A Methodology for Automatically Tuned Parallel Tridiagonalization on Distributed Memory Vector-Parallel Machines

Takahiro Katagiri[12] *, Hisayasu Kuroda[1], 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
{katagiri, kuroda, kanada}@pi.cc.u-tokyo.ac.jp

Abstract. In this paper, we describe an auto-tuning methodology for the parallel tridiagonalization to attain high performance. By searching the optimal set of three parameters for the performance, a highly efficient routine can be obtained automatically. Evaluation of the methodology on the distributed memory parallel machines, the HITACHI SR2201 and HITACHI SR8000, has been provided. The experimental results show 1.3–1.8 times speed-up to a not auto-tuned routine which was specified with reasonable parameters, and the ratios increased for growing problem sizes. Comparison between the execution time of our routine with that of the ScaLAPACK's routine shows that our auto-tuned routine is faster in many cases.

1 Introduction

Tuning computational kernels is time-consuming work. We still have to use several techniques to attain high performance. To avoid the tuning work, many linear algebra programs are constructing by using vendor-tuned BLAS (Basic Linear Algebra Subprograms) routines. The BLAS routines give us high efficiency if the BLAS routines were implemented optimally. However, if the BLAS routines were implemented with low efficiency, the performance will be poor. Solution for such implementation problem for BLAS is to use auto-tuning software for BLAS, such as PHiPAC [1] or ATLAS [11]. We call these software as *auto-tuning software for general usage*.

On the other hand, tuning software automatically that does not or can not use BLAS is hard. Accordingly, every piece of software that can be tuned automatically has a special auto-tuning facility. For example, FFTW [4] for the

* *Candidate to the Best Student Paper Award*

discrete Fourier transformation, and the auto-tuning libraries [9] for sparse linear equation solvers. We call these software packages as *auto-tuning software for dedicated usage*. This paper includes that the report of the development of such auto-tuning software for dedicated usage. The reasons for this report are as follows:

1. Presently, auto-tuning software for parallel processing is not available.
2. We believe that an auto-tuning facility should be contained in each package.

As for reason 2, if the auto-tuning facility is separated from the package, users will be in trouble to attain high performance, because they have to install auto-tuning software into their environments separately. In addition, the time needed for auto-tuning may be enormous because it may tune even non-relevant subroutines (consider the tuning time of all BLAS subroutines.) Hence, our routine contains this auto-tuning facility.

This paper is organized as follows. Description of our parallel dense eigensolver in Section 2. Section 3 is about the parameters of auto-tuning, and how to search for the optimal parameters. In Section 4, we show the results of the auto-tuned parameters and execution time of our routines using the auto-tuning methodology on the HITACHI SR2201 and HITACHI SR8000. The result of the SR2201 includes a comparison with the ScaLAPACK routine. Finally, Section 5 gives the conclusion of this paper.

2 Dense symmetric eigensolver

2.1 Entire process

Our eigensolver can perform the following eigendecomposition:

$$A = X\Lambda X^{-1}, \tag{1}$$

where $A \in \mathbb{R}^{n\times n}$ is a symmetric dense matrix, $\Lambda \in \mathbb{R}^{n\times n}$ is a diagonal matrix which contains eigenvalues $\lambda_i \in \mathbb{R}$, $i = 1, 2, ..., n$ as the i-th diagonal elements, and $X \in \mathbb{R}^{n\times n}$ is a matrix which contains eigenvectors $x_i \in \mathbb{R}^n$ as the i-th row vectors, where n is problem size. In our eigensolver, the decomposition (1) is performed by using a well-known method, the Householder-bisection method. To perform the Householder-bisection method, the following four processes are needed.

1. Tridiagonalization by the Householder method: $T = QAQ$.
2. Eigenvalues of the tridiagonal matrix T are calculated by using the bisection method.
3. By using the inverse-iteration method, eigenvectors of the tridiagonal matrix T are calculated.
   (The processes 2 and 3 yield the eigendecomposition $T = Y\Lambda Y^{-1}$.)
4. Reconstructing eigenvalues for the matrix A : $X = QY$.

Concerning the above four processes, the processes 1 and 4 can affect the whole performance if we need no orthogonalization in process 3. Process 4 depends on the data distribution of the matrices Q and Y [5]. For this reason, determining the optimal parallelization of process 4 is hard, and hence, the parallelization has not been treated in this paper. Next section describes how to parallelize process 1.

2.2 Householder tridiagonalization

Consider the following transformation: $A^{(1)} \equiv A$ to tridiagonal form $A^{(n-2)}$, where $A^{(k)}$ is defined as the k-th iteration of the matrix A. This transformation is denoted by $H^{(k)}(x) = H^{(k)}(A^{(k)}_{k:n,k})$, where $A^{(k)}_{k:n,k}$ is a row vector of A which is constructed by the k-th row and from the k-th to the n-th columns in the k-th iteration. By substituting $H^{(k)} = I - \alpha uu^T$ for $H^{(k)}(x)$ in the $k+1$-th iteration, the following equations are derived:

$$\begin{aligned} A^{(k+1)} &= H^{(k)} A^{(k)} H^{(k)} \\ &= A^{(k)} - \alpha A^{(k)} uu^T - \alpha uu^T A^{(k)} + \alpha^2 uu^T A^{(k)} uu^T \\ &= A^{(k)} - xu^T - uy^T + \alpha uu^T xu^T \\ &= A^{(k)} - uy^T + u\mu u^T - xu^T \\ &= A^{(k)} - u(y^T - \mu u^T) - xu^T, \end{aligned} \tag{2}$$

where

$$x = \alpha A^{(k)} u, \quad y^T = \alpha u^T A^{(k)}, \quad \mu = \alpha u^T x. \tag{3}$$

Here $\alpha, \mu \in \mathbb{R}$, and $u, x, y \in \mathbb{R}^n$. As matrix A is symmetric, $x = y^T$, and we obtain the following formula:

$$A^{(k+1)} = A^{(k)} - u(x^T - \mu u^T) - xu^T. \tag{4}$$

Note that to execute the k-th iteration, the column vector $A_{k:n,k}$ from the partial matrix $A_{k:n,k:n}$ is needed.

2.3 Parallel implementation of the Householder tridiagonalization

Let p be the number of processor elements (PEs). The objective matrix A is distributed by $r \times q$ 2-D grid distribution, called grid-wise distribution (Cyclic, Cyclic), where $r \times q = p$. The grid-wise distribution (Cyclic, Cyclic) distributes the elements of A to the following PEs:

$$a_{ij} \mapsto P_{(i \bmod r,\ j \bmod q)}, \tag{5}$$

where the $P_{(idx,idy)}$, $(idx = 0, 1, ..., r-1,\ idy = 0, 1, ..., q-1)$ means the PE identification number on the 2-D grid distribution. We do not support block-cyclic distribution because the block-cyclic distribution causes poor load balance when n/p is small.

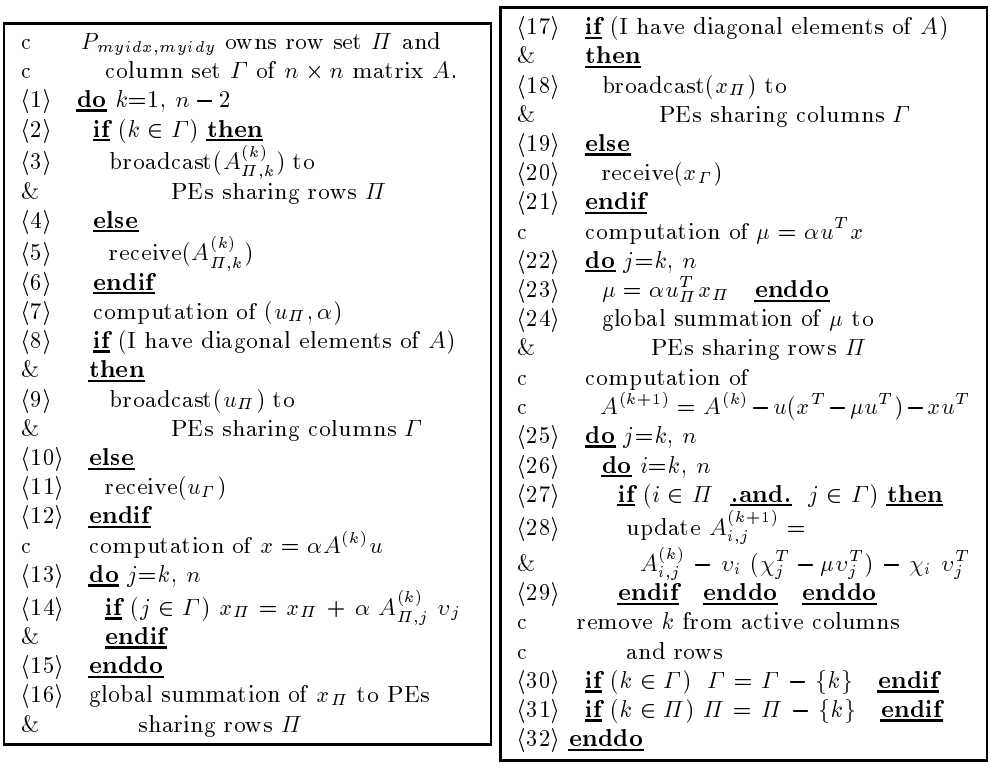


Fig. 1. Parallel algorithm for the tridiagonalization (the (Cyclic, Cyclic) grid-wise distribution).

We already developed the parallel tridiagonalization and Hessenberg reduction routines [8] by the Householder transformation. Figure 1 shows our parallel tridiagonalization algorithm. The routine of Figure 1 reduces communication and broadcast times for vector reduction to a ratio of $1/\sqrt{p}$. The same idea appears in [3, 6, 5]. Symmetry of the matrix A was not used in the algorithm of Figure 1, and hence, the algorithm has the computational complexity of $8n^3/3$, while the algorithm using symmetry has $4n^3/3$. This is because, the algorithm based on the symmetry causes complex data accesses, and the complex data accesses prevent easy parallel implementation.

Figure 1 gives a conclusion that implementations of the following three operations affect the total performance.

1. The global summations of the lines ⟨7⟩, ⟨16⟩, and ⟨24⟩.
2. The matrix-vector product of the lines ⟨13⟩–⟨15⟩.
3. The process to update the matrix A of the lines ⟨25⟩–⟨29⟩.

These three operations are the basic operations for parallel tridiagonalization, and the system will tune the three basic operations automatically in our autotuning process.

3 Method for searching parameters

In this section, the method of tuning the three basic operations automatically is described. Hereafter, we use MPI (Message Passing Interface) as the communication library.

3.1 Parameter for the global summations

To perform the global summations, the following two implementations were selected.

1. A routine based on the binary tree-structured communication, or
2. The `MPI_ALLREDUCE` function on MPI.

It depends on the implementation of MPI functions which implementation has the higher performance. Hence, measuring their real performance is necessary to select the best implementation. For that reason, our auto-tuning routine has a parameter for the above two implementations.

3.2 Parameter for the matrix-vector product

To perform the parallel matrix-vector product ($x = \alpha A^{(k)} u$) at high performance, the size of the stride for loop unrolling must be selected. The size of the stride depends on the machine architectures, operating systems, and compilers we use. Therefore, selecting the optimal number of stride without measuring its real execution time is hard.

For example, a three-stride unrolled routine on the matrix-vector product are shown, where the value of `ilocal_length_x` can be divided by 3 to simplify the explanation.

```
m = ilocal_length_x/3
j = 1
do k=1, m
  dt1 = 0.0d0
  dt2 = 0.0d0
  dt3 = 0.0d0
  do i=1, ilocal_length_y
    du_y = u_y(i)
    ix = init_x+i
    iy = init_y+j
    dt1 = dt1 + A(ix, iy  ) * du_y
    dt2 = dt2 + A(ix, iy+1) * du_y
    dt3 = dt3 + A(ix, iy+2) * du_y
  enddo
  x_k(j  ) = dt1 * al
  x_k(j+1) = dt2 * al
  x_k(j+2) = dt3 * al
```

```
  j = j + 3
enddo
```

This example shows a case of the loop unrolling for the outer-loop `k` only. We can unroll the inner-loop `i` or both of the loops `k` and `i`. Current target machines are vector architecture machines as explained in the Section 4. Then, we only unrolled the outer-loop, since unrolling the inner-loop shortens the loop length which is not good for vector architecture machines. For the auto-tuning parameter, we take the size of the stride.

3.3 Parameter for the process to update

As in the case of the matrix-vector product, it is necessary to set the size of the stride for unrolling in the process to update ($A^{(k+1)} = A^{(k)} - u(x^T - \mu u^T) - xu^T$). For example, a two-stride unrolled routine on the process to update is shown, where the value of `ilocal_length_x` also can be divided by 2 to simplify the explanation.

```
m = ilocal_length_x/2
do k=1, m
  j = 2*(k-1)+1
  dtu1 = u_x(j  )
  dtu2 = u_x(j+1)
  dtr1 = mu * dtu1 - x_k(j  )
  dtr2 = mu * dtu2 - x_k(j+1)
  do i=1, ilocal_length_y
    du_y = u_y(i)
    dx_k = x_k(i)
    ix = init_x+i
    iy = init_y+j
    A(ix, iy  ) = A(ix, iy  ) + du_y * dtr1 - dx_k * dtu1
    A(ix, iy+1) = A(ix, iy+1) + du_y * dtr2 - dx_k * dtu2
  enddo
enddo
```

For the same reason as for the matrix-vector product, we only unroll the outer loop `k`. The auto-tuning parameter for the process to update is the stride for unrolling.

3.4 How to search these parameters

Let the parameters for the global summation, the matrix-vector product, and the process to update be denoted as `Comm.Type`, `Mat-Vec`, and `Updating`, respectively. The `Comm.Type` can take on the values { `Tree`, `MPI_ALLREDUCE` }, where `Tree` means a routine based on binary tree-structured communication, and the `MPI_ALLREDUCE` means communication by a MPI function. The `Mat-Vec` can have the values { `None`, 2, 3, 4, 5, 6, 8, 16 }, where the numbers show the size

of the stride for unrolling. The `Updating` can be chosen as { `None`, `2`, `3`, `4`, `5`, `6`, `8`, `16` } like in the `Mat-Vec` case.

Following is the description of how to search for optimal parameters. We first set the following default parameter values:

$$\texttt{Comm.Type} \equiv \texttt{Tree},\ \texttt{Mat-Vec} \equiv 8,\ \texttt{Updating} \equiv 6. \tag{6}$$

Secondly, we searched the optimal parameters by using the above initial parameters. Method for varying the parameters is as follows.

1. `Comm.Type`=`Tree`, `Mat-Vec`=`8`, and `Updating` is varied as { `None`, `2`, `3`, `4`, `5`, `6`, `8`, `16` }.
2. `Comm.Type`=`Tree`, `Mat-Vec` is varied as { `None`, `2`, `3`, `4`, `5`, `6`, `8`, `16` }, and `Updating`={ the selected value from the process 1 }
3. `Comm.Type` is varied as { `Tree`, `MPI_ALLREDUCE` }, `Mat-Vec`={ the selected value from the process 2 }, and `Updating`={ the selected value from the process 1 }.

This method can not find optimal parameters if there is a dependency among the three parameters. However, the basic operations we mentioned are separated physically (see Figure 1), hence, there is no dependency in the three parameters. Therefore, we may be confident that our method can find an almost optimal set of parameters.

As for the problem sizes, $n = 100$ is specified as the initial values. The problem size is increased by using the stride of 100 while problem size n is under 1000, the stride of 1000 while $1000 \leq n < 10000$, and the stride of 10000 while n is over 10000. This increment is used in each searching process.

4 Experimental results

We implemented the 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 the compile option we used was *-rdma -W0,'OPT(O(SS))'*.

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. The SR8000 system has two types

of parallel environments, named inner-node parallel processing and inter-node parallel processing. The inner-node parallel processing is so-called parallel processing in a sheard memory parallel machine, and there is no interprocessor communication. On the other hand, the inter-node parallel processing is like parallel processing as a distributed memory parallel machine, and it can perform interprocessor communications. We used the HITACHI Optimized Fortran90 V01-00 compiler, and compile option we used was *-W0,'OPT(O(SS)),mp(p(0))'* in the inner-node parallel processing, and *-W0,'OPT(O(SS)),mp(p(4))'* in the inter-node parallel processing.

The communication library used for the SR2201 and SR8000 was MPI. Both machines have vector PEs in a sense, i.e. the Pseudo Vector Processor [2]. Therefore, we can regard both machines as vector-parallel machines.

We implemented our tridiagonalization routine by using dedicated subroutines which satisfy functions for the three parameters. For instance, our routine contains a two-stride unrolled matrix-vector product subroutine, or a three-stride unrolled subroutine to update, and so on. By using such subroutines, we can specify the arbitrary parameters. Note that our software does not generate Fortran codes dynamically in this experiments. All auto-tuning was done at run time.

4.1 The results of the SR2201

Results of auto-tuning Table 1 shows parameters auto-tuned on the SR2201. The tuning time depended on the number of PEs, and the CPU elapsed time

Table 1. The auto-tuned parameters on the SR2201.

(a) Case of 4 PEs

Size	Comm.Type	Mat-Vec	Updating
100	MPI_ALLREDUCE	6	3
200	Tree	8	4
300	Tree	8	6
400	Tree	5	2
500	Tree	8	5
600	Tree	5	6
700	Tree	8	6
800	Tree	3	3
900	Tree	8	4
1000	Tree	5	5
2000	Tree	5	6
3000	Tree	5	5
4000	Tree	3	3
5000	MPI_ALLREDUCE	5	5
6000	MPI_ALLREDUCE	5	5
7000	MPI_ALLREDUCE	5	5
8000	MPI_ALLREDUCE	3	2
	Tuning time	118401 [Sec.]	(32.8 [Hours])

(b) Case of 32 PEs

Size	Comm.Type	Mat-Vec	Updating
100	MPI_ALLREDUCE	6	16
200	MPI_ALLREDUCE	4	5
300	MPI_ALLREDUCE	4	4
400	MPI_ALLREDUCE	6	3
500	MPI_ALLREDUCE	6	4
600	MPI_ALLREDUCE	6	4
700	MPI_ALLREDUCE	8	3
800	MPI_ALLREDUCE	5	3
900	MPI_ALLREDUCE	4	3
1000	MPI_ALLREDUCE	5	3
2000	MPI_ALLREDUCE	5	5
3000	MPI_ALLREDUCE	8	5
4000	MPI_ALLREDUCE	5	5
5000	MPI_ALLREDUCE	8	5
6000	MPI_ALLREDUCE	5	5
7000	MPI_ALLREDUCE	5	5
8000	MPI_ALLREDUCE	3	3
	Tuning time	15555 [Sec.]	(4.3 [Hours])

was about 32 hours at most. The tendency of the tuned parameter of `Comm.Type` were different between 4 and 32 PEs, and the tuned parameters of `Mat-Vec` and `Updating` was different on every problem size. From these facts, we expected that the routine is effective in speeding up.

Comparison to ScaLAPACK To evaluate execution time of the tridiagonalization routine (hereafter TRD), we used the HITACHI optimized ScaLAPACK version 1.2 [7]. Its communication library used was PVM, and PBLAS (Parallel BLAS) which is the computational kernel for ScaLAPACK and is optimized by HITACHI limited. ScaLAPACK's tridiagonalization routine (hereafter SLP TRD) is implemented by using block-cyclic distribution, a blocked algorithm, and symmetry of the matrix [10]. Because of using a blocked algorithm, the size of blocking (BL) can greatly affect the performance of ScaLAPACK. According to [7], if the problem size n is less than 4000, the desirable BL is 60, and if n is over 4000, the desirable BL is 100 on the SR2201. Considering these recommended values, we evaluated the performance of the SLP TRD routines with $BL = \{40, 60, 80, 100, 120\}$ to find which BL gives the best performance. In [7] it is shown that $\sqrt{p} \times \sqrt{p}$ is the best layout for the PE grid. We measured execution time in the PE grid for a large number of PEs. When the number of PEs is small, such as 4, 32, and 64, we measured time in all combinations for the PE grid to find which PE grid gives the best performance.

Table 2 shows execution time of the TRD1 (not auto-tuned), TRD2 (auto-tuned), and SLP TRD. Reasonable parameters of `Comm.Type` $\equiv$ `Tree`, `Mat-Vec` $\equiv$ `8`, and `Updating` $\equiv$ `6` are specified in the TRD1 (not auto-tuned). Note that the optimal BL size and PE grids for the SLP TRD are used, and the values are included in Table 2.

Table 2. Execution time on the SR2201. Unit is in second.

(a) Case of 4 PEs

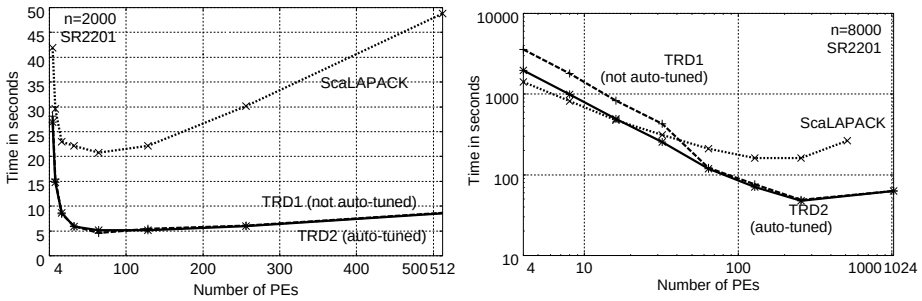
Size	SLP TRD (Grid,BL)	TRD1 (not AT)	TRD2 (AT)	TRD1 /TRD2
100	0.02 (1×4, 100)	0.056 (2×2)	0.056 (2×2)	1.00
200	0.48 (1×4, 100)	0.131 (2×2)	0.133 (2×2)	0.98
400	1.73 (1×4, 40)	0.435 (2×2)	0.475 (2×2)	0.91
800	6.01 (1×4, 40)	3.732 (2×2)	2.454 (2×2)	1.5
1000	9.32 (2×2, 40)	3.817 (2×2)	3.785 (2×2)	1.0
2000	41.90 (2×2, 40)	28.165 (2×2)	26.937 (2×2)	1.0
4000	231.10 (2×2, 40)	411.666 (2×2)	242.010 (2×2)	1.7
8000	1422.69 (2×2, 100)	3589.175 (2×2)	1962.512 (2×2)	1.8

(b) Case of 32 PEs

Size	SLP TRD (Grid,BL)	TRD1 (not AT)	TRD2 (AT)	TRD1 /TRD2
100	0.09 (4×8, 100)	0.108 (4×8)	0.106 (4×8)	1.01
200	0.87 (2×16, 100)	0.250 (4×8)	0.240 (4×8)	1.04
400	2.33 (2×16, 60)	0.514 (4×8)	0.516 (4×8)	0.99
800	6.27 (2×16, 60)	1.207 (4×8)	1.228 (4×8)	0.98
1000	8.28 (2×16, 60)	1.654 (4×8)	1.687 (4×8)	0.98
2000	22.18 (4×8, 40)	5.930 (4×8)	5.886 (4×8)	1.00
4000	72.74 (4×8, 40)	32.961 (4×8)	32.124 (4×8)	1.02
8000	313.25 (4×8, 40)	427.267 (4×8)	254.937 (4×8)	1.6

Table 2 shows that we obtained 1.6–1.8 times speed-ups with respect to the TRD1 (not auto-tuned) when problem sizes were large, such as 4000, 8000. As for the SLP TRD execution time, we find that when problem size is small, the TRD was faster than the SLP TRD. On the other hand, when problem sizes per PE were large, the SLP TRD was faster than the TRD. We consider that this is explained from the computational complexity of the TRD, since the TRD has twice computational complexity to the SLP TRD.

Figure 2 shows the execution time of the TRD1 (not auto-tuned), TRD2 (auto-tuned), and SLP TRD in $n = 2000$ and 8000 cases. Note that the execution time of the SLP TRD in Figure 2 was specified the optimal BL and the PE grid. From Figure 2, we can conclude that when $n = 2000$, the TRD is always faster



(a) Case of $n = 2000$ (b) Case of $n = 8000$

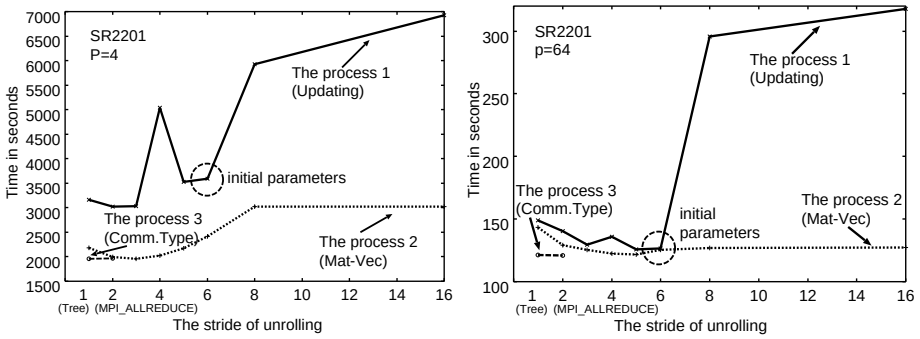
Fig. 2. Execution time for the SLP TRD and TRD in the tridiagonalization (SR2201).

than the SLP TRD, and the speed-up ratios are about 2–6 times. On the other hand, when $n = 8000$, the execution speed of the TRD was slower than the SLP TRD when the number of PEs was under 64, however, when over 64, the TRDs became faster than the SLP TRD. The effect of auto-tuning was high when the number of PEs was under 64.

From the experimental results, we conclude that our methodology is useful, especially, when the problem sizes are large. In addition, the TRD is fast when the problem sizes are small on the SR2201.

The execution time in every auto-tuning process To evaluate the auto-tuning process in detail, we analyzed the execution time in each of our auto-tuning process. Figure 3 shows the time when the problem size was 8000.

From Figure 3 (a), we see that the specified initial parameters (`Comm.Type` $\equiv$ `Tree`, `Mat-Vec` $\equiv 8$, `Updating` $\equiv 6$) were worse than the case of Figure 3 (b), because the 6-stride of the process 1 (`Updating`) and the 8-stride of the process 2 (`Mat-Vec`) in Figure 3 (a) were not optimal parameters, and the change of the elapsed time when varying these strides was high. Hence, we conclude that the initial parameters were not good for the case of 4 PEs, and this caused



(a) Case of $p = 4$ (b) Case of $p = 64$

Fig. 3. The execution time in every auto-tuning process. (SR2201, $n = 8000$)

high speed-up ratios. On the other hand, Figure 3 (b) shows that the initial parameters we specified were almost optimal values. For this reason, we conclude that we did not obtain better speed-ups on 64 PEs than the speed-ups on 4PEs on the SR2201.

4.2 The results of SR8000

Results of auto-tuning and execution time Table 3 shows auto-tuned parameters on the SR8000. From Table 3, the tendency of tuned parameters was found to be different between the inner-node parallel and inter-node parallel environments. From this fact, we could also find the cases for the speed up. Table 4 shows execution time of the TRD1 (not auto-tuned) and TRD2 (auto-tuned). From Table 4, we obtained about 1.1–1.3 times speed-ups with respect to the TRD1 (not auto-tuned). The effect became stronger when problem size increased. So, the authors conclude that our auto-tuning methodology is also useful on the SR8000.

5 Conclusion

The authors have implemented and evaluated a tridiagonalization routine by using an auto-tuning methodology. Selecting suitable implementations for the global summation, the matrix-vector product, and the process to update on the parallel tridiagonalization is the auto-tuning methodology we mentioned in this paper, and the methodology is quite simple. Even though we used this quite simple methodology, we could obtain about 1.1–1.8 times speed-ups with respect to the routine for which the reasonable parameters in the SR2201 and the SR8000 were specified. From these results, the authors concluded that such an auto-tuning methodology is an effective technique.

The auto-tuning methodology is for vector-parallel machines. The auto-tuning methodology for the RISC based parallel machines, such as selecting blocking factors in blocked algorithms, and evaluation on the RISC based parallel machines are parts of the future work.

Table 3. The auto-tuned parameters on the SR8000.

(a) Case of 1 Node (8 IPs) (SR8000, inner-node parallel, sheard memory)

Size	Comm.Type	Mat-Vec Updating	
100	MPI_ALLREDUCE	None	None
200	MPI_ALLREDUCE	4	None
300	MPI_ALLREDUCE	8	None
400	MPI_ALLREDUCE	4	None
500	MPI_ALLREDUCE	5	None
600	MPI_ALLREDUCE	6	3
700	MPI_ALLREDUCE	6	None
800	MPI_ALLREDUCE	6	3
900	MPI_ALLREDUCE	6	None
1000	MPI_ALLREDUCE	6	3
2000	MPI_ALLREDUCE	6	None
3000	MPI_ALLREDUCE	6	None
4000	MPI_ALLREDUCE	6	None
5000	MPI_ALLREDUCE	4	None
6000	MPI_ALLREDUCE	4	None
7000	MPI_ALLREDUCE	6	None
8000	MPI_ALLREDUCE	6	None
Tuning time		16325 [Sec.]	(4.5 [Hours])

(b) Case of 4 Nodes (32 IPs) (SR8000, inter-node parallel, distributed memory)

Size	Comm.Type	Mat-Vec Updating	
100	Tree	None	2
200	Tree	None	None
300	Tree	None	2
400	Tree	None	None
500	Tree	None	None
600	Tree	None	None
700	Tree	None	None
800	Tree	4	None
900	Tree	4	None
1000	Tree	6	None
2000	MPI_ALLREDUCE	6	4
3000	MPI_ALLREDUCE	6	4
4000	MPI_ALLREDUCE	4	16
5000	MPI_ALLREDUCE	4	16
6000	MPI_ALLREDUCE	4	16
7000	MPI_ALLREDUCE	6	16
8000	MPI_ALLREDUCE	6	16
Tuning time		4443 [Sec.]	(1.2 [Hours])

Table 4. Execution time on the SR8000. Unit is in second.

(a) Case of 1 Node (8 IPs) (SR8000, inner-node parallel, sheard memory)

size	TRD1 (not AT)	TRD2 (AT)	TRD1 /TRD2
100	0.024 (2×4)	0.022 (2×4)	1.09
200	0.053 (2×4)	0.049 (2×4)	1.08
400	0.162 (2×4)	0.145 (2×4)	1.11
800	0.678 (2×4)	0.587 (2×4)	1.15
1000	1.155 (2×4)	0.988 (2×4)	1.16
2000	7.098 (2×4)	5.595 (2×4)	1.26
4000	50.451 (2×4)	39.263 (2×4)	1.28
8000	389.297 (2×4)	308.307 (2×4)	1.26

(b) Case of 4 Nodes (32 IPs) (SR8000, inter-node parallel, distributed memory)

Size	TRD1 (not AT)	TRD2 (AT)	TRD1 /TRD2
100	0.038 (2×2)	0.036 (2×2)	1.05
200	0.077 (2×2)	0.072 (2×2)	1.06
400	0.176 (2×2)	0.162 (2×2)	1.08
800	0.490 (2×2)	0.450 (2×2)	1.08
1000	0.714 (2×2)	0.648 (2×2)	1.10
2000	2.806 (2×2)	2.345 (2×2)	1.19
4000	14.957 (2×2)	11.392 (2×2)	1.31
8000	102.369 (2×2)	75.398 (2×2)	1.35

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. Bilmes, K. Asanović, C.-W. Chin, and J. Demmel. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-performance, ANSI C Coding Methodology. In *Proceedings of International Conference on Supercomputing 97* (Vienna, Austria, 1997) 340–347.
2. T. Boku, K. Itakura, H. Nakamura, and K. Nakazawa. CP-PACS: A Massively Parallel Processor for Large Scale Scientific Calculations. In *Proceedings of International Conference on Supercomputing 97* (Vienna, Austria, 1997) 108–115.
3. H. Chang, S. Utku, M. Sakama, and D. Rapp. A Parallel Householder Tridiagonalization Stratagem Using Scattered Square Decomposition. *Parallel Computing* **6** (1988) 297–311.
4. M. Frigo. A Fast Fourier Transform Compiler. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Atlanta, Georgia, 1999) 169–180.
5. B. Hendrickson, E. Jessup, and C. Smith. Toward an Efficient Parallel Eigensolver for Dense Symmetric Matrices. *SIAM J. Sci. Comput.* **20(3)** (1999) 1132–1154.
6. B. A. Hendrickson and D. E. Womble. The Tours-wrap Mapping for Dense Matrix Calculation on Massively Parallel Computers. *SIAM Sci. Comput.* **15(5)** (1994) 1201–1226.
7. HITACHI Ltd. Using ScaLAPACK and PBLAS Libraries for the HITACHI SR2201. *Computer Centre News, the University of Tokyo* **30(2)** (1998) 36–58. in Japanese.
8. T. Katagiri and Y. Kanada. Performance Evaluation of Householder Method for the Eigenvalue Problem on Distributed Memory Architecture Parallel Machine. *IPSJ SIG Notes 96-HPC-62* (1996) 111–116. in Japanese.
9. H. Kuroda and Y. Kanada. Performance of Automatically Tuned Parallel Sparse Linear Equations Solver. *IPSJ SIG Notes, 99-HPC-76* (1999) 13–18. in Japanese.
10. K. S. Stanley. *Execution Time of Symmetric Eigensolver*. Ph.D Thesis, The University of California at Berkeley, 1997.
11. R. C. Whaley and J. J. Dongarra. Automatically Tuned Linear Algebra Software, ATLAS project, http://www.netlib.org/atlas/index.html.