Performance of Automatically Tuned Parallel GMRES(m) Method on Distributed Memory Machines

Hisayasu KURODA[1] *, Takahiro KATAGIRI[12], and Yasumasa KANADA[3]

[1] Department of Information Science, Graduate School of Science, The University of Tokyo
[2] Research Fellow of the Japan Society for the Promotion of Science
[3] Computer Centre Division, Information Technology Center, The University of Tokyo
2-11-16 Yayoi, Bunkyo-ku, Tokyo 113-8658, JAPAN
Phone: +81-3-5841-2736, FAX: +81-3-3814-2731
{kuroda, katagiri, kanada}@pi.cc.u-tokyo.ac.jp

Abstract. As far as the presently available public parallel libraries are concerned, users have to set parameters, such as a selection of algorithms, an unrolling level, and a method of communication. These parameters not only depend on execution environment or hardware architecture but also on a characteristic of the problems. Our primary goal is to solve two opposite requirements of reducing users parameters and getting high performance. To attain this goal, an auto-tuning mechanism which automatically selects the optimal code is essential. We developed a software library which uses GMRES(m) method on distributed memory machines, the HITACHI SR2201 and HITACHI SR8000. The GMRES(m) method is one of the iterative methods to solve large linear systems of equations. This software library can automatically adjust some parameters and selects the optimal method to find the fastest solution. We show the performance of this software library and we report a case where our library is approximately four times as fast as the PETSc library which is widely used as a parallel linear equation solver.

1 Introduction

Linear algebra, in particular the solution of linear systems of equations and eigenvalue problems, is the basic of general calculations in scientific computing. When a coefficient matrix of linear systems of equations is large and sparse, iterative methods are generally used. For example, if a coefficient matrix is real symmetric and positive definite, the Conjugate Gradient method (CG) is often used. In the case of a nonsymmetric matrix, there are a number of iterative methods with lots of variations [1, 2]. Therefore, in the nonsymmetric case, the most efficient method is left for to further discussion. In addition, there are a few parallel implementations of the iterative methods in the nonsymmetric case. In this

* *Candidate to the Best Student Paper Award*

paper, we focussed on the GMRES(m) which is improved GMRES(Generalized Minimal RESidual method), and developed its library on distributed memory machines.

The GMRES is one of the Krylov subspace solution methods and finds a suitable approximation for the solution x of $Ax = b$ by using the minimum residual approach at every iteration step.

The GMRES(m) restarts the GMRES after each m steps, where m is a suitably chosen integer value. The original method without restarting is often called full-GMRES. This restarting reduces both calculation counts and the size of memory allocation.

This paper is organized as follows. Description of the algorithm of our version of GMRES(m) in Section 2. Section 3 is about the parameters for auto-tuning, and how to search for the optimal parameters. In Section 4, we show auto-tuned parameters and execution time of our library using the auto-tuning methodology. Finally, Section 5 gives conclusions for this paper.

2 The GMRES(m) Algorithm

When given an $n \times n$ real matrix A and a real n-vector b, the problem considered is: Find x which belongs to $\mathbb{R}^n$ such that

$$Ax = b \ . \tag{1}$$

Equation(1) is a linear system. A is the coefficient matrix, b is the right-hand side vector, and x is the vector of unknowns.

Figure 1 shows our version of preconditioned GMRES(m) algorithm. Note that K in the Figure 1 is a preconditioning matrix and this algorithm uses right-preconditioning because of the advantage that it only affects the operator and does not affect the right-hand side.

2.1 Parallel implementation of GMRES(m)

The coefficient matrix A, the vector of unknowns x, the right-hand side vector b, the temporary vectors v_i ($m + 1$ vectors), and orthogonalized vector w are distributed by row-block distribution and each processor element(PE) except for the last PE has the same number of rows. On the other hand, the matrix H, the vector s, and the vector c are the same on every PE.

Since the vector x and v_i are needed to be gathered on every PE, a temporary vector of size n, where n is the size of matrix A, is required.

In the parallel implementation, lines 2, 7, and 29 in the Figure 1 which include matrix-by-vector products and line 9 in the Figure 1 which includes dot product require interprocessor communications.

In lines 14–22 in the Figure 1 which include QR decomposition by using Givens rotations, every PE updates the matrix H which holds the same data. It

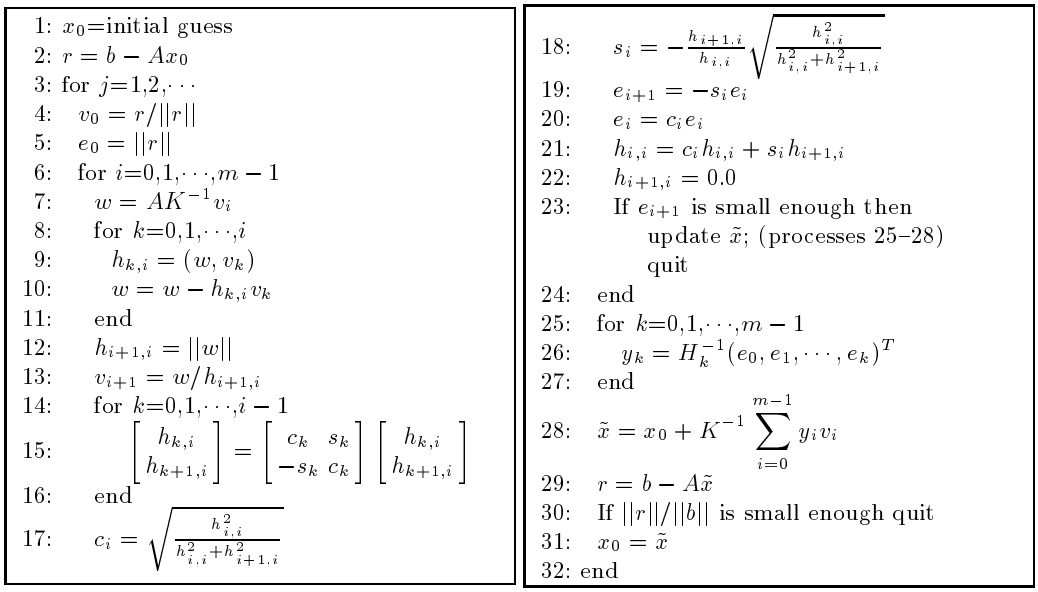


Fig. 1. The preconditioned GMRES(m) algorithm

r, v_i, and w are vectors and if $i \neq j$ then v_i and v_j are different vectors, and not elements of the same vector v. m is the restarting frequency.

seems that holding H is inefficient. However m is several hundreds at the most and the decomposition of the matrix H whose size is $(m+1) \times m$ is inefficient to parallelize. Therefore, the overhead time that every PE updates at the same time is very small.

3 Method for searching parameters

We have developed automatically tuned parallel library by using CG [3]. In this section description of several tuning factors to be considered in preconditioned GMRES(m) algorithm and methods of tuning parameters automatically are provided.

Our library automatically sets several parameters to get high performance. This action is executed after being given a problem. Therefore, our library can select the best method according to a characteristic of the problem. Our library provides a lot of source codes. Users only have to compile them once. While our library code is being executed, the optimal code is selected one after another automatically.

3.1 Matrix storage formats

In the sparse matrix formats, we store the nonzero elements by rows, along with an array of corresponding column numbers and an array of pointers to the beginning of each row (see Figure 2). It is called as *compressed row storage format*.

$$A = \begin{bmatrix} a & b & 0 & 0 \\ c & d & e & 0 \\ 0 & f & g & h \\ 0 & 0 & i & j \end{bmatrix}$$

```
rp[5]={0,2,5,8,10}; /* pointers to the beginning of each row */
cval[10]={0,1,0,1,2,1,2,3,2,3};      /* indices */
aval[10]={a, b, c, d, e, f, g, h, i, j}; /* elements */
```

Fig. 2. Compressed row storage format

In case that the number of nonzero elements at each row are almost equal, compressed row storage format was converted to a matrix format whose size of each row is fixed (see Figure 3). This is called *compressed row storage format for unrolling*. Using such a matrix format, we expected to save the execution time because of the effect of unrolling.

```
cval[12]={0,1,0,0,1,2,1,2,3,2,3,0};      /* indices */
aval[12]={a, b, 0, c, d, e, f, g, h, i, j, 0}; /* elements */
nsize[4]={2,3,3,2};      /* the number of elements of each row*/
csize=3;                 /* fixed size */
```

Fig. 3. Compressed row storage format for unrolling

Before executing the main iteration, the actual time of the matrix-by-vector product was measured. With this information, we can select the best matrix storage format.

3.2 The stride size of loop unrolling for matrix-by-vector products

To perform the matrix-by-vector product at high performance, we should select the best size of the stride for loop unrolling. This depends on the machine architectures and optimization level of the compilers.

Our library prepares a large number of loop unrolling codes. For example, if the number of nonzero elements of a matrix A at each row is smaller than

10, all expanded loop unrolling codes are examined. In addition to unrolling the inner-loop, we unroll also the outer loop with strides 1, 2, 3, 4, and 8.

There are two types of codes, i.e., a non-prefetch code and a prefetch code. The non-prefetch code uses indirect access just as it is or entrusts the compiler with the treatment. The prefetch code takes off indirect access to the element of arrays (see Figure 4).

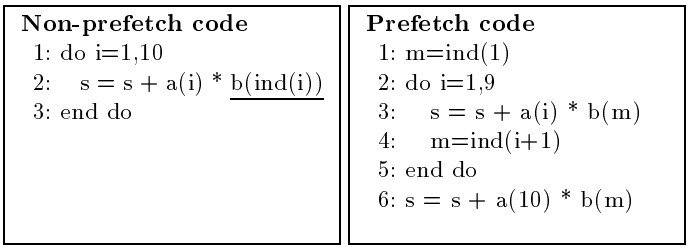


Fig. 4. Non-prefetch code and prefetch code

As for the prefetch codes, we unroll the outer loop so that the sizes of the stride can be 1, 2, 3, and 4. In total, there are 9 ways of the unrolling codes in each of the number of nonzero elements at each row.

Same as in the case of the matrix format, actual time of the matrix-by-vector product was measured in order to select the best unrolling code in the above.

3.3 How to comunicate in matrix-by-vector products

In matrix-by-vector products, we need a gather operation. Because the elements of a vector are distributed on all of the PEs. We can select the following five implementations for the communications.

No dependence on the location of the nonzero elements of matrix A:

1. Use MPI_Allgather function from the MPI library.
2. Gather in 1 PE then broadcast with MPI_Bcast function.

Dependence on the location of the nonzero elements of matrix A:

3. First use MPI_Isend function, next use MPI_Irecv function.
4. First use MPI_Irecv function, next use MPI_Isend function.
5. Use MPI_Send and MPI_Recv functions.

A communication table is used in the method from 3 to 5. This table indicates the relation between a element index of a vector and a PE number which requires the indexed value. This relation is created from nonzero element indices of a matrix A. By using this communication table, communication traffic becomes very small because an element is transmitted to a PE which requires it. We communicate all of the elements from minimal index to maximal index to other

PEs so that we need only one communication step. In the case that nonzero elements of matrix A are located in limited parts, using the communication table is quite effective.

In the method 5, since both MPI_Send and MPI_Recv functions are blocking communication, execution of other instructions is suspended. However this method saves starting time for the communication. If the total amount of communicated data is small, we can get high performance by selecting the order of communication.

As in the case of the matrix format, the actual time of the matrix-by-vector product was measured and the best way from the alternatives above was selected.

3.4 Restarting frequency

The larger the restarting frequency m, the smaller the iteration count we need. However if m is large, the execution time of one iteration increases with the increment of iteration counts. Because in the orthogonalization, we must caluculate a new vector to be orthogonalized to all vectors which have caluculated by earlier iteration (lines 8-11 in the Figure 1). The total amount of calculations for the orthogonalization is proportional to the square of the iteration counts.

There are many ways to decide on m [5]. In our implementation we change the value of m dynamically [6]. Here, let $m_{\max}$ be maximal restarting frequency. We decide on the value of m as follows:

(1) m=2 (initial value).
(2) Add 2 to m if $m < m_{\max}$.
(3) Back to (1).

Our library sets 128 to $m_{\max}$. If the library cannot allocate memory, it sets the maximal size to $m_{\max}$ within the maximum permissible memory allocation. The reason why we decide $m_{\max}$ is to save the amount of calculation for the orthogonalization.

3.5 Gram-Schmidt orthogonalization

Modified Gram-Schmidt orthogonalization (MGS) is often used on single processor machines because of its smaller computational error. However on distributed memory machines, it is not efficient because of the frequent synchronization especially in the case of a large iteration count.

On the other hand, classical Gram-Schmidt orthogonalization (CGS) is efficient on distributed memory machines because the synchronization is needed only once.

In case of using CGS, lines 8–11 in the Figure 1 are replaced as shown in Figure 5.

CGS has less in computational error than MGS. Therefore, our library provides *iterative refinement Gram-Schmidt orthogonalization* [4] (see Figure 6).

```
1:   for k=0,1,···,i
2:     h_{k,i} = (w, v_k)
3:   end
4:   for k=0,1,···,i
5:     w = w − h_{k,i} v_k
6:   end
```

Fig. 5. Classical Gram-Schmidt orthogonalization

```
1:   for k=0,1,···,i
2:     h_{k,i} = (w, v_k)
3:   end
4:   for k=0,1,···,i
5:     w = w − h_{k,i} v_k
6:   end
7:   for k=0,1,···,i
8:     ĥ_k = (w, v_k)
9:     h_{k,i} = h_{k,i} + ĥ_k
10:  end
11:  for k=0,1,···,i
12:    w = w − ĥ_k v_k
13:  end
```

Fig. 6. Iterative refinement Gram-Schmidt orthogonalization

The execution time by this method is twice as large as primary CGS. However it is comparable to MGS in computational error. In our library, we measure the orthogonalization time for calculating a new vector to be orthogonalized to $m_{\max}/2$ vectors. We then select the fastest method, MGS or CGS.

On single processor machines, the MGS is advantageous to the CGS because of localization of memory access. On the other hand, on distributed memory machines, it is not clear which is the best because we must consider the combination of cache memory size, the number of vectors, the number of PEs, interprocessor communications speed and so on.

When the convergence is not improved at two straight steps, we change to the iterative refinement Gram-Schmidt orthogonalization.

3.6 Preconditioning

There are many occasions and applications where iterative methods fail to converge or converge very slowly. Therefore, it is important to apply preconditioning.

In our library, we apply *diagonal scaling* to a coefficient matrix A. In this case, we expect that not only it helps to reduce the condition number and often has a beneficial influence on the convergence behavior but also the computational complexity and memory allocation are reduced by fixing to 1 in all diagonal elements. In addition to the diagonal scaling, we can select the following three implementations.

1. No preconditioning.
2. Polynomial Preconditioning [7].
3. Block incomplete LU decomposition [8].

Let A be the scaled matrix such that $\mathrm{diag}(A) = I$.

In case 2, the matrix A can be written $A = I - B$, and A^{-1} can be evaluated in a Neumann series as

$$A^{-1} = (I - B)^{-1} = I + B + B^2 + B^3 + \cdots . \tag{2}$$

We take a truncated Neumann series as the preconditioner, e.g. approximating A^{-1} by $K^{-1} = I + B$. In this case, K^{-1} is very similar to A but plus and minus signs of elements of K^{-1} are reversed except for the diagonal elements. Since this preconditioning does not need extra memory allocation which holds matrix $I + B$ data, in particular GMRES(m) which requires a lot of memory allocation, it is very useful.

However, approximation of A^{-1} by $I + B$ is efficient only when matrix A is diagonally dominant, namely, spectral radius of B satisfies the relations $\rho(B) < 1$. If $\rho(B) \geq 1$ then $I + B$ does not approximate A^{-1}.

In the preconditioner 3, our library employs zero fill-in ILU factorization called as ILU(0) on each individual block, which is diagonal submatrix on each PE. In this case, we assume $K = LU$.

In lines 7 and 28 in the the Figure 1, there is the matrix-by-vector product in the form of $K^{-1}r$. When we assume $q = (LU)^{-1}r$, we can solve linear system of equations $LUq = r$, where q is the vector of unknowns.

To solve q of the linear system is as follows.

$$\begin{array}{ll} Lz = r & \text{(Forward substitution)} \\ Uq = z & \text{(Backward substitution),} \end{array} \tag{3}$$

where z is a temporary vector. As shown above, it is possible to calculate $K^{-1}r$.

If restarting frequency m is large, the preconditioning is more efficient. Because the overhead time of preconditioning depends on the whole iteration count to converge, setting m at a large value reduces the total iteration count.

The best preconditioning selection is as follows. We iterate the main loop (lines 6–24 in the Figure 1) for $m = 2$ by using every method. Next, we select the method whose relative decrease of the residual norm ($||r_2||/||r_0||$) is the smallest. Note that we do not consider the execution time, we employ the method which reduces the residual norm the most after the same number of iterations.

4 Experimental results

We implemented our auto-tuning methodology on the HITACHI SR2201 and HITACHI SR8000.

The HITACHI SR2201 system is a distributed memory, message-passing parallel machine of the MIMD class. It is composed of 1024 PEs, each having 256

Megabytes of main memory, interconnected via a communication network having the topology of a three-dimensional hyper-crossbar. The peak interprocessor communications bandwidth is 300 Mbytes/s in each direction. We used the HITACHI Optimized Fortran90 V02-06-/D compiler, and compile option we used was *-W0,'opt(o(ss),fold(1))'*. We also used the HITACHI Optimized C compiler, and compile option we used was *+O4 -Wc,-hD1*.

The HITACHI SR8000 system is a distributed memory, message-passing parallel machine of the MIMD class like the HITACHI SR2201. It is composed of 128 nodes, each having 8 Instruction Processors (IPs), 8 Gigabytes of main memory, interconnected via a communication network having the topology of a three-dimensional hyper-crossbar. The peak interprocessor communications bandwidth is 1 Gbytes/s in each direction. We used the HITACHI Optimized Fortran90 V01-00 compiler, and compile option we used was *-W0,'opt(o(4),fold(1))' -noparallel*. We also used the HITACHI Optimized C compiler, and compile option we used was *+O4 -Wc,-hD1 -noparallel*.

We evaluated performance with the following conditions:

- Convergence result : $||r_k||/||r_0|| < 1.0 \times 10^{-12}$
- Initial guess : $x_0 = (0, 0, \cdots, 0)^T$
- Precision type : double

4.1 Test problems

Evaluation on our library by employing three problems whose maximal number of nonzero elements of a matrix A at each row is 3, 5, and 7.

Problem 1

The coefficient matrix A is a Toeplitz matrix such as

$$A = \begin{bmatrix} 2 & 1 & & & \\ 0 & 2 & 1 & & \\ R & 0 & 2 & 1 & \\ & R & 0 & 2 & \cdots \\ & & \cdots & \cdots & \end{bmatrix},$$

where $R = 1.0$, 1.5, and 2.0. The right-hand side vector is $b = (1, 1, \cdots, 1)^T$. The size of matrix A is 4,000,000.

Problem 2

An elliptic boundary value problem of partial differential equation:

$$-u_{xx} - u_{yy} + Ru_x = g(x, y) \ ,$$
$$u(x, y)|\partial\Omega = 1 + xy \ ,$$

where the region is $\Omega = [0, 1] \times [0, 1]$, and $R = 1.0$. The right-hand side vector b is set to the exact solution of $u = 1 + xy$. We discretize the region by using a 5-point difference scheme on a 400×400 mesh. The size of matrix A is 160,000.

Problem 3

An elliptic boundary value problem of partial differential equation:

$$-u_{xx} - u_{yy} - u_{zz} + Ru_x = g(x,y,z) \ ,$$
$$u(x,y)|\partial\Omega = 0.0 \ ,$$

where the region is $\Omega = [0,1]\times[0,1]\times[0,1]$, and $R = 1.0$ and 100.0. The right-hand side vector b is set to the exact solution of $u = e^{xyz}\sin(\pi x)\times\sin(\pi y)\times\sin(\pi z)$. We discretize the region by using a 7-point difference scheme on a $80\times80\times80$ mesh. The size of matrix A is 512,000.

4.2 The results

Tables 1–3 show the execution time on each problem in the case of no auto-tuning, auto-tuning, and using PETSc[4]. In addition, they show auto-tuned parameters in the auto-tuning case.

The calculation time of the QR decomposition in the lines 14–23 of Figure 1 was less than 1 second on every problem. Even though each PE contains the QR decomposition, this overhead time was very small and it can be ignored.

Following parameters were set as the sample of the no auto-tuning case. These are common parameters which were used comparison with the no auto-tuning case and the auto-tuning case.

Matrix storage format	: Compressed row storage format for unrolling.
Unrolling	: Non-prefetch and no unrolling code.
Communication	: Use MPI_Allgather funuction from the MPI library.
Restarting frequency	: 30 (fixed)
Orthogonalization	: Iterative refinement Gram-Schmidt.
Preconditioning	: None.

In case of the auto-tuning version the leftmost explanation has the following meaning.

iter. : Iteration count.
time : Total execution time including auto-tuning. (sec)
unro. : Unrolling type. For example, P(2,3) means prefetch code,
two outer loops expanded, and three inner loops expanded.
On the other hand, N(2,3) means non-prefetch code.
com. : Communication type.
Send $\cdots$ use MPI_Send and MPI_Recv in pairs.
Isend $\cdots$ use MPI_Isend and MPI_Irecv in pairs.
Irecv $\cdots$ use MPI_Irecv and MPI_Isend in pairs.
orth. : Orthogonalization type.
prec. : Preconditioning type.
$I+B$ $\cdots$ polynomial preconditioning.
BILU $\cdots$ block incomplete LU decomposition.

The matrix storage format in Tables 1–3 has been omitted since the compressed row storage format for unrolling is selected in all problems.

As for PETSc, we used it with almost default parameter values. For example, the restarting frequency is 30, the technique for orthogonalization is the iterative refinement Gram-Schmidt method and so on. Only the convergence is decided to be set 10^{-12} in order to compare with our library.

Table 1. The results for problem 1

(a) R=1.0 **SR2201**

PE	8	16	32	64	128
No auto-tuning					
iter.	43	43	43	43	43
time	49.6	36.7	25.0	20.9	22.0
Auto-tuning					
iter.	18	19	19	19	20
time	40.5	24.0	14.3	8.3	5.3
unro.	N(1,3)	N(1,3)	N(3,3)	N(3,3)	P(3,3)
com.	Send	Send	Irecv	Irecv	Send
orth.	MGS	MGS	CGS	CGS	CGS
prec.	BILU	BILU	BILU	BILU	BILU
PETSc					
iter.	20	20	21	21	21
time	93.1	45.9	24.5	11.9	6.0

(b) R=1.5 **SR2201**

PE	8	16	32	64	128
No auto-tuning					
iter.	93	93	93	93	93
time	101.2	85.5	56.7	45.7	44.3
Auto-tuning					
iter.	50	93	93	93	93
time.	72.5	29.1	16.6	9.2	5.7
unro.	N(1,3)	N(1,3)	N(3,3)	N(3,3)	N(3,3)
com.	Send	Send	Irecv	Send	Send
orth.	MGS	MGS	CGS	CGS	CGS
prec.	BILU	None	None	None	None
PETSc					
iter.		55	56	57	58
time	fail.	148.0	75.5	38.2	19.7

(c) R=2.0 **SR2201**

PE	8	16	32	64	128
No auto-tuning					
iter.	337	323	323	323	323
time	353.3	277.6	186.9	153.9	143.9
Auto-tuning					
iter.	332	321	321	321	321
time	124.9	86.6	45.0	22.8	13.7
unro.	N(1,3)	N(1,3)	N(3,3)	N(3,3)	P(3,3)
com.	Send	Send	Send	Send	Send
orth.	MGS	MGS	CGS	CGS	CGS
prec.	None	None	None	None	None
PETSc					
iter.					
time	fail.	fail.	fail.	fail.	fail.

(d) R=1.0 **SR8000**

IP	8	16	32	64	128
No auto-tuning					
iter.	43	43	43	43	43
time	25.3	19.0	20.4	20.0	25.1
Auto-tuning					
iter.	18	19	19	19	20
time	23.8	12.9	7.6	5.9	5.1
unro.	N(1,3)	P(2,3)	N(1,3)	N(1,3)	N(1,3)
com.	Send	Send	Send	Send	Irecv
orth.	MGS	MGS	CGS	CGS	CGS
prec.	BILU	BILU	BILU	BILU	BILU

(e) R=1.5 **SR8000**

IP	8	16	32	64	128
No auto-tuning					
iter.	93	93	93	93	93
time	53.7	40.0	43.1	41.8	52.8
Auto-tuning					
iter.	50	93	93	93	93
time	38.7	11.9	7.1	5.7	5.7
unro.	N(2,3)	P(2,3)	N(1,3)	N(1,3)	N(2,3)
com.	Irecv	Send	Send	Send	Send
orth.	MGS	CGS	CGS	CGS	CGS
prec.	BILU	None	None	None	None

(c) R=2.0 **SR8000**

IP	8	16	32	64	128
No auto-tuning					
iter.	323	323	323	323	323
time	180.3	134.8	145.8	141.4	178.8
Auto-tuning					
iter.	321	321	321	321	321
time	52.2	27.3	14.6	9.6	9.2
unro.	N(2,3)	N(2,3)	N(2,3)	N(1,3)	N(1,3)
com.	Irecv	Irecv	Send	Send	Irecv
orth.	MGS	CGS	CGS	CGS	CGS
prec.	None	None	None	None	None

Table 2. The results for problem 2

(f) R=1.0 **SR2201**

PE	8	16	32	64	128
No auto-tuning					
iter.	21842	21842	21842	21842	21842
time	1205.7	769.3	540.9	520.3	413.1
Auto-tuning					
iter.	1349	1328	1429	1596	1497
time	90.7	44.3	24.3	14.1	7.9
unro.	P(3,5)	P(3,5)	P(2,5)	P(2,5)	P(2,5)
com.	Send	Send	Send	Send	Send
orth.	CGS	CGS	CGS	CGS	CGS
prec.	BILU	BILU	BILU	BILU	BILU
PETSc					
iter.	2614	2049	2913	3213	3934
time	576.0	219.3	153.7	81.0	57.0

(a) R=1.0 **SR8000**

IP	8	16	32	64	128
No auto-tuning					
iter.	21842	21842	21842	21842	21842
time	499.7	332.5	278.3	225.8	
Auto-tuning					
iter.	1349	1328	1429	1596	1497
time	45.8	23.6	12.3	7.6	4.5
unro.	P(1,5)	P(1,5)	P(1,5)	P(2,5)	N(1,5)
com.	Send	Send	Send	Send	Send
orth.	CGS	CGS	CGS	CGS	CGS
prec.	BILU	BILU	BILU	BILU	BILU

Table 3. The results for problem 3

(a) R=1.0 **SR2201**

PE	8	16	32	64	128
No auto-tuning					
iter.	1265	1265	1265	1265	1265
time	250.6	149.4	98.9	90.8	80.9
Auto-tuning					
iter.	288	300	417	417	417
time	81.9	42.5	12.5	7.3	4.7
unro.	P(2,7)	P(2,7)	P(1,7)	P(1,7)	P(1,7)
com.	Isend	Isend	Isend	Isend	Isend
orth.	MGS	CGS	CGS	CGS	CGS
prec.	BILU	BILU	$I+B$	$I+B$	$I+B$
PETSc					
iter.	236	254	343	465	538
time	182.2	96.9	67.0	44.2	25.2

(b) R=100.0 **SR2201**

PE	8	16	32	64	128
No auto-tuning					
iter.	626	626	626	626	626
time	124.2	72.1	50.9	44.7	39.3
Auto-tuning					
iter.	176	199	207	306	272
time	45.7	25.3	13.3	11.3	4.3
unro.	P(2,7)	P(1,7)	P(2,7)	P(1,7)	P(1,7)
com.	Isend	Isend	Isend	Irecv	Isend
orth.	MGS	CGS	CGS	CGS	CGS
prec.	BILU	BILU	BILU	BILU	BILU
PETSc					
iter.	203	262	331	310	370
time	156.8	100.6	65.6	29.7	17.1

(c) R=1.0 **SR8801**

IP	8	16	32	64	128
No auto-tuning					
iter.	1265	1265	1265	1265	1265
time	111.4	66.4	42.0	31.5	
Auto-tuning					
iter.	288	300	417	417	417
time	43.9	23.5	6.8	4.8	4.4
unro.	P(1,7)	P(1,7)	P(1,7)	P(1,7)	P(1,7)
com.	Irecv	Isend	Isend	Irecv	Irecv
orth.	CGS	CGS	CGS	CGS	CGS
prec.	BILU	BILU	$I+B$	$I+B$	$I+B$

(d) R=100.0 **SR8801**

IP	8	16	32	64	128
No auto-tuning					
iter.	598	598	598	598	598
time	53.1	31.6	20.0	15.0	
Auto-tuning					
iter.	176	199	207	306	272
time	24.7	14.0	7.8	7.1	3.8
unro.	P(1,7)	P(1,7)	P(1,7)	P(1,7)	P(1,7)
com.	Irecv	Irecv	Isend	Irecv	Irecv
orth.	CGS	CGS	CGS	CGS	CGS
prec.	BILU	BILU	BILU	BILU	BILU

Comparison with no auto-tuning and auto-tuning If the problem size is large, the execution time of auto-tuning is relatively smaller as compared to the total execution time. Tables 1–3 show that auto-tuning method works very well.

Unrolling type In problem 1, non-prefetch code is selected as the unrolling type. In the other problems, prefetch code is selected. In problem 1, our library often selects the code of 3-unrolled outer loop and 3-unrolled inner loop because loop size is small. In problem 3, it often selects no expanded code for the outer loop. These results mean that auto-tuning behavior depends on machine architectures and compilers.

Communication type Since the nonzero elements of a coefficient matrix A was located at near diagonal intensively in all problems, the communication table usage was selected. In the problems 1 and 2, since communication data size was small, the method using MPI_Send and MPI_Recv in pairs was selected so often. In the problem 3, since communication data size was large, the method using MPI_Isend and MPI_Irecv in pairs was selected.

Orthogonalization type If the number of PEs became large, the selected method was changed from the MGS into the CGS. However the number of PEs where the changes happen is different in each problem. For example it changed into the CGS for 32 PEs in problem 1, for 8 PEs in problem 2, and for 16 PEs in problem 3.

Preconditioning type In many cases, BILU was selected as preconditioner. Table 3 shows that $I + B$ is included in the selected method. Because when the number of PEs is large, preconditioning effect of using BILU is small. On the other hand, preconditioning with $I + B$ is invariable and it has nothing to do with the change of the number of PEs.

Comparison to the PETSc In the Tables 1 (b) and 1 (c), the PETSc failed to converge. In this case, users have to set parameters suitably. On the whole, our library is approximately four times as fast as the PETSc library.

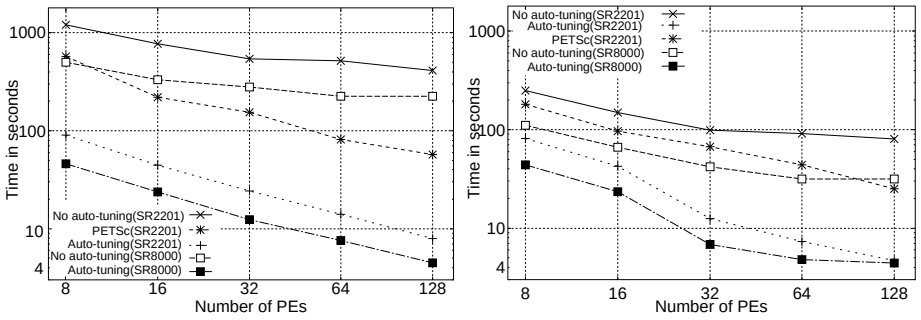
Scalability The execution time is reduced with the number of PEs. Speed-ups for some problems are shown in Figure 7.

5 Conclusion

Selecting optimal codes to get high performance is very important. It brings not only effective utilization of computer resource but also highly user friendly library.

How we can get high performance without setting parameters in detail will be the center of public interest.

Our library is open source and available on-line from out project home page at http://www.hints.org/. Evaluation on the other parallel machines are part of the future work.



(a) Problem 2, R=1.0 (b) Problem 3, R=1.0

Fig. 7. Speed-ups in all problems

Acknowledgments

The authors are much obliged to Dr. Aad van der Steen at the Utrecht University for giving us useful comments in this paper. This research is partly supported by Grant-in-Aid for Scientific Research on Priority Areas "Discovery Science" from the Ministry of Education, Science and Culture, Japan.

References

1. J.J.Dongarra, I.S.Duff, D.C.Sorensen, H.A.van der Vorst: Numerical Linear Algebra for High-Performance Computers. SIAM (1998).
2. Y.Saad: Iterative Methods for Sparse Linear Systems. PWS Publishing Company (1996).
3. H.Kuroda, T.Katagiri, Y.Tsukuda, Y.Kanada: Constructing Automatically Tuned Parallel Numerical Calculation Library — A Case of Symmetric Sparse Linear Equations Solver —. *Proc. 57th National Convention IPSJ*, No.1, pp.1-10 – 1-11 (1998) in Japanese.
4. S.Balay, W.D.Gropp, L.C.McInnes, B.F.Smith: PETSc 2.0 Users Manual. ANL-95/11 - Revision 2.0.24, Argonne National Laboratory (1999).
5. N. Tsuno, T. Nodera: The Speedup of the GMRES(m) Method Using the Early Restarting Procedure. *Trans.IPS.Japan*, Vol.40,No.4,pp.1760–1773 (1998) in Japanese.
6. H. Kuroda, Y. Kanada: Performance of Automatically Tuned Parallel Sparse Linear Equations Solver. *IPSJ SIG Notes* 99-HPC-76-3, pp. 13–18 (1998) in Japanese.
7. O.G.Johnson, C.A.Micchelli, G.Paul: Polynomial Preconditioners for Conjugate Gradient Calculations. *SIAM J. Numer. Anal.*, Vol.20,No.2 (1983).
8. J.S.Kowalik, S.P.Kumar: An Efficient Parallel Block Conjugate Gradient Method for Linear Equations. *Proc.* 1982 *Int. Conf. Par. Proc.*, pp. 47–52 (1982).