Performance Modeling and Optimal Block Size Selection for a BLAS-3 Based Tridiagonalization Algorithm

Yusaku Yamamoto
Department of Computational Science & Engineering
Nagoya University
Furo-cho, Chikusa, Nagoya, Aichi, 464-8603, Japan
yamamoto@na.cse.nagoya-u.ac.jp

Abstract

We construct a performance model for Bischof \& Wu’s tridiagonalization algorithm that is fully based on the level-3 BLAS. The model has a hierarchical structure, which reflects the hierarchical structure of the original algorithm, and given the matrix size, the two block sizes and the performance data of the underlying BLAS routines, predicts the execution time of the algorithm. Experiments on the Opteron and Alpha 21264A processors show that the model is quite accurate and can predict the performance of the algorithm for matrix sizes from 1920 to 7680 and for various block sizes with relative errors below 10%. The model will serve as a key component of an automatic tuned library that selects the optimal block sizes itself. It can also be used in a Grid environment to help the user find which of the available machines to use to solve his/her problem in the shortest time.

1. Introduction

Computing the eigenvalues of a real symmetric matrix is one of the most important problems in numerical linear algebra and has applications in many areas of computational science and engineering. When the matrix is dense and of a moderate size (say, less than 100,000), the standard procedure is to first reduce the matrix to a tridiagonal matrix by similarity transformations and then compute the eigenvalues of the tridiagonal matrix [7]. Of these two steps, the former part requires about \( \frac{4}{3}N^3 \) work when the matrix size is \( N \) and usually accounts for most of the computational time. We therefore focus on this step in the following.

Reduction to a tridiagonal matrix is usually done with the Householder method. To fully exploit the performance of modern microprocessors, it is necessary to modify the algorithm so that as many operations as possible are performed in the form of the level-3 BLAS, or matrix-matrix multiplications. To this end, Dongarra’s blocked Householder method [6], which aggregates \( M \) Householder transformations and thereby replaces the rank-2 update of the original algorithm with the rank-2M update, is widely used. However, it is well known that this algorithm can use the level-3 BLAS for only half of the total operations and so can achieve 10 to 25% of the peak performance on modern cache-based microprocessors [12].

An alternative approach is to use a two-stage algorithm proposed by Bischof et al. [1], which first transform the input matrix \( A \) into a band matrix of half bandwidth \( L \), and then reduce the band matrix to a tridiagonal matrix. The first part, which requires about \( \frac{2}{3}N^3 \) work, can be done almost entirely with the level-3 BLAS, while the second part requires only \( 6N^2L \) work though it cannot use the level-3 BLAS. Moreover, Wu et al. [11] showed that Dongarra’s blocking technique can be applied to the former part to further enhance cache utilization. The resulting algorithm, which we call Bischof \& Wu’s algorithm, has been shown to achieve more than 50% of the peak performance on various microprocessors [12]. However, to attain such high performance, it is critical to choose the two block sizes \( L \) and \( M \) properly. This is not an easy task since there are two parameters to optimize and their optimal values vary depending on the platform, the BLAS routines used and the matrix size \( N \).

In this paper, we present a performance model for Bischof \& Wu’s algorithm. It is based on the hierarchical approach proposed by Cuenca et al. [2][3], and given the performance model of the underlying BLAS routines, along with the matrix size \( N \) and the block sizes \( L \) and \( M \), gives accurate prediction of the ex-
cution time. This model makes it possible to choose the optimal block sizes prior to execution and can be used as a key component to construct an automatically tuned tridiagonalization routine. Another application of the model would be in a Grid environment, where an accurate performance model will greatly help the user to determine which machine to use to obtain the answer of his/her eigenproblem as quickly as possible.

Performance modeling of linear algebra routines has been studied by many authors [2][3][4][8][9]. However, many of the studies have been limited to relatively simple routines such as the LU and QR decompositions. The main contribution of this paper is that it applied the hierarchical modeling approach to such a complicated and realistic routine as the level-3 BLAS based tridiagonalization and showed that it can predict the performance accurately in this case as well. See Section 5 for more discussion.

The paper is structured as follows: in Section 2, we review Bischof’s & Wu’s algorithm for the tridiagonalization of symmetric matrices. Section 3 gives the details of our performance model. Experimental results that show the effectiveness of our model are given in Section 4. Section 5 compares our approach with related works. Finally, Section 6 gives some conclusion.

2. Bischof & Wu’s algorithm for tridiagonalizing a symmetric matrix

2.1. Transformation into a band matrix

The tridiagonalization algorithm we treat here is composed of two stages, namely, transformation of the input dense matrix into a band matrix and reduction of the band matrix to tridiagonal form. The algorithm for the first stage is shown as Algorithm 1 [1]. Here, \(N\) and \(L\) denote the matrix size and the half bandwidth of the resulting band matrix, respectively, and we assume for simplicity that \(N\) is divisible by \(L\).

The algorithm consists of \(N/L - 1\) steps. The operations at each step is analogous to that of the ordinary Householder method [7], except that the quantities \(\alpha^{(K)}\) and \(\beta^{(K)}\) appearing in the algorithm are \(L\times L\) matrices rather than scalars, and that \(D^{(K)}, U^{(K)}, P^{(K)}\) and \(Q^{(K)}\) are block vectors whose elements are \(L\times L\) matrices. At the \(K\)-th step, we first compute a block Householder transformation \(H^{(K)} = I - U^{(K)}\alpha^{(K)}U^{(K)\dagger}\) that transforms the block vector \(D^{(K)}\) (see Fig. 1) into a block vector whose first block is an upper triangular matrix and the following blocks are zero. By multiplying the matrix at the \((K - 1)\)th step by \(H^{(K)}\) and \((H^{(K)})^{-1}\) from the left and right, respectively, the transformation into a band matrix proceeds by one block (Fig. 1). Thus the transformation is completed in \(N/L - 1\) steps. Note that all the operations at the \(K\)-th step, except for the generation of the block Householder transformation, can be done with the level-3 BLAS. The computational cost is about \(\frac{4}{3}N^3\), which is the same as that of the Householder method.

![Figure 1. The matrices before and after the \(K\)-th step of Bischof’s algorithm.](image)

Algorithm 1: Bischof’s algorithm

```plaintext
do  
  \(K = 1, N/L - 1\)
  Compute the block Householder transformation \(H^{(K)} = I - U^{(K)}\alpha^{(K)}U^{(K)\dagger}\). 
  \([\text{Matrix-block vector multiplication}]\)
  \(P^{(K)} = C^{(K)}U^{(K)\dagger}\alpha^{(K)}\) 
  \(\beta^{(K)} = \alpha^{(K)\dagger}U^{(K)\dagger}P^{(K)}/2\) 
  \(Q^{(K)} = P^{(K)} - U^{(K)}\beta^{(K)}\) 
  \([\text{Rank-2L update of the matrix}]\)
  \(C^{(K)} := C^{(K)} - U^{(K)\dagger}Q^{(K)\dagger} - Q^{(K)}U^{(K)\dagger}\)
  end do
```

As can easily be seen from Algorithm 1, operations at each step consist of matrix-vector multiplications of which one or more of the three size parameters are \(L\). Consequently, increasing \(L\) will provide the BLAS routines with greater opportunity to enhance the data reuse and thereby increase the performance. On the other hand, as we will note below, the computational cost for the second stage grows proportionally with \(L\). Hence there should be an optimal value of \(L\) that minimizes the total computation time.

2.2. Wu’s modification

To further improve the performance of the first stage, Wu [11] proposed to apply Dongarra’s blocking technique [6] to Algorithm 1. The resulting algorithm
is shown as Algorithm 2. In this algorithm, the applications of the block Householder transformations are deferred until \( M \) block Householder transformations are ready, at which time they are applied as an aggregated block Householder transformation. This incurs a little extra work in the matrix-block vector multiplication (computation of \( P^{(K)} \)), but makes it possible to replace the rank-2L update in Algorithm 1 with rank-2ML update, further increasing the opportunity for data reuse. So there should be some optimal value of \( M \) that minimizes the computation time of the first stage. Note that the output of the algorithm does not change due to Wu’s modification, so the cost of the second is independent of \( M \).

Algorithm 2: Bischof & Wu’s algorithm

```latex
\begin{algorithm}
\begin{algorithmic}
\State \textbf{do } \( K = 1 \), \( N/(M L) - 1 \)
\State \( U^{((K-1) + M L)} = \phi \), \( Q^{((K-1) + M L)} = \phi \)
\State \textbf{do } \( K = (K - 1) \times M + 1, K \times M \) \hfill (Partial Householder transformation)
\State \( D^{(K)} := D^{(K)} \)
\State \( U^{(K-1)}(Q^{(K-1)\dagger})_{K-\times(K-1) + M} \)
\State \( Q^{(K-1)}(U^{(K-1)\dagger})_{K-\times(K-1) + M} \)
\State Compute the block Householder transformation \( H^{(K)} = I - U^{(K)}Q^{(K)}U^{(K)\dagger} \).
\State \textbf{[Matrix-block vector multiplication]}
\State \( P^{(K)} := (C^{(K)} - U^{(K-1)}Q^{(K-1)\dagger})U^{(K)\dagger} \)
\State \( \beta^{(K)} := \alpha^{(K)}U^{(K)\dagger}P^{(K)\dagger} \)/2
\State \( Q^{(K)} := P^{(K)} - \beta^{(K)}U^{(K)} \)
\State \( U^{(K)} := [U^{(K-1)}U^{(K)}] \)
\State \( Q^{(K)} := [Q^{(K-1)}Q^{(K)}] \)
\State \textbf{end do}
\State \textbf{[Rank-2ML update of the matrix]}
\State \( C^{((K+M)\times M)} := C^{((K-1)\times M)} - U^{(K+M)}Q^{(K+M)\dagger} \)
\State \(- Q^{(K+M)\dagger}U^{(K+M)} \)
\State \textbf{end do}
\end{algorithmic}
\end{algorithm}
```

2.3. Reduction to a tridiagonal matrix

To reduce the band matrix to tridiagonal form, we use Rutishauser’s algorithm [12]. Although the standard level-3 BLAS routines cannot be used in this algorithm, its computational cost is \( 6N^2L \) and the computational time is small compared with the first stage as long as \( L \) is moderately small.

2.4. BLAS routines used in the first stage

Implementation of Algorithm 2 requires various level-3 BLAS routines. Let’s assume that all the matrices are stored in the column-by-column order (in BLAS notation, ‘N’ or ‘Non-transposed’). Then, corresponding to the matrix multiplications of types \( AB, AB^t \) and \( A' B \) in the algorithm, three kinds of DGEMM calls with transposition specifications ‘\( N' \)’ ‘\( N' \)’ ‘\( T' \)’ ‘\( N' \)’, respectively, are needed. In addition, the matrix-block vector multiplications and the rank-2ML updates are performed via calls to DSYMM and DSYR2K, respectively. Finally, another routine, which we call BlockHouse hereafter, is called to generate the block Householder transformations. Thus we need to know the performance of these six routines to estimate the performance of Algorithm 2.

3. Performance modeling

3.1. The basic idea

To construct a performance model for Algorithm 2, we adopt a hierarchical approach [2][3][4]. In this approach, we first construct empirical models of BLAS performances that predict the execution time of each BLAS routine as a function of the input parameters such as the matrix sizes and transposition specifications. Then, based on the number of calls to each routine and the parameters for each call, the execution time for each call is estimated using the models and is accumulated to yield the total execution time. This methodology has been applied to the performance modeling of basic linear algebra functions such as the LU and QR decompositions [3][4] and has been shown to give satisfactory results in these cases.

3.2. Modeling the BLAS performance

As mentioned in subsection 2.4, Algorithm 2 requires calls to five level-3 BLAS routines (DSYMM, DSYR2K and three types of DGEMMs) and the routine BlockHouse. We therefore need to model the execution time of these six routines. To illustrate the process of modeling, we use the case of DGEMM with transposition specifications ‘\( N' \)’ ‘\( N' \)’. This routine computes the matrix-matrix product \( C = AB \), where \( A, B \), and \( C \) are \( m \times k, k \times n \) and \( m \times n \) matrices. It is assumed that all the matrices are stored column-by-column. Our objective is to approximate the execution time of this routine as a function \( f_{DGEMM_NN}(m,n,k) \) of \( m, n, \) and \( k \).

From an analysis of Algorithm 2, it can be shown that \( n \) is always equal to \( L \), the block size, and takes moderate values between 1 and one 100 (say), while \( m \) and \( k \) may take larger values up to the matrix size \( N \). Based on this observation, we chose to model the execution time for different values of \( n \) by different func-
tions, because the DGEMM performance when one of the size parameters is small might be quite irregular and not appropriate to approximate by a single function. More specifically, we take \{3, 6, 12, 24, 48, 96\} as representative values of \(n\) and approximate the execution time when \(n\) is fixed to one of these values by a bilinear function of \(m\) and \(k\), namely,

\[
\begin{aligned}
    f_{\text{DGEMM,NN}}(m, n, k) &= f_{\text{DGEMM,NN}}^n(m, k) \\
    &= (a_{11}^n m + a_{10}^n) k + (a_{01}^n m + a_{00}^n).
\end{aligned}
\]  

(1)

The bilinear form was chosen because it can naturally express the fact that the execution time consists of the setup time and the time proportional to the computational work. The superscript \(n\) of the coefficients in eq. (1) means that they are the coefficients for the fixed value of \(n\). They are determined by the least squares from the execution time measured at some grid points in the \((m, k)\) plane. Since the function \(f_{\text{DGEMM,NN}}\) is specified for only 6 values of \(n\), we define the function values for other \(n\) by linear interpolation. The execution times of DGEMMs with transposition specifications ‘N’ ‘T’ and ‘T’ ‘N’ were modeled in the same manner. The modeling of DSYMM and DSYR2K is similar but easier because they have only two size parameters.

### 3.3. Modeling the performance of BlockHouse

The routine BlockHouse has two size parameters, namely, the block size \(L\) and the length of the block vector, \(n\). In this case, modeling by linear or bilinear function does not give good results because the routine cannot be written with the level-3 BLAS and the performance will degrade for large \(L\) or \(n\) due to cache miss. Note that for an optimized level-3 BLAS routine, the effect of cache miss is negligible and we can safely assume that its performance approaches a constant as the matrix size increases. This is the reason we can use the bilinear function in the modeling in the previous subsection. In the case of BlockHouse, instead, we chose to measure the execution time at some grid points in the \((L, n)\) plane and interpolate the results by bilinear functions.

### 3.4. Modeling the performance of Algorithm 2

Having constructed the performance of the BLAS primitives, we are now in a position to build a performance model for the entire algorithm. The conventional approach to this has been to derive an analytical expression of the computational work performed by each BLAS routines and then use the BLAS performance model to calculate the total time consumed by each routine [2][3][4]. However, this approach takes considerable effort for model building if one tries to obtain an accurate model that takes into account all the lower order terms in \(N, L\) and \(M\).

Instead, we take an extremely simple approach. We first write functions such as Dgemm\_TIME, Dsymm\_TIME, Dsyrm2\_TIME and BlockHouse\_TIME. These functions have the same set of arguments as the original routines, but these arguments are dummy except for the size parameters and transposition specifications. Instead of doing actual computations, these functions estimate using the performance models the execution time of the corresponding routines given the set of input size parameters and return the estimated time. We next rewrite the program of Algorithm 2 so that the calls to the BLAS routines are replaced with the calls to the corresponding timing routines and the estimated execution time is accumulated at each call. Thus the original program is transformed into a program for estimating its execution time given the values of \(N, L\) and \(M\).

With this approach, an accurate model that takes into account all the lower order terms of the size parameters can be constructed with little effort. The cost of estimating the execution time for one set of \((N, L, M)\) is \(O(N/L)\), as can be easily seen from Algorithm 2, and is negligible compared with \(\frac{A}{N^3}\), the computational cost of the original program.

### 4. Experimental results

#### 4.1. Performance models of BLAS routines

To validate the effectiveness of our model, we performed experiments on two platforms, namely, a 1.6GHz Opteron processor with 2GBytes of memory and a 750MHz Alpha 21264A processor with 512MBytes of memory. In both cases, we used gcc FORTRAN compiler with optimization level -O4 and the GOTO BLAS. The subroutine BlockHouse was handwritten using FORTRAN and no particular optimization was done.

We first constructed the performance models of BLAS routines following the procedure described in subsection 3.2. To construct a model for DGEMM with transposition specifications ‘N’ ‘N’, we set \(n\), \(m\) and \(k\) to one of the values of \{3, 6, 12, 24, 48, 96\}, \{3, 6, 12, 24, 48, 96\} and \{100, 200, 400, 800, 1000\}, respec-
tively, and performed $6 \times 6 \times 5 = 180$ measurements. Then, for each value of $n$, we approximated the execution time by a bilinear function of $m$ and $k$ using the least squares and obtained the coefficients $a_{11}^n$, $a_{10}^n$, $a_{01}^n$ and $a_{00}^n$ in eq. (1). These coefficients for the Opteron processor are given in Table 1. In Table 2, we show the comparison of the actual execution times and the times estimated by the model (both in seconds) for the case of $n = 12$. It can be seen that they agree well, especially when $k$ is large. The construction of models for other BLAS routines and BlockHouse is similar, so we omit them due to the limitation of space.

4.2. Prediction of the execution time

Next we built a model for Algorithm 2 following the methodology stated in subsection 3.4 and compared the execution time predicted by the model with the actual execution time for various values of $N$, $L$ and $M$. The matrix size $N$ was set to 1920, 3840 or 7680 (Opteron only), while $L$ and $M$ were set to one of the values of \{3, 6, 12, 24, 48, 96\} and \{1, 2, 4, 8, 16, 32, 64\}, respectively, with the restriction that $M \times L \leq 384$.

Table 1. The coefficients of eq. (1) for DGEMM('N','N') on the Opteron.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$a_{11}^n$</th>
<th>$a_{10}^n$</th>
<th>$a_{01}^n$</th>
<th>$a_{00}^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8.75E-9</td>
<td>5.54E-8</td>
<td>-1.54E-9</td>
<td>2.21E-6</td>
</tr>
<tr>
<td>6</td>
<td>9.83E-9</td>
<td>7.99E-8</td>
<td>1.08E-8</td>
<td>3.18E-6</td>
</tr>
<tr>
<td>12</td>
<td>1.19E-8</td>
<td>1.73E-7</td>
<td>3.63E-8</td>
<td>4.92E-6</td>
</tr>
<tr>
<td>24</td>
<td>1.98E-8</td>
<td>2.14E-7</td>
<td>7.64E-8</td>
<td>8.33E-6</td>
</tr>
<tr>
<td>48</td>
<td>3.55E-8</td>
<td>3.33E-7</td>
<td>1.57E-7</td>
<td>1.60E-5</td>
</tr>
<tr>
<td>96</td>
<td>6.73E-8</td>
<td>4.77E-7</td>
<td>3.11E-7</td>
<td>3.29E-5</td>
</tr>
</tbody>
</table>

The results on the Opteron with $N = 1920$ are shown in Table 3, where the upper figure for each $(L, M)$ pair denotes the actual time and the lower figure the estimated time (both in seconds). We can see that these figures agree with the maximum relative error of 10%. The results are also graphically shown in Figures 2 and 3. From the graphs, it is clear that the model not only predict the execution time for each pair of $(L, M)$, but also reproduce the variation of the execution time as $L$ and $M$ change. Thus we can expect to choose the optimal value of these block sizes using this model. We obtained similar results for other values of $N$. In particular, when $N = 7680$, the model gave extremely accurate results and the maximum relative error of the predicted execution time was below 2%.

The results on the Alpha 21264A with $N = 1920$ are shown in Table 4. The predicted and actual times
agree within 4% error in this case. When \( N = 3840 \), the maximum error was 5%. From these results, we know that our model gives accurate prediction on both platforms.

4.3. Selection of the optimal block sizes

Now we consider the problem of determining the block sizes \( L \) and \( M \) that minimize the total execution time of the tridiagonalization routine. To this end, we estimate the total time as the sum of the predicted time of stage 1 (transformation to a band matrix) and the actual execution time of stage 2 (reduction of the band matrix to tridiagonal form). Although it is possible to estimate the latter with the model, we chose this approach because the execution time of stage 2 is much shorter than that of stage 1 and also there are only two parameters, \( N \) and \( L \), to vary. This greatly reduces the cost of performance measurement.

Tables 5 and 6 show the execution time of stage 2 for \( N = 1920 \) on the Opteron and Alpha 21264A processors, respectively. From Tables 3 and 5, we can predict that on the Opteron processor, the optimal combination for \( N = 1920 \) is \((L, M) = (24, 2)\). From the same tables, we know that the combination that actually minimize the total time is also \((L, M) = (24, 2)\). Thus the model made the right choice in this case. For the 21264A, the model also chose the combination \((24, 2)\), which is actually the choice that minimizes the total execution time.

The block sizes chosen by the model, the actual execution time corresponding to them, the optimal block sizes, and the corresponding execution time for \( N = 1920 \), 3840 and 7680 are summarized in Tables 7 (Opteron) and 8 (Alpha 21264A). It can be seen from the tables that the block sizes selected by the model are optimal in most cases and even when this is not the case, the execution time corresponding to them is at most 1% longer than the optimal execution time. Thus our model proved useful in finding the optimal block sizes.

5. Related works

In this section, we compare our study with existing works on the performance modeling and automatic tuning of linear algebra programs.

Katagiri et al. [8] propose I-LIB, which is an automatically tuned linear algebra library for distributed-memory parallel machines. They report the performance results of optimizing many parameters such as unrolling sizes and the choice of interprocessor communication routines in the tridiagonalization routine and show that substantial speedup is possible on both the SR2201 and SR8000. However, they don't adopt the hierarchical approach and measure the time of the entire routine for each combination of the parameters. This is apparently too costly.

The hierarchical approach we adopt here was first proposed by Dackland et al [4]. Using this approach, Cuenca et al. [2][3] construct an effective framework for an automatically tuned linear algebra library called SOLAR. However, as far as we know, the application of the hierarchical approach has been limited to basic
Table 4. Actual (above) and estimated (below) execution times (s) of reduction to a band matrix (\textit{Alpha}, \(N = 1920\)).

<table>
<thead>
<tr>
<th>(L)</th>
<th>3</th>
<th>6</th>
<th>12</th>
<th>24</th>
<th>48</th>
<th>96</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>42.01</td>
<td>23.04</td>
<td>14.27</td>
<td>11.02</td>
<td>9.98</td>
<td>10.55</td>
</tr>
<tr>
<td></td>
<td>41.59</td>
<td>23.81</td>
<td>14.54</td>
<td>11.11</td>
<td>9.95</td>
<td>10.31</td>
</tr>
<tr>
<td></td>
<td>37.27</td>
<td>20.83</td>
<td>13.55</td>
<td>10.86</td>
<td>10.08</td>
<td>11.02</td>
</tr>
<tr>
<td></td>
<td>38.41</td>
<td>21.42</td>
<td>13.81</td>
<td>11.03</td>
<td>10.16</td>
<td>10.90</td>
</tr>
<tr>
<td></td>
<td>35.26</td>
<td>20.16</td>
<td>13.42</td>
<td>11.04</td>
<td>10.60</td>
<td>11.96</td>
</tr>
<tr>
<td></td>
<td>36.09</td>
<td>20.76</td>
<td>13.77</td>
<td>11.28</td>
<td>10.79</td>
<td>11.76</td>
</tr>
<tr>
<td></td>
<td>34.66</td>
<td>20.24</td>
<td>13.68</td>
<td>11.70</td>
<td>11.71</td>
<td></td>
</tr>
<tr>
<td></td>
<td>35.63</td>
<td>20.91</td>
<td>14.15</td>
<td>12.04</td>
<td>11.78</td>
<td></td>
</tr>
<tr>
<td></td>
<td>35.02</td>
<td>20.81</td>
<td>14.63</td>
<td>13.04</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>36.23</td>
<td>21.72</td>
<td>15.21</td>
<td>13.32</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>36.36</td>
<td>22.67</td>
<td>16.59</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>37.93</td>
<td>23.61</td>
<td>17.06</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>39.94</td>
<td>26.36</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>41.56</td>
<td>27.05</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5. Execution time (s) of tridiagonalization of a band matrix (\textit{Opteron}, \(N = 1920\)).

<table>
<thead>
<tr>
<th>(L)</th>
<th>3</th>
<th>6</th>
<th>12</th>
<th>24</th>
<th>48</th>
<th>96</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>0.31</td>
<td>0.42</td>
<td>0.69</td>
<td>1.08</td>
<td>2.16</td>
<td>4.93</td>
</tr>
</tbody>
</table>

decompositions such as the LU and QR decompositions [4][3] and relatively simple algorithms such as the Jacobi method [3]. We applied this approach to level-3 BLAS based tridiagonalization, a relatively complicated algorithm with calls to five different kinds of level-3 BLAS routines and a routine for generating a block Householder transformation, and showed that it works well in this case as well. As another point, the method we use here estimates the execution time by simulating the actual computing process instead of approximating it as a polynomial in the problem size and the block sizes, and therefore allows more accurate prediction to be obtained.

Dongarra et al. [5] propose a framework for self-adapting numerical software called SANS. It aims at automatically choosing the best library routine for solving a given problem based on the properties of the problem and available computational resources. Thus the main emphasis of SANS is on the optimal selection at a higher (between algorithms or libraries) level, while the target of our study is optimization at a mid (within algorithm) level. The two approaches are therefore complementary and our model can be used in SANS framework to predict and optimize the performance of the level-3 BLAS based tridiagonalization algorithm.

There are many efforts that focus on modeling and optimizing the performance of basic linear algebra routines such as the BLAS for dense matrices [10] and matrix vector multiplication for sparse matrices [9]. We can say that these works also deal with problems that lie at a different level from ours. In fact, the tridiagonalization algorithm treated here uses these routines as its components and benefits much from their optimized performance.

6. Conclusion

In this paper, we proposed a performance model for Bischof & Wu’s algorithm for tridiagonalizing a symmetric matrix. The model has a hierarchical structure, which reflects the hierarchical structure of the algorithm, and given the problem size, the two block sizes and the performance models of the underlying BLAS routines, predicts the execution time accurately.

The numerical experiments show that our model can
predict the execution time with the maximum relative error of 10% on the Opteron and Alpha 21264A processors for various problem and block sizes. Based on these results, we succeeded in finding the optimal or near-optimal block sizes in each case.

Future works include extension of this model to parallel tridiagonalization algorithms for shared memory and distributed memory parallel machines and development of an efficient search algorithm that can find near-optimal block sizes without an exhaustive search of all possible candidates.

Acknowledgements I would like to thank Professor Takahiro Katagiri, Professor Toshitsugu Yuba, Professor Reiji Suda, Professor Toshiyuki Imamura and Mr. Ken Naono for fruitful discussion. I am also grateful to Professor Masaaki Sugihara and Professor Shao-Liang Zhang for continuous support. This work was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for young Scientists, 16700053, 2004 and Grant-in-Aid for the 21st Century COE "Frontiers of Computational Science".

References