Performance Evaluation of Parallel Sparse Matrix–Vector Products on SGI Altix3700

Hisashi Kotakemori[1], Hidehiko Hasegawa[2], Tamito Kajiyama[1], Akira Nukada[1], Reiji Suda[1], and Akira Nishida[1]

[1] CREST, Japan Science and Technology Agency
Graduate School of Information Science and Technology, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033, Japan
{kota, kajiyama, nukada, reiji, nishida}@is.s.u-tokyo.ac.jp
[2] CREST, Japan Science and Technology Agency
Graduate School of Library, Information and Media Studies, University of Tsukuba
Tsukuba 305-8550, Japan
hasegawa@slis.tsukuba.ac.jp

Abstract. The present paper discusses scalable implementations of sparse matrix-vector products, which are crucial for high performance solutions of large-scale linear equations, on a cc-NUMA machine SGI Altix3700. Three storage formats for sparse matrices are evaluated, and scalability is attained by implementations considering the page allocation mechanism of the NUMA machine. Influences of the cache/memory bus architectures on the optimum choice of the storage format are examined, and scalable converters between storage formats shown to facilitate exploitation of storage formats of higher performance.

1 Introduction

Fast solution of linear equations with large sparse coefficient matrices is an essential requirement of advanced computations in science and engineering, and considerable research has been performed on solvers and preconditioners, and high performance implementations have been conducted to that end. We are planning to develop a new library for large-scale sparse matrix solutions that features a wide range of iterative solvers, preconditioners, and storage formats for sequential, shared memory and distributed memory parallel architectures. In the present paper, we discuss the performance of sparse matrix-vector products on a cc-NUMA machine SGI Altix3700.

The matrix-vector product is the most important kernel operation for iterative linear solvers, and its performance has a significant effect on the performance of linear solvers. We will show that satisfactory scalability cannot be attained unless the implementation is aware of the page allocation mechanism of the cc-NUMA machine. In addition, we will show that the storage format of the highest performance may be different for different matrices and on different architectures (CPU and memory), which indicates the importance of the availability of various

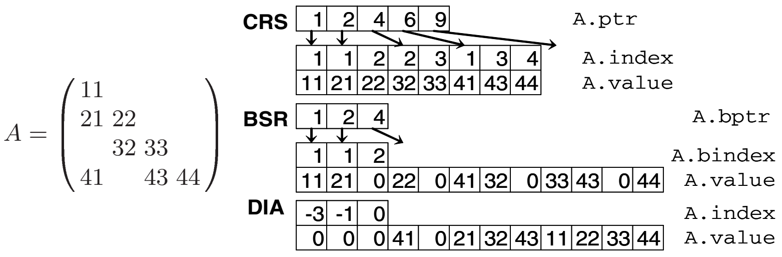

Fig. 1. Data structures of CRS, BSR and DIA. Example matrix A is stored in these storage formats. The arrows in the figures of CRS and BSR designate that the elements of `(b)ptr` are used as indices to `(b)index`. The BSR of the block size is $r = 2$ and $c = 2$.

storage formats in a library. Since modifying their application programs for different storage formats is a burden on users, subroutines for conversions between storage formats are necessary. We will show that scalable parallel implementations of storage conversion routines enable performance enhancements by use of storage formats of higher performances.

2 Storage Formats and Their Implementations

A number of storage formats have been proposed for sparse matrices. They have been proposed for various objectives, such as simplicity, generality, performance, or convenience with respect to a specific algorithm. We implemented seven formats: Compressed Row Storage (CRS), Compressed Column Storage (CCS), Modified Compressed Sparse Row (MSR), Block Sparse Row (BSR), Diagonal (DIA), Ellpack-Itpack generalized diagonal (ELL) and Jagged Diagonal (JDS). In addition to CRS as the baseline format, only BSR and DIA are discussed in the following experiments, because the performance of matrix–vector products in the other formats was lower. In this section, the data structures and the parallel implementations of the matrix–vector products of CRS, BSR and DIA formats are discussed. The structures of the other formats can be found in [1,2,3,4,5]. Figure 1 shows an example matrix A and data structures of CRS, BSR and DIA for A. In the following explanation, mathematically A is assumed to be a square $n \times n$ matrix.

2.1 Compressed Row Storage (CRS)

The CRS format is shown by three arrays (`ptr,index,value`). Let nnz be the number of the non-zero elements in matrix A. The double-precision array `value` of length nnz stores the value of non-zero elements of matrix A as they are traversed row-wise. The integer array `index` of length nnz stores the column indices of the non-zero elements as stored in the array `value`. The integer array `ptr` of length $n + 1$ stores pointers to the beginning of each row in the arrays `value` and `index`.

The following code shows the implementation of the matrix–vector product $y = Ax$ in the CRS format. It is parallelized at the outer loop, and thus (the computations related to) the rows of the matrix are distributed to the threads.

```
#pragma omp parallel for private(i,j,t)
  for(i=0; i<n; i++)  {
    t = 0.0;
    for(j=A.ptr[i];j<A.ptr[i+1];j++)
      t += A.value[j] * x[A.index[j]];
    y[i] = t;
  }
```

2.2 Block Sparse Row (BSR)

For BSR, the matrix is split into $r \times c$ submatrices (called blocks), where r and c are fixed integers. BSR stores the non-zero blocks (submatrices with at least one non-zero element) in a manner similar to CRS. Let $nr = n/r$ and $nnzb$ be the number of non-zero blocks in A. BSR is shown by three arrays (`bptr, bindex, value`). The double precision array `value` of length $nnzb \times r \times c$ stores the elements of the non-zero blocks: the first $r \times c$ elements are of the first non-zero block, and the next $r \times c$ elements are of the second non-zero block, etc. The integer array `bindex` of length $nnzb$ stores the block column indices of the non-zero blocks. The integer array `bptr` of length $nr + 1$ stores pointers to the beginning of each block row in the array `bindex`.

The code of the parallel matrix–vector product for BSR of the 2×2 block (i.e. $r = 2$ and $c = 2$) is shown below. A larger r reduces the number of load instructions for the elements of the vector x, and a larger c works as the unrolling of the inner loop, but this wastes memory and CPU power because of the zero elements in the non-zero blocks.

```
#pragma omp parallel for private(i,j,jj,t0,t1)
  for(i=0; i<nr; i++)  {
    t0 = t1 = 0.0;
    for(j=A.bptr[i];j<A.bptr[i+1];j++)  {
      jj  = A.bindex[j];
      t0 += A.value[j*4+0] * x[jj*2+0] + A.value[j*4+2] * x[jj*2+1];
      t1 += A.value[j*4+1] * x[jj*2+0] + A.value[j*4+3] * x[jj*2+1];
    }
    y[2*i+0] = t0; y[2*i+1] = t1;
  }
```

2.3 Diagonal (DIA)

DIA is shown by two arrays (`index, value`). The double precision array `value` of length $nnd \times n$ stores the non-zero diagonals of the matrix A, where nnd is the number of non-zero diagonals. The integer array `index` of length nnd stores the offsets of each of the diagonals with respect to the main diagonal.

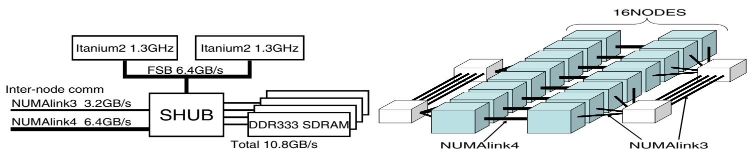

Fig. 2. System configuration of the SGI Altix3700. The left-hand side illustrates the inside of each node, and the right-hand side depicts the interconnections among the 16 nodes.

The code of the parallel matrix–vector product for DIA is shown below. In our implementation the storage scheme is modified so that the matrix elements accessed by each thread is stored in a contiguous region of memory. The inner loop is strip-mined with the number of threads, and interchanged with the outer loop.

```
#pragma omp parallel for private(i)
  for(i=0; i<n; i++)
    y[i] = 0.0;
#pragma omp parallel for private(i,j,k,is,ie,n1,n2,jj,ii)
  for(k=0;k<threads;k++)  {
    n1 = n/threads; n2 = n%threads;
    is = k<n2 ? (n1+1)*k : n1*k+n2;
    ie = k<n2 ? is+n1+1  : is+n1;
    for(j=0;j<nnd;j++)  {
      jj = A.index[j]; ii = _max(is,-jj)-_min(ie,n-jj);
      for(i=_max(is,-jj);i<_min(ie,n-jj);i++)
        y[i] += A.value[nn*k*nnd + j*nn + ii++] * x[jj+i];
    }
  }
```

2.4 The NUMA Architecture and Data Allocation for NUMA

The experiments reported in Section 3 are carried out on a cc-NUMA machine SGI Altix3700 that consists of 16 nodes, and as illustrated in the left-hand side of Fig. 2, each node has one memory controller called SHUB, to which two Itanium2 Madison 1.3-GHz processors with 16-KB L1 cache, 256-KB L2 cache, and 3-MB L3 cache for each are connected with a 6.4-GB/s shared front-side bus and four modules of 512-MB DDR333 SDRAM are connected with 10.8 GB/s. Two nodes are linked by a 6.4-GB/s NUMAlink4 interconnect, and four nodes are connected to one router through a 3.2-GB/s NUMAlink3, as shown in the right-hand side of Fig. 2.

The data is distributed by the first-touch mechanism, i.e. each page is stored in the memory of the node with a processor that accesses the page first. Because data must be transferred via interconnects when it is accessed by a processor out of the node that owns the data, each page should be assigned to the node with the processor that most often accesses the page in order to attain good performance.

To control page allocation, the arrays for matrix storage are initialized with zeros by the same threads as the matrix-vector products.

2.5 Conversion of Storage Format

Routines for the conversions from CRS to the other formats are based on those in the SPARSKIT [3]. Several modifications were necessary to control page allocation. For example, sequential implementation of the conversion from CRS to BSR fills three arrays (bptr, bindex and value) at the same time, but parallel implementation requires bptr to be filled first, because accesses to the arrays bindex and value are distributed referring to bptr as shown in the code in Section 2.2.

3 Experimental Results

The experiments are carried out on a cc-NUMA machine SGI Altix3700. An Intel C/C++ Compiler 8.1 is used with option -O3. In the experiments with 16 or fewer threads, the threads are allocated to different nodes using the `dplace` command, so that the front-side bus is dedicated to a single thread. In the experiments with 32 processors, the bus of each node is shared with the two processors in the node, and the effective memory access performance can be lowered.

Table 1 shows the dimensions, the number of non-zero elements, and the average number of non-zero elements per row for each test matrix used in the experiments. Matrices (a)–(e) are matrices from the Matrix Market[6], and matrix (f) is obtained by finite element discretization of the three–dimensional Poisson equation on a cube.

3.1 Parallel Performance

Table 2 shows the execution time in seconds for 1,000 iterations of matrix–vector products in various storage formats and numbers of threads. Other than CRS as the baseline, the storage formats that give the highest performance, designated in bold face digits, are presented.

First note the dependency of the performance on the matrix. For matrices (a), (c), (d), and (e), the best performance is attained by the same BSR_41. However, the relative performance of BSR_41 is less than twice that of CRS for most cases of (c) and (d), but is more than double that of CRS for (a) and (e). For matrix (b) BSR with another block size is the optimum, and for matrix (f) BSR with yet another block size is the best for eight threads or less and DIA is the best for 16 and 32 threads. These results lead to observations that (1) the performance is improved by optimizing the choice of the matrix storage format, and that (2) the best storage format differs for different matrices and machines (here, the number of processors used), and thus the availability of various storage formats in a library package is important.

Table 1. Test matrices for the experiments. Matrices (a) to (e) are from Matrix Market, and (f) is obtained by finite element discretization of the three–dimensional Poisson equation on a cube. The average number of the non-zero elements per row is shown in the column "Ave.".

Name	Application area	Dimension	Nonzeros	Ave.
(a) af23560	flows over airfoils	23,560	484,256	20.55
(b) fidapm37	finite element modeling	9,152	765,944	83.69
(c) fidap011	finite element modeling	16,614	1,091,362	65.69
(d) bcsstk30	structural engineering	28,924	2,043,492	70.65
(e) s3dkq4m2	cylindrical shell	90,449	4,820,891	53.30
(f) Poisson	Poisson eq. on a cube	1,000,000	26,463,592	26.46

Table 2. Execution times (in seconds) of 1000 iterations of matrix–vector products (performance relative to CRS in parentheses). The block size for BSR is shown as BSR_rc for $r \times c$ blocks. The fastest implementation for each matrix and parallelism are shown in bold.

# of threads		1	2	4	8	16	32
Matrix	Format						
(a)	CRS	3.79 (1.00)	1.89 (1.00)	0.91 (1.00)	0.46 (1.00)	0.24 (1.00)	0.14 (1.00)
	BSR_41	**1.46 (2.59)**	**0.72 (2.64)**	**0.28 (3.22)**	**0.15 (3.04)**	**0.09 (2.64)**	**0.07 (2.07)**
(b)	CRS	2.53 (1.00)	1.33 (1.00)	0.63 (1.00)	0.32 (1.00)	0.18 (1.00)	0.10 (1.00)
	BSR_22	**2.24 (1.13)**	**1.19 (1.12)**	**0.57 (1.11)**	**0.24 (1.34)**	**0.14 (1.26)**	**0.09 (1.18)**
(c)	CRS	3.87 (1.00)	1.98 (1.00)	1.01 (1.00)	0.48 (1.00)	0.26 (1.00)	0.15 (1.00)
	BSR_41	**2.51 (1.54)**	**1.30 (1.52)**	**0.65 (1.55)**	**0.24 (2.04)**	**0.13 (1.91)**	**0.09 (1.63)**
(d)	CRS	6.81 (1.00)	3.53 (1.00)	1.88 (1.00)	0.97 (1.00)	0.46 (1.00)	0.24 (1.00)
	BSR_41	**4.48 (1.52)**	**2.34 (1.51)**	**1.30 (1.44)**	**0.61 (1.60)**	**0.23 (1.96)**	**0.14 (1.75)**
(e)	CRS	20.87 (1.00)	10.47 (1.00)	5.26 (1.00)	2.71 (1.00)	1.43 (1.00)	0.68 (1.00)
	BSR_41	**9.17 (2.28)**	**4.65 (2.25)**	**2.39 (2.20)**	**1.30 (2.08)**	**0.62 (2.29)**	**0.27 (2.49)**
(f)	CRS	149.50 (1.00)	74.96 (1.00)	37.43 (1.00)	18.76 (1.00)	9.51 (1.00)	4.97 (1.00)
	BSR_31	**85.60 (1.75)**	**43.25 (1.73)**	**21.53 (1.74)**	**10.92 (1.72)**	5.63 (1.69)	4.87 (1.02)
	DIA	178.50 (0.84)	89.19 (0.84)	44.34 (0.84)	16.40 (1.14)	**4.72 (2.02)**	**2.81 (1.77)**

The speed-up ratios for the parallel matrix–vector products are shown in Table 3. The parallelization speed-ups are nearly ideal in most cases. Super linear speed-ups (speed-up ratios larger than the number of threads) are sometimes observed, possibly due to the improved cache hit rates because of much smaller data size. The speed-ups for 32 threads are much less than twice those for 16 threads, which may be due to sharing the bus with the two processors on a node. The lower speed-up ratios for 32 threads are most obvious for BSR formats, perhaps because BSR requires a greater number of memory accesses.

DIA outperforms BSR only for matrix (f), which has a regular 27-diagonal structure, and thus is stored very efficiently in the DIA format. However, DIA is still slower than BSR for eight threads or less. For 16 threads, the parallelization speed-up ratio of DIA is more than twice the number of threads, which may be ascribed to the lower memory requirement of DIA, which is approximately half

Table 3. Speed-up ratios for parallel matrix–vector products

# of threads		1	2	4	8	16	32
Matrix	Format						
(a)	CRS	1.00	2.00	4.18	8.19	15.51	27.16
	BSR_41	1.00	2.04	5.19	9.59	15.77	21.69
(b)	CRS	1.00	1.90	3.99	7.93	14.23	24.14
	BSR_22	1.00	1.88	3.91	9.42	15.90	25.18
(c)	CRS	1.00	1.95	3.82	8.03	15.13	26.50
	BSR_41	1.00	1.93	3.83	10.60	18.75	28.03
(d)	CRS	1.00	1.93	3.63	7.00	14.91	28.07
	BSR_41	1.00	1.91	3.45	7.34	19.22	32.21
(e)	CRS	1.00	1.99	3.97	7.70	14.61	30.72
	BSR_41	1.00	1.97	3.83	7.04	14.73	33.63
(f)	CRS	1.00	1.99	3.99	7.97	15.72	30.07
	BSR_31	1.00	1.98	3.97	7.84	15.20	17.58
	DIA	1.00	2.00	4.03	10.88	37.84	63.51

Table 4. Execution times (in seconds) of 1000 iterations of matrix–vector products. Results of another set of experiments for matrix (f) in BSR_31.

# of threads	1	2	4	8	16	32
One thread per node	85.60	43.25	21.53	10.92	5.63	—
Two threads per node	—	73.07	36.58	18.48	9.33	4.87
All data on a single node	85.60	80.88	149.21	224.83	264.22	267.94

Table 5. Conversion times T_{conv} (in milliseconds with the threshold numbers of iterations N_{th} in parentheses) for the same sets of experiments as in Table 2

# of threads		1	2	4	8	16	32
Matrix	Format						
(a)	BSR_41	61.2 (27)	30.7 (27)	15.0 (24)	8.5 (28)	6.7 (45)	10.4 (144)
(b)	BSR_22	96.9 (332)	50.8 (366)	24.9 (410)	12.4 (153)	7.7 (209)	8.5 (531)
(c)	BSR_41	132.8 (98)	68.1 (100)	35.4 (99)	17.8 (73)	11.1 (92)	14.1 (251)
(d)	BSR_41	247.6 (107)	132.3 (112)	69.8 (122)	35.9 (99)	20.2 (90)	22.2 (215)
(e)	BSR_41	575.9 (50)	292.7 (51)	148.5 (52)	78.2 (56)	47.7 (60)	53.5 (132)
(f)	BSR_31	3370.8 (53)	1720.3 (55)	1073.5 (68)	478.6 (61)	303.8 (79)	439.2 (4306)
	DIA	907.4 (-31)	485.6 (-34)	270.3 (-39)	178.0 (76)	165.7 (35)	178.8 (83)

that of BSR. With 32 threads, the parallelization speed-up ratio of BSR drops, which may be ascribed to the higher memory requirements of BSR. These results exemplify the heavy influences of the memory architecture (cache and shared bus) on the optimum choice of the storage format.

Table 4 shows the results of another set of experiments on the matrix (f) in BSR_31. The first line reproduces the results in Table 2. The second line

Table 6. Speed-up ratios for parallel conversion from CRS to target storage format

# of threads		1	2	4	8	16	32
Matrix	Format						
(a)	BSR_41	1.00	1.99	4.07	7.21	9.11	5.90
(b)	BSR_22	1.00	1.91	3.89	7.79	12.54	11.44
(c)	BSR_41	1.00	1.95	3.75	7.47	11.92	9.38
(d)	BSR_41	1.00	1.87	3.54	6.90	12.28	11.15
(e)	BSR_41	1.00	1.97	3.88	7.37	12.08	10.76
(f)	BSR_31	1.00	1.96	3.14	7.04	11.10	7.68
	DIA	1.00	1.87	3.36	5.10	5.48	5.07

gives the execution times with two threads assigned to each node. Here, the influences of the share of the front-side bus by the processors in a node on the performance are observed, and the absolute performances are much lower than in the previous experiments (expect for 32 threads). The speed-ups relative to the performance with two threads are steady up to 32 threads, which confirms that the shared bus is the reason for the lower speed-up ratio for 32 threads. For the third line of Table 4, the data structure of BSR_31 is constructed by a single thread, and thus all of the data are allocated to a single node. The data for the computations on the other node are accessed via the interconnections, and the resulting performances were poor. This confirms the importance of the data distribution discussed in Section 2.4.

3.2 Performance of Storage Format Conversions

Assume that a matrix A is given in the CRS format (e.g. the user prefers the CRS format for some reason) and is to be multiplied to many vectors (e.g. an iterative linear solver is used). If a certain storage format (referred to hereinafter as the target format) is known to attain a higher performance than CRS in matrix–vector products for A, then it may be better to convert into the target format before the matrix–vector multiplications.

Let T_{crs} and T_{tgt} be the execution times of the matrix–vector product in the CRS and target formats, respectively, and let T_{conv} be the execution time of the conversion from the CRS format to the target format. Define the threshold number of iterations N_{th} as $N_{th} = \lceil T_{conv}/(T_{crs} - T_{tgt}) \rceil$. If the number of matrix–vector multiplications is at least N_{th}, then it is better to use the target format; otherwise, it is better to use CRS format without conversion.

Table 5 tabulates the conversion times T_{conv} (with the threshold numbers of iterations N_{th} in parentheses) for the same set of matrices, storage formats, and numbers of threads as Table 2. Note that the number of threads (if it is 16 or less) has little effect on N_{th} in most cases of BSR. This is the case in which the parallelization speed-up ratios for the matrix–vector products in CRS and in BSR and for the conversion from CRS to BSR are similar. The speed-up ratios of the conversion routines are shown in Table 6.

With 32 threads, however, T_{conv} tends to be longer than that with 16 threads, which can be ascribed to the share of the memory bus by the processors in a node. In this case, a strange phenomenon occurs. For matrix (a), execution by 16 threads takes less time than execution by 32 threads if the number of matrix–vector products is from 134 to 185, because the conversion time is shorter with 16 threads than with 32 threads. Accordingly, the threshold number of iterations N_{th} for 32 threads is much larger than those for 16 threads or less. If the code is added using 32 threads, but following the conversion routine for 16 threads, then processors $2n$ and $2n+1$ $(n = 0, ..., 15)$ become the same node. Therefore, efficient execution is possible, because the data locality changes only slightly for 16 threads or 32 threads.

```
   omp_set_num_threads(16);
   #pragma omp parallel
   cpubind(omp_get_thread_num()*2);
   The Storage Format Conversion
   omp_set_num_threads(32);
   #pragma omp parallel
   cpubind(omp_get_thread_num());
The Matrix-Vector Products
```

The conversion times T_{conv} for the DIA format are shorter than those for BSR, perhaps because the amount of data to be stored for DIA is approximately half that for BSR. Negative N_{th} means $T_{dia} > T_{crs}$, thus it is better NOT to convert the matrix into DIA format irrespective of the number of iterations.

4 Related Works

There are a variety of portable software packages that are applicable to the iterative solver of sparse linear systems. SPARSKIT [3] is a toolkit for sparse matrix computations written in Fortran. SPARSKIT provides a number of matrix storage formats, each of which has a pair of converters to and from the CRS format. Together with a rich set of matrix computation subroutines, the toolkit contains several sequential iterative solvers implemented based on reverse communication [7]. PETSc [8] is a C library for the numerical solution of partial differential equations and related problems, and can be used in application programs written in C, C++, and Fortran. The library provides an extensible set of matrix storage formats including various specialized formats that can be directly passed to external libraries. PETSc includes parallel implementations of iterative solvers and preconditioners based on MPI. Aztec [9] is another library of parallel iterative solvers and preconditioners and is written in C. Aztec provides two matrix storage formats. The library is fully parallelized using MPI and can be used in applications written in C and Fortran. From the viewpoint of functionality, our library and all three of the libraries mentioned above support different sets of matrix storage formats, iterative solvers, and preconditioners. Moreover, our library is parallelized using OpenMP and takes the cc-NUMA architecture into consideration.

The performance-enhancing techniques of sparse matrix–vector products are reported in [10,11]. A related issue is the selection of the best storage format for a given matrix and machine. In order to address this problem, E. Im [12] and Demmel *et al.* [13] proposed an automated empirical tuning mechanism for sparse matrix computations that selects an appropriate matrix storage format and solver implementation based on benchmarking data gathered in advance and structural characteristics of non-zero elements in a sparse matrix in hand.

5 Conclusions

In the present paper, we have discussed the parallel performance of matrix-vector product routines, which is crucial for high performance implementation of iterative linear solvers, on a cc-NUMA machine SGI Altix3700. Implementations that take into account the page allocation mechanism have attained satisfactory scalabilities. The memory architecture (specifically, the cache and the memory bus) have been observed to greatly affect the performance of matrix-vector products, and, consequently, storage formats that require more memory are influenced more. The baseline format CRS has scaled well up to 32 threads, and the performance of the BSR format that requires the most memory began to decrease at 32 threads, and the DIA format that requires the least memory has become faster for 16 threads or more. In order to maximize the performance of a machine, users must be able to choose an appropriate storage format for each matrix and each machine, and our scalable implementations of matrix-vector products and storage format conversions in a variety of storage formats enable such selection.

The target machine examined herein (SGI Altix3700) is a cc-NUMA machine. We are planning to port and to evaluate our codes to other shared memory parallel machines, including those having UMA (Uniform Memory Access) and SDSM (Software Distributed Shared Memory) architectures. Parallelization for distributed memory parallel machines through MPI and MPI-OpenMP hybrid parallelization is our next goal. We will also work toward high-performance iterative linear solvers using these kernel routines and effective preconditioners for the solvers, with the goal of developing a complete sparse linear solver library for sequential, shared memory and distributed memory parallel architectures.

Acknowledgements

The authors would like to thank Associate Professor Kengo Nakajima (Tokyo University), Professor Yoshio Oyanagi (Tokyo University) and Dr. Akihiro Fujii (Kogakuin University) who provided valuable advice and discussions regarding the advanced research.

References

1. Barrett, R., et al.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM, Philadelphia (1994)
2. Duff, I., Grimes, R., Lewis, J.: Sparse matrix test problems. ACM Trans. Math. Soft. 15, 1–14 (1989)

3. Saad, Y.: SPARSKIT: a basic took kit for sparse matrix computations, version 2, (June 1994), `http://www.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html`
4. Kincaid, D., Oppe, T., Respess, J., Young, D.: ITPACKV2C User's Guide, Report CNA191. The University of Texas at Austin (1984)
5. Saad, Y.: Krylov subspace methods on supercomputers. SIAM J. Sci. Stat. Comput. 10, 1200–1232 (1989)
6. Matrix Market, `http://math.nist.gov/MatrixMarket`
7. Dongarra, J., Eijkhout, V., Kalhan, A.: Reverse communication interface for linear algebra templates for iterative methods. Technical Report UT-CS-95-291, University of Tennessee (May 1995)
8. Balay, S., Buschelman, K., Eijkhout, V., Gropp, W., Kaushik, D., Knepley, M., McInnes, L., Smith, B., Zhang, H.: PETSc users manual. Technical Report ANL-95/11, Argonne National Laboratory (August 2004)
9. Tuminaro, R.S., Heroux, M., Hutchinson, S.A., Shadid, J.N.: Official Aztec user's guide, version 2.1. Technical Report SAND99-8801J, Sandia National Laboratories (November 1999)
10. Toledo, S.: Improving the memory-system performance of sparse-matrix vector multiplication. IBM Journal of Research and Development 41(6), 711–725 (1997)
11. Pinar, A., Heath, M.T.: Improving Performance of Sparse Matrix-Vector Multiplication. Supercomputing 99 (1999)
12. Im, E.J.: Optimizing the performance of sparse matrix-vector multiplication. Ph.D. thesis, University of California (May 2000)
13. Demmel, J., Dongarra, J., Eijkhout, V., Fuentes, E., Petitet, A., Vuduc, R., Whaley, R.C., Yelick, K.: Self adapting linear algebra algorithms and software. Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Adaptation 93(2), 293–312 (2005)