Intelligent BEE Method for Matrix-vector Multiplication on Parallel Computers

Seiji Fujino
Research Institute for Information Technology, Kyushu University,
Fukuoka, Japan, 812-8581
E-mail: fujino@cc.kyushu-u.ac.jp

Yusuke Onoue
Kyushu DTS Ltd.,
Fukuoka, Japan 812-0011
E-mail: yusuke@zeal.cc.kyushu-u.ac.jp

George Abe
Graduate School of Information Science and Electrical Engineering,
Kyushu University,
Fukuoka, Japan 812-8581
E-mail: g.abe@zeal.cc.kyushu-u.ac.jp

***Abstract*—This paper compares the performance of sparse Matrix-vector multiplication paralleled by the conventional Block-Cyclic distribution and its improved variant on parallel computer with shared memory. The underlying idea is to exchange nonzero entries of matrix assigned to each thread with block unit. Numerical results demonstrate that the proposed distribution using exchange nonzero entries of matrix with block unit gives or improves parallelism.**

***Keywords*–Block exchanging; Matrix-vector multiplication; iBEE method; Parallel computers**

I. INTRODUCTION

We consider the problem of efficient Matrix-vector multiplication on parallel computers. As you know well, Matrix-vector multiplication appears often in solution of linear system of equations, and its efficient computation is crucial. In particular, Matrix-vector multiplication has a large part of computation of solving linear system of equations on parallel computers [10][11]. Many studies on fast computation of Matrix-vector multiplication have been proposed [3][6][7][12][13]. Fast computation depends greatly on evenly distributed nonzero entries of matrix onto each thread or process. In general, Block Cyclic (BC) distribution [8] is to be effective approach in order to evenly distribute nonzero entries of matrix. However, in the BC distribution, it is well known that parallel performance changes greatly as treated number of blocks changes. Therefore, in the BC distribution, it is not easy to decide optimum number of blocks.

In this paper, we propose an intelligent approach which distributes evenly nonzero entries of matrix to each thread by means of blocking exchange. We refer to intelligent Blocking Exchange for Evenly distributed nonzero entries of matrix (iBEE) method. We adopt double strategies for the iBEE method based on the conventional BC distribution. The first strategy is to decide the number of blocks so as to be evenly distributed for nonzero entries of matrix on each thread. The second strategy is to adopt a blocking exchange technique for refined and sufficiently even distribution. As a result, the iBEE method makes nonzero entries able to be significantly evenly distributed on each thread with the BC distribution.

The paper is organized as follows. In Section 2, we introduce a brief outline of the conventional Block and BC distributions. In Section 3, we describe our proposed iBEE method in detail. The iBEE method includes double strategies for the purpose of fast computation of the Matrix-vector multiplication on parallel computers. Moreover, in Section 4, we evaluate effectiveness of the iBEE method through numerical experiments. Finally, in Section 5, we will make concluding remarks.

II. THE CONVENTIONAL DISTRIBUTION METHODS

We assume that nonzero entries are stored in Compressed Row Storage (CRS) format [2] and the pseudo program of computation of Matrix-vector multiplication is written in Fortran 90 [1]. Here, matrix A is sparse. In this case, concerning the conventional method for nonzero entries, the Block and BC distributions exist as simple distribution. Below, we give an outline of the Block and BC distributions. "$ncol$" means dimension of matrix, and "$rowptr$", "$colind$" and "val" mean arrays for starting pointer of each row, column index of each element and value of nonzero entries, respectively.

Pseudo program 1: Matrix-vector multiplication [2]

```
Do i = 1, ncol
   temp = 0.0
   Do j = rowptr(i), rowptr(i+1) - 1
      temp = temp + val(j) * x(colind(j))
   End Do
   y(i) = temp
End Do
```

A. Block distribution

In block distribution, we divide nonzero entries into blocks with the same number of threads. Moreover, we divide also nonzero entries such that the number of matrix row in each block is same each other.

In Fig.1 we show an example of two block distribution for matrix with dimension of 8 and with nonzero entries of 19. In this case, the difference of number of nonzero entries included in each block is three. Here, we set the thread number as "nth" and the number of blocks as "$nblk$". In block distribution, we get that $nblk = nth$.

 ISBN: 978-1-61208-339-1

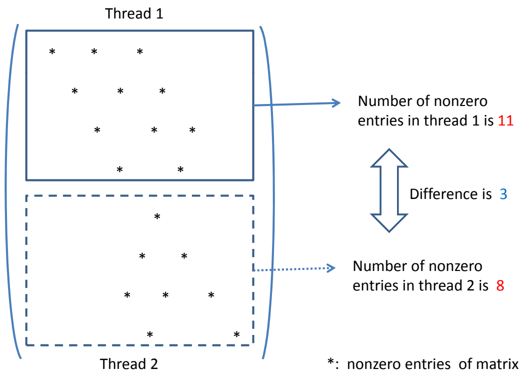


Fig.1 An example of block distribution in case of two threads.

We exhibit pseudo program for production of array of bst which stores the first row in each block.

Pseudo program 2: Production of array of bst [2]

```
bst(1) = 1
tmp1 = ncol/nblk
tmp2 = mod(ncol, nblk)
Do i = 1, tmp2
   bst(i + 1) = bst(i) + tmp1
End Do
Do i = tmp2 + 1, nblk
   bst(i + 1) = bst(i) + tmp1 + 1
End Do
```

B. Block Cyclic distribution

We store block-id which is assigned to a thread to an array of asb (=assigned block). That is, we store block-id of the jth of block, which is assigned to the ith thread, to array of $asb(i, j)$. We produce an array of asb in the BC distribution as below.

Pseudo program 3: Production of array of asb in the BC distribution [2]

```
Do i = 1, nth
   Do j = 1, nblk/nth
      asb(i, j) = i + nth * (j - 1)
   End Do
End Do
```

Below, we present a parallel version of Matrix-vector multiplication with the OpenMP library [4][9] in the BC distribution. The array of bst stores row number on the starting row of each block. "$ncol$" means dimension of matrix, and "$rowptr$", "$colind$" and "val" mean arrays for starting pointer of each row, column index of each element and value of nonzero entries, respectively. "omp parallel do" means a directive for thread parallelism with the OpenMP library.

Pseudo program 4: Parallel version of Matrix-vector multiplication with OpenMP.

```
!$omp parallel do private(i, j, k, l, tmp, temp)
Do i = 1, nth
   Do j = 1, nblk/nth
      tmp = asb(i, j)
      Do k = bst(tmp), bst(tmp + 1) - 1
         temp = 0.0
         Do l = rowptr(k), rowptr(k + 1) - 1
            temp = temp + val(l) * x(colind(l))
         End Do
         y(k) = temp
      End Do
   End Do
End Do
```

We present an example of the BC distribution in case of two threads in Fig.2. The number of nonzero entries in thread 1 is nine, and the number of nonzero entries in thread 2 is ten. In this case, the difference of number of nonzero entries included in each block is only one.

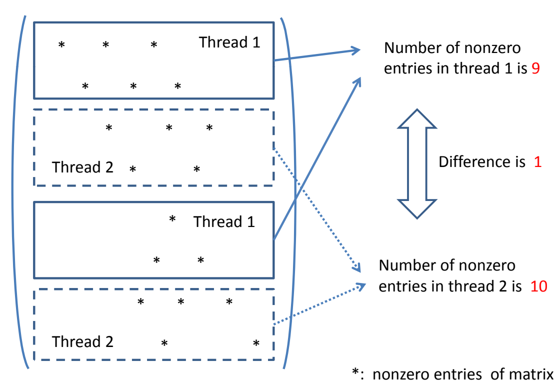


Fig.2 An example of the BC distribution in case of two threads.

III. INTELLIGENT BEE METHOD

In this section, we propose the iBEE method. The iBEE means intelligent blocking exchange technique for evenly distributed nonzero entries of matrix. The iBEE method is constructed based on the BC distribution, and adopts the following two intelligent strategies.

1) To determine the number of blocks automatically.
2) To apportion nonzero entries evenly with block exchanging technique.

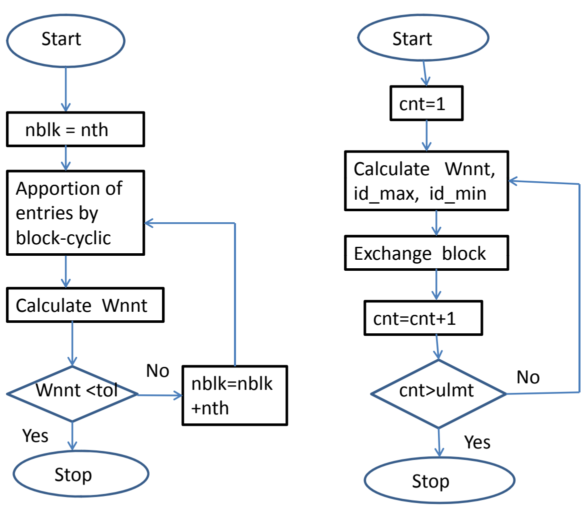
A. To determine automatically the number of blocks

In order to determine the number of blocks automatically, we introduce indicator $Wnnt$ (Width of nnt). $Wnnt$ is defined as follows:

$$\text{Wnnt} := \max_{i \text{ in thread}}(nnt(i)) - \min(nnt(i)) \quad (1 \leq i \leq nth). \tag{1}$$

Here, "nth" means the thread number and "nnt" means the number of nonzero entries per thread.

Fig.3 (a) shows an algorithm to automatically compute the number of blocks per thread. In Fig.3, "$nblk$" means the number of blocks. At first, $nblk$ is initialized to nth. Next, we calculate indicator $Wnnt$, and check if $Wnnt < tolerance$ or not. If $Wnnt < tolerance$ then $nblk$ is determined to nth. On the other hand, if $Wnnt \geq tolerance$ then increase $nblk$ by nth. Until $Wnnt < tolerance$, $nblk$ is increased by nth.



(a)Determination of number of blocks (b)Exchanging blocks

Fig.3 Algorithm to automatically compute the number of blocks per thread.

B. To apportion evenly nonzero entries with block exchanging technique

Fig.3 (b) shows the algorithm of exchanging blocks. In Fig.3 (b), "u_lmt" means upper limit of the number of blocks exchanging. "id_max" means thread ID of the thread most apportioned nonzero entries and "id_min" means thread ID of the thread least apportioned nonzero entries. In Fig.3 (b), at first, cnt is initialized to one. Next, we calculate $Wnnt$ and id_max, id_min. Furthermore, the block apportioned to id_max and the block apportioned to id_min are exchanged. We increase cnt by one, and if $cnt > u\_lmt$ then the block exchange is finished.

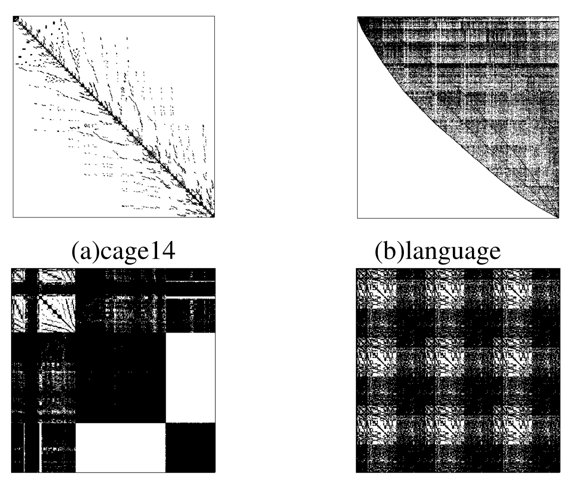
IV. Numerical Experiments

In this section we discuss numerical experiments of the BC distribution and the iBEE method. All computations were carried out in double precision floating-point arithmetic on FUJITSU PRIMEQUEST 580 (clock: 1.6GHz). FUJITSU optimized Intel Fortran Compiler90 and compile option "-Kfast, OMP" were used. We implemented all programs with the OpenMP library. The thread numbers are 1, 2, 4, 8, 16, 24, 32, 48 and 64. We set parameters of the iBEE method as $tolerance = 10000$ and u_lmt is the same as the thread number. Four test matrices are taken from Florida Sparse Matrix Collection [5]. The description of test matrices is shown in Table I. In this Table, "nnz" means number of nonzero entries, and "ave_nnz" means number of nonzero entries per single row. Moreover, "ave_$nnz8$" means average number of total nonzero entries per eight threads.

TABLE I. The description of test matrices.

matrix	dimension	nnz	ave_nnz	ave_$nnz8$	analytic field
cage14	1,505,785	27,130,349	18.02	3,391,294	DNA electrophoresis
language	399,130	1,216,334	3.04	152,041	language processing
poisson3Db	85,623	2,374,949	27.74	296,869	structural
sme3Dc	42,930	3,148,656	73.34	393,582	

Fig.4 shows the structure of four matrices. That is, Fig.4 plots nonzero entries of matrices. From Fig.4, we can see that a lower row decreases the number of nonzero entries of matrix language. Therefore, it is difficult to apportion sufficiently nonzero entries of matrix language evenly when we adopt the BC distribution. It is also difficult to apportion nonzero entries of matrix cage14 because the number of nonzero entries of matrix cage14 is very large.



(a)cage14 (b)language (c)poisson3Db (d)sme3Dc

Fig.4 Pattern of nonzero entries of four matrices.

We present differences between minimum and maximum of nonzero entries when the thread numbers are 8 and 64 in

TABLE II. DIFFERENCE BETWEEN MINIMUM AND MAXIMUM OF NONZERO ENTRIES WHEN THE THREAD NUMBERS ARE 8 AND 64.

number of threads	matrix	ave. nnz per thread	(a)diff. of BC	(b)diff. of iBEE	ratio (=(b)/(a))
8	cage14	3,391,294	303,593 (8.95%)	7,028 (0.21%)	1/42.6
	language	152,041	35,626 (23.4%)	8,121 (5.34%)	1/4.38
	poisson3Db	296,869	19,088 (6.43%)	1,511 (0.51%)	1/12.6
	sme3Dc	393,582	89,719 (22.8%)	7,923 (2.01%)	1/11.3
64	cage14	423,911	173,813 (41.0%)	8,942 (2.11%)	1/19.4
	language	19,005	13,099 (68.9%)	9,362 (49.3%)	1/1.40
	poisson3Db	37,109	13,636 (36.7%)	9,654 (26.0%)	1/1.41
	sme3Dc	49,198	61,864 (125.7%)	6,638 (13.5%)	1/9.32

Table II. We see that the difference of the iBEE method is much smaller than that of the BC distribution at both 8 and 64 processors.

A. Computational cost of the iBEE method

We define D' as a difference between minimum and maximum of nonzero entries after exchanging blocks. We denote m as exchanging block-id included in minimum nonzero entries and M as that included in maximum nonzero entries, respectively. Moreover, we denote $bnz(m)$ and $bnz(M)$ as number of nonzero entries with block-id of m and that of nonzero entries with block-id of M, respectively. Then, the difference D' is written as

$$D' = |W_{\rm nnt} - 2(bnz(M) - bnz(m))|. \quad (2)$$

That is, first we calculate the above difference of D' to all combinations of block exchanging, secondly we may exchange blocks so as to be minimum nonzero entires for difference D'. For example, when number of blocks included in each thread is 100, number of combination of block exchanging is estimated as only $100 \times 100 = 10,000$. Then, we may calculate the above difference D' at 10000 times. Therefore, we can do it quickly. As a result, we can estimate that the cost of iBEE method is not expensive at all, because there is no reordering of nonzero entries of matrix.

We ran two experiments. The first experiment is performance estimation of parallel Matrix-vector multiplication with the BC distribution and the iBEE method.

B. Performance estimation of parallel Matrix-vector product with block-cyclic distribution and the iBEE method

In this section, we show numerical results of parallel Matrix-vector product with the BC distribution and the iBEE method. We tested Matrix-vector multiplication at 1000 times. Table III shows the performance of the iBEE method. In Table III, "nth" means the thread number and "$nblk$" means the number of blocks. Bold figures means minimum total time among the BC distribution and the iBEE method.

From Table III, we can see that $Wnnt$ of the iBEE method is much smaller than that of the BC distribution, and time of the iBEE method is shorter than that of the BC distribution for all thread number. In particular, the iBEE method works well when the thread number becomes larger than 32 threads. Moreover, it is concluded that the iBEE method is very effective for matrix sme3Dc.

TABLE III. COMPARISON OF PERFORMANCE OF PARALLELED MATRIX-VECTOR MULTIPLICATION WITH THE BC DISTRIBUTION AND THE iBEE METHOD.

(a)matrix: cage14

method	nth	$nblk$	$Wnnt$	ratio(%)	time[s] $A\boldsymbol{v}$-t	total-t	speed-up
BC	1	1	0	-	184.598	184.598	1.0
iBEE							
BC	2	10	1,233,727	100.0	101.987	101.987	1.81
iBEE			6,633	0.54	101.423	**101.423**	1.82
BC	4	28	745,785	100.0	71.941	71.941	2.56
iBEE			7,435	1.00	71.606	**71.606**	2.57
BC	8	104	303,593	100.0	38.499	38.499	4.79
iBEE			7,028	2.31	37.522	**37.522**	4.92
BC	16	96	438,366	100.0	26.635	26.635	6.93
iBEE			9,191	2.10	25.127	**25.127**	7.34
BC	24	120	321,120	100.0	18.538	18.538	9.95
iBEE			9,748	3.04	17.974	**17.974**	10.27
BC	32	224	251,881	100.0	11.688	11.688	15.79
iBEE			8,272	3.28	10.993	**10.993**	16.79
BC	48	288	193,305	100.0	3.863	3.863	47.78
iBEE			6,300	3.26	3.212	**3.212**	57.47
BC	64	256	173,813	100.0	2.340	2.340	78.88
iBEE			8,942	5.14	2.138	**2.138**	86.34

(b)matrix: language

method	nth	$nblk$	$Wnnt$	ratio(%)	time[s] $A\boldsymbol{v}$-t	total-t	speed-up
BC	1	1	0	-	20.382	20.382	1.0
iBEE							
BC	2	10	112,544	100.0	8.964	8.964	2.27
iBEE			2,078	1.85	8.793	**8.793**	2.31
BC	4	28	54,286	100.0	4.034	4.034	5.05
iBEE			4,147	7.64	4.008	**4.008**	5.08
BC	8	280	35,626	100.0	2.007	2.007	10.15
iBEE			8,121	22.80	1.964	**1.965**	10.37
BC	16	368	28,770	100.0	1.046	1.046	19.48
iBEE			9,623	33.45	1.012	**1.013**	20.12
BC	24	600	27,922	100.0	0.744	0.744	27.39
iBEE			9,453	33.86	0.702	**0.703**	28.99
BC	32	800	21,982	100.0	0.602	0.602	33.85
iBEE			9,802	44.59	0.554	**0.556**	36.65
BC	48	1008	15,896	100.0	0.449	0.449	45.39
iBEE			9,916	62.38	0.421	**0.422**	48.29
BC	64	1152	13,099	100.0	0.391	0.391	52.12
iBEE			9,362	71.47	0.354	**0.355**	57.41

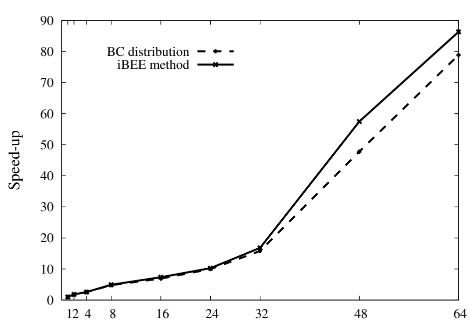
(c)matrix: poisson3Db

method	nth	$nblk$	$Wnnt$	ratio(%)	time[s] $\boldsymbol{Av}$-t	total-t	speed-up
BC	1	1	0	-	14.921	14.921	1.0
iBEE							
BC	2	14	10,581	100.0	6.853	6.853	2.17
iBEE			665	6.28	6.707	**6.707**	2.22
BC	4	64	21,988	100.0	3.036	3.036	4.91
iBEE			1,607	7.31	3.015	**3.015**	4.94
BC	8	128	19,088	100.0	1.527	1.527	9.77
iBEE			1,511	7.92	1.517	**1.517**	9.83
BC	16	144	33,619	100.0	0.833	0.833	17.91
iBEE			7,705	22.92	0.789	**0.787**	18.95
BC	24	216	21,758	100.0	0.573	0.573	26.04
iBEE			5,570	25.56	0.545	**0.542**	27.53
BC	32	256	11,863	100.0	0.434	0.435	34.30
iBEE			9,004	75.90	0.424	**0.424**	35.19
BC	48	384	8,781	100.0	0.318	0.318	46.92
iBEE			6,136	69.88	0.313	**0.313**	47.67
BC	64	384	13,636	100.0	0.296	0.296	50.40
iBEE			9,654	70.80	0.273	**0.273**	54.65

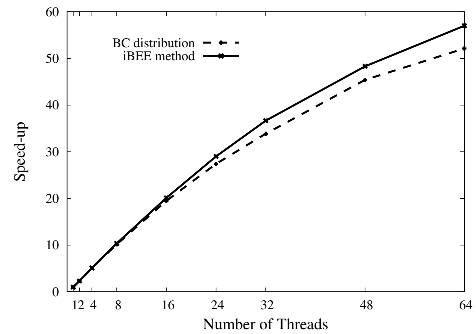
(d)matrix: sme3Dc

method	nth	$nblk$	$Wnnt$	ratio(%)	time[s] $\boldsymbol{Av}$-t	total-t	speed-up
BC	1	1	0	-	13.335	13.335	1.0
iBEE							
BC	2	8	76,358	100.0	6.719	6.719	1.98
iBEE			6,330	8.29	6.702	**6.702**	1.99
BC	4	20	80,777	100.0	2.680	2.680	4.97
iBEE			4,839	5.99	2.606	**2.606**	5.11
BC	8	40	89,719	100.0	1.344	1.344	9.92
iBEE			7,923	8.83	1.291	**1.291**	10.32
BC	16	112	41,814	100.0	0.720	0.720	18.52
iBEE			3,772	9.02	0.654	**0.654**	20.39
BC	24	168	28,798	100.0	0.496	0.496	26.88
iBEE			3,080	10.70	0.449	**0.449**	29.69
BC	32	224	24,769	100.0	0.386	0.386	34.54
iBEE			1,873	7.56	0.348	**0.348**	38.31
BC	48	336	16,990	100.0	0.292	0.292	45.66
iBEE			1,505	8.86	0.263	**0.263**	50.70
BC	64	384	61,864	100.0	0.334	0.334	39.92
iBEE			6,638	10.73	0.240	**0.240**	55.56

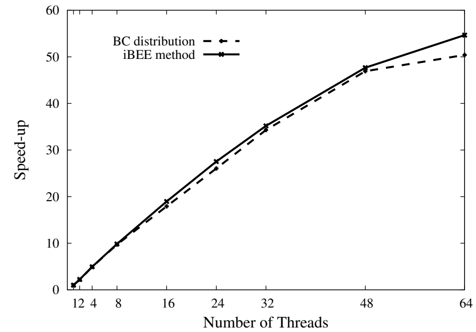
Fig.5 (a)-(d) plots the speed-up of parallel Matrix-vector multiplication with the BC distribution and the iBEE method. In Fig.5 (a)-(d), dashed line plots speed-up of the BC distribution and solid line plots that of the iBEE method. From Fig.5 (a)-(d), speed-up of the iBEE method is larger than that of the BC distribution.



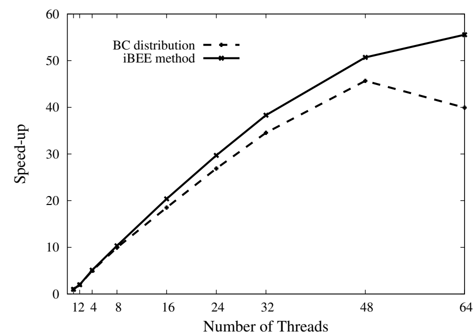
(a)matrix: cage14



(b)matrix: language



(c)matrix: poisson3Db



(d)matrix: sme3Dc

Fig.5 Comparison of speed-up of paralleled Matrix-vector multiplication with the BC distribution and the iBEE method.

In Fig.6, we show the tendency of ratio (%) of $Wnnt$ when the thread number changes. We see that ratios of $Wnnt$ are very low for four matrices compared with the BC distribution.

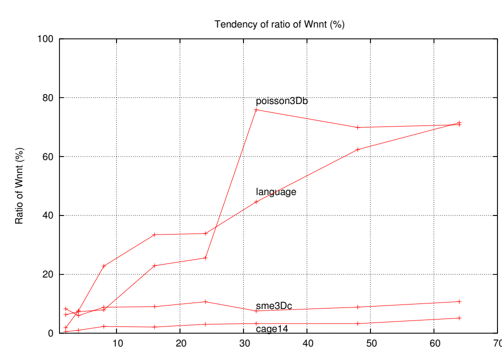


Fig.6 Tendency of ratio (%) of $Wnnt$ when the thread number changes.

V. CONCLUDING REMARKS

We proposed an intelligent Blocking exchange technique of Evenly distributed for nonzero Entries of matrix. Moreover, we evaluated the performance of parallel Matrix-vector product using the iBEE method. As a result, it turned out that the iBEE method can distribute nonzero entries of matrix more evenly than the conventional BC distribution. Moreover, parallel performance of Matrix-vector multiplication with the iBEE method is faster than that with the BC distribution.

REFERENCES

[1] C. Arai, Introduction of Fortran90, Morikita Corp., 1998.

[2] B. Barrett, et al., Templates, SIAM, Philadelphia, 2000.

[3] R. H. Bisseling and W. Meesen, "Communication balancing in parallel sparse matrix-vector multiplication", Electronic Trans. on Numerical Analysis, vol.21, pp.47–65, 2005.

[4] R. Chandra, et al., Parallel programming in OpenMP, Morgan Kaufmann Publishers, 2001.

[5] T. Davis, Univ. of Florida sparse matrix collection, http://www.cise.ufl.edu/research/sparse/matrices/index.html [accessed: 2014-04-08].

[6] E. J. Im, K. Yelick and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels", Int. J. of High Performance Computing Applications, vol.18, no.1, pp.135–158, 2004.

[7] S. Lee and R. Eigenmann, "Adaptive Runtime Tuning of Parallel Sparse Matrix-Vector Multiplication on Distributed Memory Systems", The Proc. of the 22nd Annual Int. Conference on Supercomputing, June 2008, pp.195–204.

[8] J. J. Lo, et al., "Tuning Compiler Optimizations for Simultaneous Multithreading", Int. J. of Parallel Programing, vol.27, no.6, pp.114–124, 1999.

[9] T. G. Mattson, B. A. Sanders and B. L. Massinggill, Pattern for Parallel programing, Addison Wesley, 2005.

[10] Y. Saad, Iterative methods for sparse linear systems 2nd edition, SIAM, Philadelphia, 2003.

[11] H. A. van der Vorst, Iterative Krylov methods for large linear systems, Cambridge University Press, Cambridge, 2003.

[12] S. Williams, et al., "Optimization of sparse matrix-vector multiplication on emerging multicolor platforms", Parallel Computing, vol.35, pp.178–194, March 2009.

[13] A. N. Yzelman and R. H. Bisseling, "Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods", SIAM, J. on Scientific Computing, vol.31, no.4, pp.3128–3154, 2009.

 ISBN: 978-1-61208-339-1