Proceedings of the 2014 Federated Conference on Computer Science and Information Systems pp. 569–576
DOI: 10.15439/2014F313
ACSIS, Vol. 2

Performance Analysis of Multicore and Multinodal Implementation of SpMV Operation

Beata Bylina[1], Jarosław Bylina[2],
Przemysław Stpiczyński[3], Dominik Szałkowski[4]
Institute of Mathematics
Maria Curie-Sklodowska University
Lublin, Poland
Email: beata.bylina@umcs.pl[1], jaroslaw.bylina@umcs.pl[2]
przemyslaw.stpiczynski@umcs.pl[3], dominik.szalkowski@umcs.pl[4]

***Abstract*—In this paper we present two algorithms for performing sparse matrix-dense vector multiplication (known as SpMV operation). We show parallel (*multicore*) version of algorithm, which can be efficiently implemented on the contemporary multicore architectures. Next, we show distributed (so-called *multinodal*) version targeted at high performance clusters. Both versions are thoroughly tested using different architectures, compiler tools and sparse matrices of different sizes. Considered matrices comes from The University of Florida Sparse Matrix Collection. The performance of the algorithms is compared to the performance of SpMV routine from widely used Intel Math Kernel Library.**

Keywords: sparse matrix-dense vector multiplication, SpMV operation, parallel matrix-vector multiplication, multicore platforms, computer cluster.

I. Introduction

IN THIS paper we consider multiplication of a sparse matrix by a dense vector, which is called SpMV operation. This operation is fundamental part of many numerical algorithms [4], [8]. In particular SpMV is used for iterative solving of systems of linear equations, e.g. in projective GMRES method or CG method.

Given a $n \times n$ square, sparse matrix $\mathbf{A}$ and a dense vector $\mathbf{x}$ of dimension n we define operation SpMV as

$$\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}.$$

Let us denote ith row of matrix $\mathbf{A}$ by $\mathbf{A}(i, 1 : n)$. Then, to compute ith element of vector $\mathbf{y}$, we have to compute the dot product of $\mathbf{A}(i, 1 : n)$ and $\mathbf{x}$ vectors. So the whole operation of computing $\mathbf{y}$ vector can be easy parallelized, since the computations of all resulting elements are independent. Hence SpMV operation can be treated as n distinct tasks, which have ith row of $\mathbf{A}$ and $\mathbf{x}$ as a input data and produce ith element of $\mathbf{y}$. Note that $\mathbf{x}$ is shared between all computing tasks.

In the paper [11] authors focus on SpMV operation in the case of multicore platforms. They survey some low level optimization techniques related to hardware properties, while using CSR format for storing sparse matrices. All techniques are then benchmarked on a few multicore environments.

In the article [12] authors show a new format suitable for multicore architectures, which they call *Compressed Sparse Block* (*CSB*). It allows effective storage and efficient computations. It also uses special optimizations in the case of multiplication of banded matrices.

The aim of this paper is to present our research on the efficient implementation of a sparse matrix by a dense vector multiplication with the use of contemporary parallel multicore and distributed computer architectures to gain high performance at low cost.

We propose a parallel SpMV algorithm based on modified SPARSKIT library routine [9] targeted at multicore platforms. We investigate the performance of this algorithm using various architectures, compilers and a few sparse matrices, which arises in real life problems. These matrices comes from The University of Florida Sparse Matrix Collection [3]. We include optimized SpMV routine from Intel Math Kernel Library [5] in the comparison.

Next we introduce a distributed algorithm for computing SpMV on computer clusters consisting of multiple nodes. Our universal approach allows to use any existing SpMV implementation locally within one node. For performance comparison we use the same set of sparse matrices as previously.

The paper is structured as follows. Section II describes data structures suitable for representing sparse matrices and their usability for the implementation of SpMV operation. Next section contains short description of standard, sequential SpMV algorithm. In Section IV we present multicore version of existing SPARSKIT SpMV routine. The description of SpMV algorithm for distributed environments is included in Section V. Then we present some numerical results and concluding remarks in sections VI and VII respectively.

II. Storage formats for sparse matrices

Special data structures and algorithms are used for storing sparse matrices (for efficient memory usage) and performing basic mathematical operations. The survey of many storage formats can be found in [8]. Note, that the same formats are used in algorithms designed for sequential and parallel architectures. However, due to different properties of these architectures, different formats may be preferred in each case.

978-83-60810-58-3/$25.00

$$A = \begin{bmatrix} -4 & 0 & 0 & 0 & 1 \\ 0 & -1 & 0 & 8 & 0 \\ 0 & 0 & 0 & 5 & 0 \\ -1 & 31 & 0 & 21 & -1 \\ 0 & 0 & 0 & 0 & -8 \end{bmatrix}$$

Fig. 1. Sparse matrix stored in dense format

$$data = \begin{bmatrix} -4 & 1 & -1 & 8 & 5 & -1 & 31 & 21 & -1 & -8 \end{bmatrix}$$
$$col = \begin{bmatrix} 0 & 4 & 1 & 3 & 3 & 0 & 1 & 3 & 4 & 4 \end{bmatrix}$$
$$row = \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 3 & 3 & 3 & 3 & 4 \end{bmatrix}$$

Fig. 2. Matrix from Fig. 1 stored in COO format

Below we shortly present three widely used formats for storage of sparse matrices. Fig. 1 shows square, sparse matrix of dimension 5 stored in dense format, which, from the programmers point of view, is equivalent to using one two-dimensional array.

A. Coordinate Format (COO)

The simplest and the most flexible format for storing any sparse matrix is so-called *Coordinate Format* or *COO* for short. In this format, only nonzero values are stored, together with rows and columns indexes. Technically, it uses three one-dimensional arrays:

- $data$ for storing nonzero elements,
- col for storing indexes of columns of nonzero elements in the original matrix,
- row for storing indexes of rows of nonzero elements in the original matrix.

On Fig. 2 we see COO storage scheme for the matrix from Fig. 1. Unfortunately, there are some disadvantages of this format, namely it is not memory and computationally efficient (especially in the case of SpMV operation). Note that MATLAB software uses this format [6].

B. Matrix Market Format (MM)

The University of Florida Sparse Matrix Collection [3] is large repository of sparse matrices, which comes from real life applications. It uses Matrix Market format (*MM*) for storing sparse matrices. This format is based on COO with some optimizations added, e.g. it can store only the half of the matrix, in case it is symmetric [7].

C. Compressed Sparse Row Format (CSR)

Another way to store sparse matrix is to use *Compressed Sparse Row* format (*CSR*). As in the case of COO, only the nonzero elements are stored and their columns indexes, while the rows indexes are kept in somewhat different way. There are also three one-dimensional arrays used:

- $data$ which keeps nonzero elements,
- col which keeps indexes of columns of nonzero elements in the original matrix,

$$data = \begin{bmatrix} -4 & 1 & -1 & 8 & 5 & -1 & 31 & 21 & -1 & -8 \end{bmatrix}$$
$$col = \begin{bmatrix} 0 & 4 & 1 & 3 & 3 & 0 & 1 & 3 & 4 & 4 \end{bmatrix}$$
$$ptr = \begin{bmatrix} 0 & 2 & 4 & 5 & 9 & 10 \end{bmatrix}$$

Fig. 3. Matrix from Fig. 1 stored in CSR format

```
      do 100 i = 1,n
         t = 0.0d0
         do 99 k=ptr(i), ptr(i+1)-1
            t = t + data(k)*x(col(k))
 99      continue
         y(i) = t
100   continue
```

Fig. 4. Standard implementation of SpMV for CSR storage

- ptr which keeps indexes of the beginnings of the consecutive rows in $data$ array.

Fig. 3 shows sparse matrix stored in CSR format.

III. Sequential SpMV algorithm

CSR is the most common format used, when dealing with applications containing many SpMV operations. Basic, sequential implementation of SpMV is presented on Fig. 4. We assume that `data`, `col` and `ptr` arrays keeps a sparse matrix in CSR format, while `x` is given vector and `y` is the result of the operation.

IV. Multicore SpMV algorithm

SPARSKIT [9] is Fortran library for dealing with sparse matrices. It provides several formats for storing matrices (including CSR) and routines for performing fundamental mathematical operations. There is SpMV routine in this library for matrices stored in the CSR format, however it is strictly sequential, hence it doesn't take advantage of contemporary parallel architectures. We used OpenMP [10] directives for simple and effective parallelization (use of all present CPU cores) of available source code. The modified source code using `omp parallel do` directive is presented on Fig. 5. We will refer to this algorithm as the *multicore algorithm*.

V. Multinodal SpMV algorithm

In this section we present distributed version of SpMV for clusters consisting of multicore nodes, which we will call the *multinodal algorithm*.

Assume that $\mathbf{A} \in \mathbf{R}^{n \times n}$ matrix is divided into p^2 blocks (with possible different dimensions)

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{00} & \cdots & \mathbf{A}_{0,p-1} \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{p-1,0} & \cdots & \mathbf{A}_{p-1,p-1} \end{bmatrix},$$

```
      subroutine pamux (n, x, y, a,ja,ia)
      real*8  x(*), y(*), a(*)
      integer n, ja(*), ia(*)
      real*8 t
      integer i, k
!$omp parallel do private(t,k)
      do 100 i = 1,n
         t = 0.0d0
         do 99 k=ia(i), ia(i+1)-1
            t = t + a(k)*x(ja(k))
 99      continue
         y(i) = t
100   continue
!$omp end parallel do
      return
      end subroutine pamux
```

Fig. 5. Implementation of SPARSKIT SpMV routine using OpenMP

where $\mathbf{A}_{ij} \in \mathbf{R}^{n_i \times n_j}$ and $\sum_{i=0}^{p-1} n_i = n$. Vectors $\mathbf{x}$ and $\mathbf{y}$ are also divided into p blocks, where $\mathbf{x}_i, \mathbf{y}_i \in \mathbf{R}^{n_i}$. Then

$$\begin{bmatrix} \mathbf{y}_0 \\ \vdots \\ \mathbf{y}_{p-1} \end{bmatrix} \leftarrow \begin{bmatrix} \mathbf{A}_{00} & \dots & \mathbf{A}_{0,p-1} \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{p-1,0} & \dots & \mathbf{A}_{p-1,p-1} \end{bmatrix} \begin{bmatrix} \mathbf{x}_0 \\ \vdots \\ \mathbf{x}_{p-1} \end{bmatrix}$$

and

$$\mathbf{y}_i \leftarrow \sum_{j=0}^{p-1} \mathbf{A}_{ij}\mathbf{x}_j, \qquad i = 0, \dots, p-1. \tag{1}$$

Algorithm 1 describes the multiplication of sparse matrix by dense vector using computer cluster, which has at least p^2 distinct CPUs. The matrix is appropriately distributed between the grid of $p \times p$ tasks denoted by P_{ij}, $0 \le i, j < p$. Algorithm comprises the following steps:

1) sending the data from the first column of tasks to "diagonal" tasks,
2) broadcasting the data columnwise,
3) performing actual computations,
4) performing global reduction.

We assume that each computing task is running on different CPU, however there can be more than one CPU installed in one cluster node.

On figure 6 we see the grid of tasks in the case of dimension 2×2. The grid in the case of 4×4 dimension, together with communication scheme is presented on Fig. 7.

To implement Algorithm 1 we used routines from BLACS (Basic Linear Algebra Communication Subprograms) [1] and MKL (Intel Math Kernel Library) [5] libraries. BLACS routines were used for organizing task grid and transferring data between tasks, while optimized multicore `mkl_dcsrgem` from MKL was used for performing SpMV. The most important part of Fortran implementation of this algorithm is presented on Fig. 8. We used the following variables in the implementation:

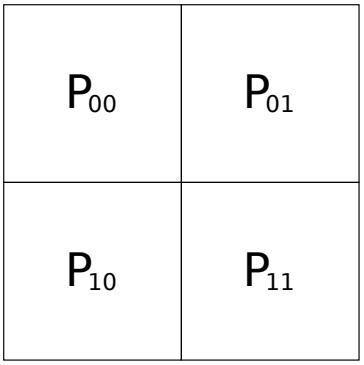

Fig. 6. Task grid of dimension 2×2

- `proc_row`, `proc_col` are coordinates of current task in the task grid,
- arrays `data`, `col`, `ptr` store block $\mathbf{A}_{\texttt{proc\_row,proc\_col}}$ of sparse matrix in CSR format,
- arrays `x` and `y` keep vectors $\mathbf{x}$ and $\mathbf{y}$ respectively,
- `nrows` and `ncols` denotes the number of rows and the number of columns of the $\mathbf{A}_{\texttt{proc\_row,proc\_col}}$ block,
- `context` describes appropriate BLACS context.

Note, that this algorithm is not tied to `mkl_dcsrgem` routine. Instead, it can use implementation from Section IV or any other available code.

VI. Numerical experiments

In this section we review the tests of our SpMV implementations for multicore (Section IV) and distributed (Section V) systems.

A. Data

We used several sparse matrices from The University of Florida Sparse Matrix Collection [3]. All matrices were downloaded in the Matrix Market format and then were converted to CSR format, which was used in all numerical experiments. We present the results for 4 matrices: parabolic_fem, bmw3_2, torso1, nd24k. Detailed parameters are shown in Table I, where we have:

- n is the number of rows and columns,
- nz is the number of nonzero elements,
- $d = nz/n$ denotes the density of the matrix.

Fig. 9 shows the sparsity patterns of these matrices.

Considered matrices were first read from the files and then distributed between all running MPI tasks using BLACS `dgesd2d` routine. We used simple distribution scheme, in which we divided the matrices into the blocks of almost the same sizes.

Note, that instead of reading data from files it is possible to generate matrices locally in each node.

B. Test environment

We used two hardware platforms for testing: E5-2660 and X5650. Their specifications are presented in Table II.

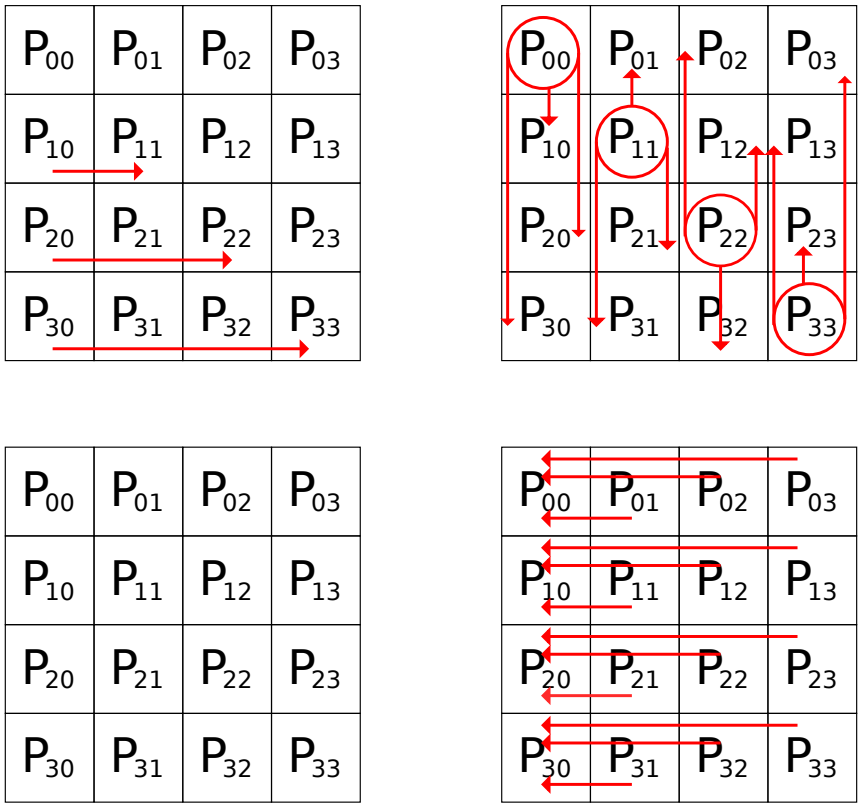

Fig. 7. Communication scheme for the 4×4 task grid: 1) sending the data from the first column of tasks to "diagonal" tasks (top left), 2) broadcasting the data columnwise (top right), 3) performing actual computations (no communication, bottom left), 4) performing global reduction (bottom right)

```
! step 1) sending the data to "diagonal" tasks
   if ((proc_col.eq.0).and.(proc_row.ne.0)) then
      call dgesd2d(context, nrows, 1, y, 1, proc_row, proc_row)
   else
      if ((proc_row.eq.proc_col).and.(proc_row.ne.0)) then
         call dgerv2d(context, nrows, 1, x, 1, proc_row, 0)
      end if
   end if

! step 2) broadcasting the data columnwise
   if (proc_row.eq.pcol) then
      call dgebs2d(context, 'C', ' ', ncols, 1, x, 1)
   else
      call dgebr2d(context, 'C', ' ', ncols, 1, x, 1, proc_col, proc_col)
   end if

! step 3) performing actual computations
   call mkl_dcsrgemv('N', nrows, data, col, ptr, x, y)

! step 4) performing global reduction
   call dgsum2d(context, 'R', ' ', nrows, 1, y, 1, proc_row, 0)
```

Fig. 8. Main part of Fortran implementation of Algorithm 1

Algorithm 1 Outline of the multinodal SpMV algorithm

Require: Each P_{ij} task holds $\mathbf{A}_{ij}$ matrix, each P_{i0}, $0 \leq i < p$, task stores $\mathbf{x}_i$ and $\mathbf{y}_i$
Ensure: Each P_{i0}, $0 \leq i < p$, task holds resulting $\mathbf{y}_i$ vector obtained using equation (1)
1: Each P_{i0}, $0 < i < p$, task sends $\mathbf{x}_i$ to P_{ii}
2: Each P_{jj}, $0 \leq j < p$, task broadcasts $\mathbf{x}_j$ to P_{ij}, $0 \leq i < p$
3: Each P_{ij}, $0 \leq i, j < p$, task performs $\mathbf{t}_{ij} \leftarrow \mathbf{A}_{ij}\mathbf{x}_j$
4: Global reduction $\mathbf{y}_i \leftarrow \sum_{j=0}^{p-1} \mathbf{t}_{ij}$ is performed by P_{ij}, $0 \leq j < p$ tasks {$\mathbf{y}_i$ vector is held by P_{i0}, $0 \leq i < p$}

TABLE I
PARAMETERS OF CONSIDERED SPARSE MATRICES

name	*n*	*nz*	*d*	*symmetricity*
parabolic_fem	525825	3674625	6.98	symmetric
bmw3_2	227632	11288630	49.59	symmetric
torso1	116158	8516500	73.32	symmetric
nd24k	72000	28715634	398.83	non-symmetric

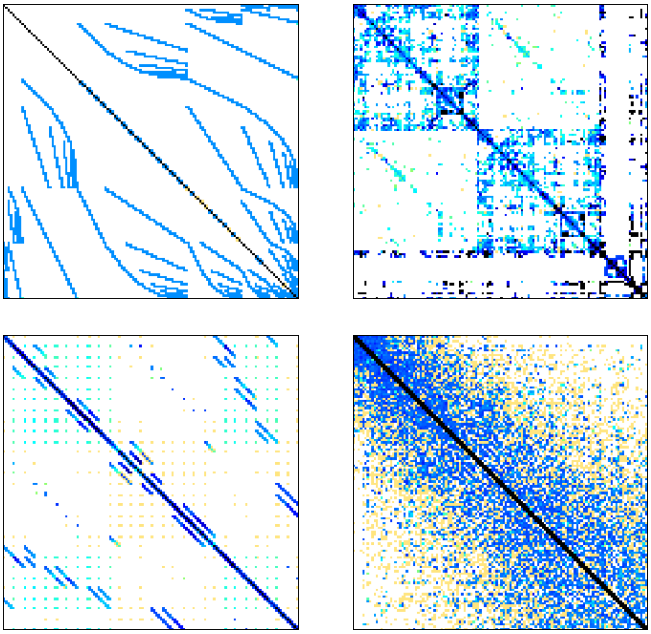

Fig. 9. Sparsity patterns of parabolic_fem (top left), bmw3_2 (top right), torso1 (bottom left) and nd24k (bottom right) matrices

Our algorithms were implemented in Fortran 95 using appropriate parallel and numerical libraries (OpenMP, MPI, SPARSKIT, MKL). Two Fortran compilers, namely `ifort` by Intel and `pgfortran` by The Portland Group, were used for compiling source codes with compiler flags, which are shown in Table III.

For time measurement we used the following routines:

- `omp_get_wtime()` for the multicore version,
- `MPI_Wtime()` in the multinodal case.

C. Results for multicore algorithm

On figures 10, 11, 12 and 13 we see the performance (in Gflops) of the multicore version of SpMV multiplication. The performance is shown for two platforms (X5650, E5-2660), two compilers (Intel, pgi), and we also include the performance chart of SpMV routine from Intel MKL library optimized for multicore CPUs (denoted by mkl).

Using obtained results we conclude that:

- For small number of running threads the performance is similar in each case.
- For growing number of threads E5-2660 architecture outperforms X5650, due to its older architecture. We were expecting this result.
- Compiler version has negligible impact on the performance of the algorithms, however there is a drop in the performance in the case of `pgfortran` dealing with large number of threads.
- Simple SPARSKIT implementation with OpenMP directives (Fig. 5) gives as good performance as the optimized MKL version of SpMV.

TABLE IV
PERFORMANCE (GFLOPS) OF THE MULTINODAL SPMV ALGORITHM

matrix	1 task	4 tasks	16 tasks
parabolic_fem	1.91	0.31	0.41
bmw3_2	2.69	2.02	3.05
torso1	3.12	4.49	8.25
nd24k	3.31	4.39	12.00

D. Results for multinodal algorithm

Table IV shows the performance of our multinodal SpMV implementation (Algorithm 1). In the column denoted by "1 task" we see the performance of the multicore version compiled by `ifort` with MKL support and running on X5650 system. Multinodal version was compiled using `mpiifort` compiler and was running on cluster consisting of 2 or 8 X5650 nodes, connected using 40Gbit/s Infiniband. To optimize the workload of each node we used the following number of MPI tasks:

- 4 tasks were running on 2 nodes with 4 CPUs (as in Fig. 6),
- 16 tasks were running on 8 nodes with 16 CPUs (as in Fig. 7).

Each MPI task was using multithreaded version of SpMV from MKL.

Looking at Table IV we see, that:

- there are cases, when the algorithm achieves very high scalability (e.g. nd24k matrix)
- for some matrices (e.g. parabolic_fem), the performance decreases,

TABLE II
SOFTWARE AND HARDWARE PROPERTIES OF E5-2660 AND X5650 SYSTEMS

	E5-2660 System	X5650 System
CPU	2x Intel E5-2660 (20M Cache, 2.20 GHz, 8 cores with HT)	2x Intel Xeon X5650 (12M Cache, 2.66 GHz, 6 cores with HT)
CPU memory	48GB DDR3	48GB DDR3
Operating system	CentOS 5.5 (Linux 2.6.18-164.el5)	Debian (GNU/Linux 7.0)
Libraries	OpenMP, SPARSKIT, Intel Composer XE 2013	OpenMP, MPI, SPARSKIT, Intel Composer XE 2013
Compilers	The Portland Group, Intel	The Portland Group, Intel

TABLE III
COMPILER FLAGS

Algorithm version	Compiler	Compiler flags
Multicore for E5-2660	`ifort` by Intel	`-O3 -openmp -xAVX`
Multicore for X5650	`ifort` by Intel	`-O3 -openmp -xSSE4.2`
Multicore for E5-2660 and MKL	`ifort` by Intel	`-O3 -openmp -mkl=parallel -xAVX`
Multicore for X5650 and MKL	`ifort` by Intel	`-O3 -openmp -mkl=parallel -xSSE4.2`
Multicore (both systems)	`pgfortran` by The Portland Group	`-O3 -mp -fastsse`
Multinodal for X5650 and MKL	`mpifort` by Intel	`-O3 -openmp -mkl=parallel -xSSE4.2`

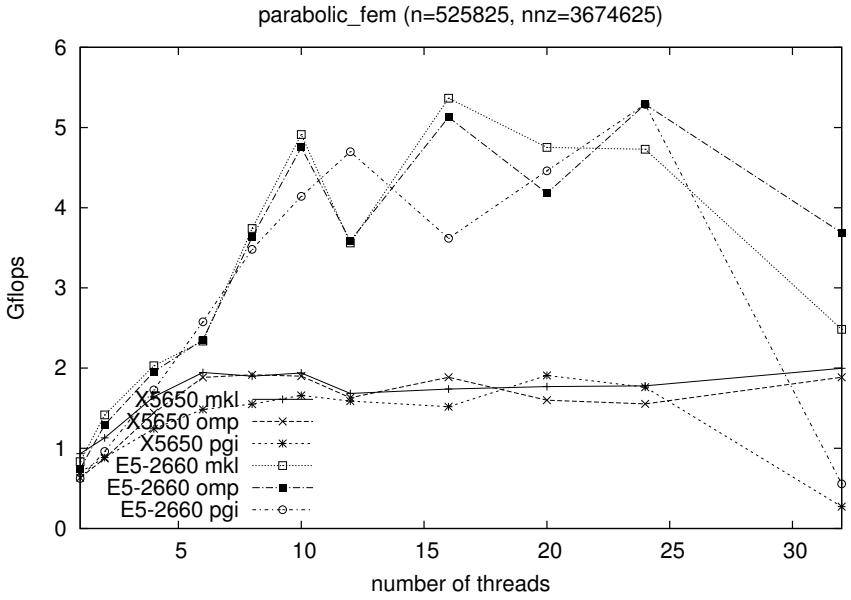

Fig. 10. The performance of SpMV operation for parabolic_fem matrix

TABLE V
DISTRIBUTION OF TORSO1 AND PARABOLIC_FEM MATRICES BETWEEN 4 TASKS

	$torso1$			$parabolic\_fem$		
$name$	n	nz	d	n	nz	d
A_{00}	58079	2407469	41.45	262912	656124	2.50
A_{01}	58079	1693760	29.16	262912	0	0
A_{10}	58079	1941287	33.42	262913	787970	3.00
A_{11}	58079	2473984	42.59	262913	656131	2.50

TABLE VI
DISTRIBUTION OF TORSO1 AND PARABOLIC_FEM MATRICES BETWEEN 16 TASKS

	$torso1$			$parabolic\_fem$		
$name$	n	nz	d	n	nz	d
A_{00}	29039	756119	26.04	131456	131456	1.00
A_{01}	29039	401235	13.82	131456	0	0.00
A_{02}	29039	75227	2.59	131456	0	0.00
A_{03}	29039	589153	20.29	131456	0	0.00
A_{10}	29039	401361	13.82	131456	262142	1.99
A_{11}	29039	848745	29.22	131456	262142	1.99
A_{12}	29039	444083	15.29	131456	0	0.0
A_{13}	29039	585297	20.15	131456	0	0.0
A_{20}	29039	75417	2.59	131456	261886	1.99
A_{21}	29039	444272	15.30	131456	121560	0.92
A_{22}	29039	806323	27.77	131456	241257	1.83
A_{23}	29039	254204	8.75	131456	0	0.00
A_{30}	29041	737556	25.40	131457	262144	1.99
A_{31}	29041	684042	23.55	131457	142380	1.08
A_{32}	29041	254204	8.75	131457	185688	1.41
A_{33}	29041	1159262	39.92	131457	229186	1.74

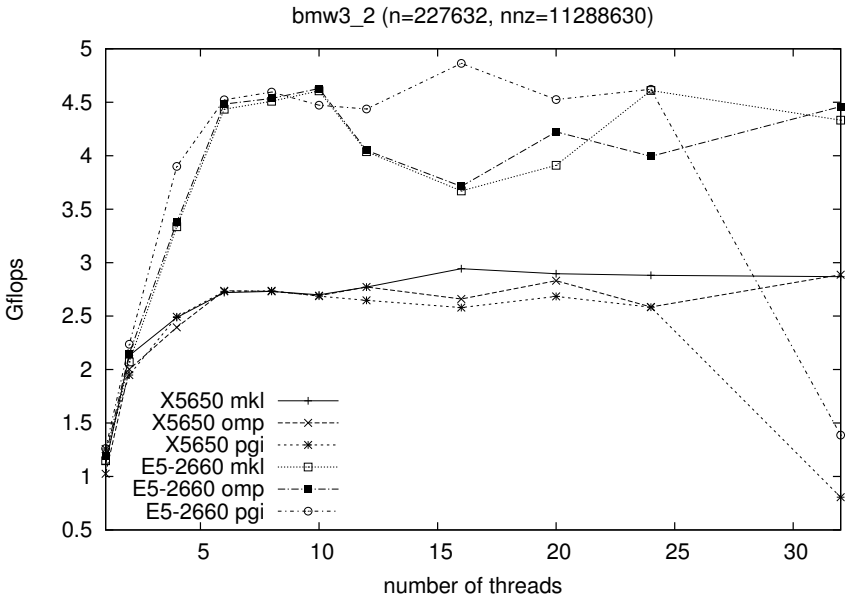

Fig. 11. The performance of SpMV operation for bmw3_2 matrix

- the scalability of the algorithm is related to the distribution scheme and to the matrix properties, especially to its density — sparser the matrix, worse the scalability. Table V shows the division of torso1 and parabolic_fem matrices in the case of 4 tasks. We see, that the densities of the resulting blocks are smaller than the densities of whole matrices (showed in the Table I). Notice, that for parabolic_fem matrix there is a block with no nonzero elements, hence one of the nodes stay idle. The situation is even worse, when we consider 16 tasks division. Looking at Table VI we see, that there are 6 empty blocks for parabolic_fem matrix and the densities of the rest of them are very low.

VII. Conclusion and future work

In this work we investigated the parallelization of SpMV operation, an important and highly-demanding numerical kernel used in many numerical methods. We compared various implementations, namely routine from MKL library, OpenMP parallelized SPARSKIT version compiled using two compilers and two architectures and our own distributed version. The results show, that the most important factor of achieving high performance is hardware architecture together with the distribution pattern and the properties of sparse matrices. It is worth to note, that it is possible to further optimize multinodal implementation (Algorithm 1) by distributing blocks of matrices between nodes taking into account the original matrix density to obtain balanced workload of each MPI task.

Another way to speed up computations is to use GPU cards or MIC architecture accelerators instead of CPUs.

Acknowledgements

This work was prepared using the supercomputer resources provided by the Institute of Mathematics of the Maria Curie-Skłodowska University in Lublin.

References

[1] *Basic Linear Algebra Communication Subprograms*, http://www.netlib.org/blacs/

[2] B. Bylina, J. Bylina, M. Karwacki: *Computational Aspects of GPU-accelerated Sparse Matrix-Vector Multiplication for Solving Markov Models*; Theoretical and Applied Informatics, 23 (2011), no. 2, ISSN 1896-5334, pp. 127–145.

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

[4] G. H. Golub, C. F. van Van Loan: *Matrix Computations*, Johns Hopkins Studies in Mathematical Sciences, 3rd Edition, 2013.

[5] *Intel Math Kernel Library*, http://software.intel.com/en-us/articles/intel-mkl/

[6] *MATLAB. The Language of Technical Computing*, http://www.mathworks.com/products/matlab/

[7] *Matrix Market Exchange Formats*, http://math.nist.gov/MatrixMarket/formats.html

[8] Y. Saad, *Iterative Methods for Sparse Linear Systems: Second Edition*, SIAM, 2003.

[9] Y. Saad, *SPARSKIT: A basic tool kit for sparse computations; Version 2*, June 1994.

[10] *The OpenMP API specification for parallel programming*, http://openmp.org/

[11] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, J. Demmel, *Optimization of sparse matrix-vector multiplication on emerging multicore platforms*, Parallel Computing 35 (2009), pp. 178-194.

[12] Yang, B., Gu, S., Gu, T.-X., Zheng, C. and Liu, X.-P. (2014) Parallel Multicore CSB Format and Its Sparse Matrix Vector Multiplication. *Advances in Linear Algebra & Matrix Theory, 4, 1-8.*

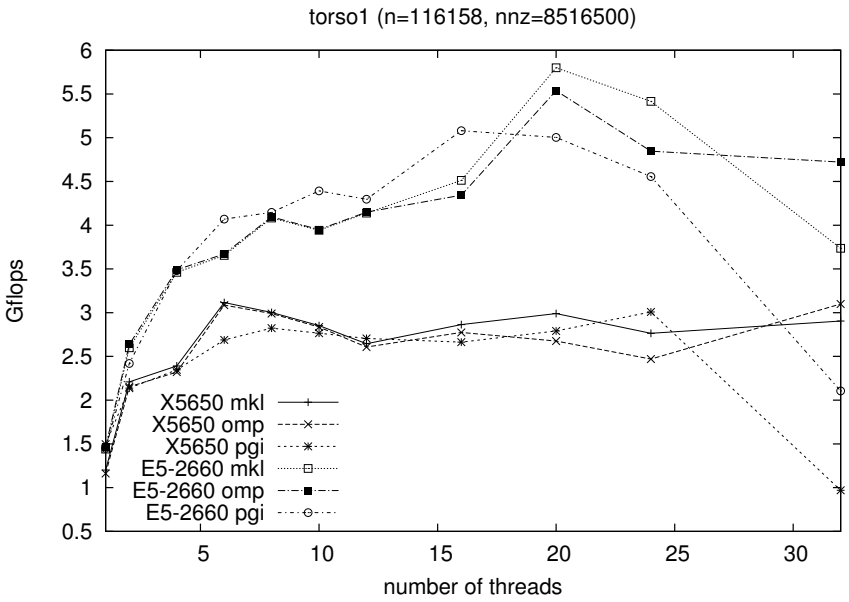

Fig. 12. The performance of SpMV operation for torso1 matrix

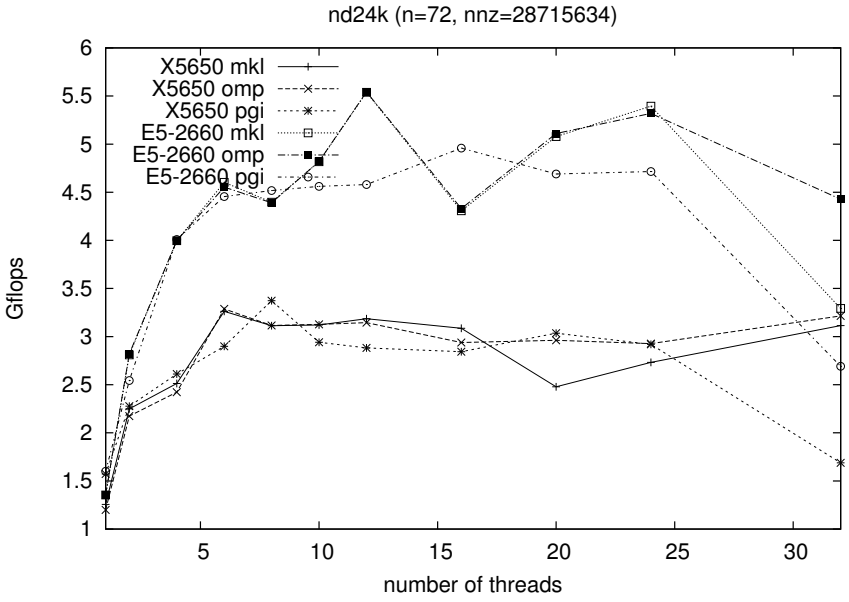

Fig. 13. The performance of SpMV operation for nd24k matrix