput. Appl. 7, 64--82.
5. P.R. Amestoy, I.S. Duff, J.-Y. L'Excellent, and J. Koster, A fully asynchronous multifrontal solver using distributed dynamic scheduling, SIAM Journal on Matrix Analysis and Applications, 23[1], 15-41 (2001)
6. M. Bollhöfer and O. Schenk, Combinatorial Aspects in Sparse Direct Solvers, GAMM Mitteilungen, 29 (2006), pp. 342-367.
7. Olaf Schenk and Klaus G¨artner. On fast factorization pivoting methods for sparse symmetric indefinite systems. Technical report, Department of Computer Science, University of Basel, 2004
8. A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering. George Karypis and Vipin Kumar. 10th Intl. Parallel Processing Symposium, pp. 314 - 319, 1996.
9. A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering. George Karypis and Vipin Kumar. Journal of Parallel and Distributed Computing, Vol. 48, pp. 71 - 85, 1998
10. Parallel Multilevel Algorithms for Multi-Constraint Graph Partitioning. Kirk Schloegel, George Karypis, and Vipin Kumar. Euro-Par, pp: 296-310, 2000
11. George Karypis, Vipin Kumar: Parallel Multilevel Graph Partitioning. IPPS 1996: 314-319
12. http://glaros.dtc.umn.edu/gkhome/views/metis