5-2003

# A Coarse-Grain Parallel Implementation of the Block Tridiagonal Divide and Conquer Algorithm for Symmetric Eigenproblems.

Robert M. Day
*University of Tennessee - Knoxville*

To the Graduate Council:

I am submitting herewith a thesis written by Robert M. Day entitled "A Coarse-Grain Parallel Implementation of the Block Tridiagonal Divide and Conquer Algorithm for Symmetric Eigenproblems.." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Robert C. Ward, Major Professor

We have read this thesis and recommend its acceptance:

Michael W. Berry, Jian Huang

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a thesis written by Robert M. Day entitled “A Coarse-Grain Parallel Implementation of the Block Tridiagonal Divide and Conquer Algorithm for Symmetric Eigenproblems.” I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Robert C. Ward

Major Professor

We have read this thesis
and recommend its acceptance:

Michael W. Berry

Jian Huang

Acceptance for the Council:

Anne Mayhew

Vice Provost and Dean of
Graduate Studies

(Original signatures are on file with official student records.)

# A Coarse-Grain Parallel Implementation of the Block Tridiagonal Divide and Conquer Algorithm for Symmetric Eigenproblems

A Thesis
Presented for the
Master of Science Degree
The University of Tennessee, Knoxville

Robert Michael Day
May 2003

## Dedication

This thesis is dedicated first and foremost to my wife Sunhee and daughter Ashley for their love and support. Also to my parents Michael and Ethel Day, for their love, guidance, and support. And of course my second parents, Mr. and Mrs. Youngkil Im.

Finally, I give thanks to God for the many, many blessings in my life.

## Acknowledgments

I wish to thank all those who helped and guided me in the completion of my Master of Science in Computer Science. My first thanks goes to Dr. Ward for his guidance in this project and help in the preparation of this thesis. I thank Dr. Berry and Dr. Huang for serving on my committee. I would also like to thank Dr. Wilfried Gansterer, Research Associate, University of Tennessee, for the many answered questions about Divide and Conquer and help with Fortran and LaTeX, and Richard P. Muller, Director, Quantum Simulations, Materials and Process Simulation Center, Beckman Institute, California Institute of Technology 139-74, Pasadena, CA 91125, for providing test cases and helpful discussions.

This work was partially supported by the Academic Strategic Alliances Program of the Accelerated Strategic Computing Initiative (ASCI/ASAP) under subcontract number B341492 of DOE contract W-7405-ENG-48.

Additionally, this research uses resources of the Scalable Intracampus Research Grid (SInRG) Project at the University of Tennessee, supported by the National Science Foundation CISE Research Infrastructure Award EIA-9972889.

## Abstract

Cuppen's divide and conquer technique for symmetric tridiagonal eigenproblems, along with Gu and Eisenstat's modification for improvement of the eigenvector computation, has yielded a stable, efficient, and widely-used algorithm. This algorithm has now been extended to a larger class of matrices, namely symmetric block tridiagonal eigenproblems. The Block Tridiagonal Divide and Conquer algorithm has shown several characteristics that make it suitable for a number of applications, such as the Self-Consistent-Field procedure in quantum chemistry.

This thesis discusses the steps taken to implement a coarse-grain parallel version of the Block Tridiagonal Divide and Conquer algorithm, suitable for a parallel supercomputer or a cluster of machines. The parallel version relies on components of the ScaLAPACK parallel linear algebra library and follows the same model as the serial code, including the implementation of full deflation.

A modest speedup (2 to 3) was achieved using a few processors (4 and 16). Increasing the number of processors from 4 to 16 produced only slightly better speedup. This implementation was not competitive with the standard ScaLAPACK symmetric eigenvalue routine. Analysis shows that the distribution scheme chosen for the eigenvector storage requires $n \times O(p^2)$ function calls to the ScaLAPACK matrix multiplication routine, where $n$ is the matrix size and $p$ is the number of blocks. The matrix multiplications are responsible for the majority of the computational cost; therefore, the associated overhead needs to be reduced in order to make this implementation more competitive.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

In the book *Matrix Computations* [6], Golub and Van Loan state that the symmetric eigenvalue problem

> with its rich mathematical structure is one of the most aesthetically pleasing problems in numerical linear algebra.

The symmetric eigenvalue problem arises in many areas of mathematics, science, and engineering. Several methods have been developed and improved through the years for computing the eigenvalues and eigenvectors of symmetric matrices. Over the last twenty years, Cuppen's divide-and-conquer algorithm for the tridiagonal symmetric eigenproblem [3] has become widely used and, with Gu and Eisenstat's modification [7] for improved eigenvector computation, has become the most efficient and stable method for such problems. Any general symmetric matrix can be reduced to tridiagonal form and then decomposed by divide-and-conquer. This algorithm has been implemented in both serial [8] and parallel [10] versions.

More recently, the divide-and-conquer paradigm has been extended from tridiagonal to block tridiagonal symmetric matrices [5]. This new algorithm has shown several advantages over other symmetric eigensolvers, which make it more suitable for some applications. However, a parallel version has not yet been implemented. The goal of this thesis is to develop a coarse-grained parallel implementation of the Block Tridiagonal Divide-and-Conquer algorithm, and discuss the implementation issues and resulting performance.

## 1.1 Motivation

In quantum chemistry, the *Self-Consistent-Field*, or SCF, procedure is a computational method used to obtain certain values and quantities of interest associated with a given material [9]. It is an iterative procedure that computes the coefficients of a basis function expansion of the molecular orbitals in the composite of the material. One of the steps within an iteration is to diagonalize a real, symmetric matrix by computing its eigenvalues and eigenvectors. This step is also one of the most computationally expensive. Reducing the time for diagonalization will significantly reduce the overall time for SCF.

Several aspects of the SCF procedure make the Block Tridiagonal Divide and Conquer algorithm an attractive choice for diagonalizing the matrix. For one, the matrix has the characteristic that the magnitude of the elements decreases significantly the farther they are from the main diagonal. The matrix can therefore be closely approximated by a block tridiagonal matrix. Another aspect is that the procedure does not require a high accuracy eigen-decomposition for the early iterations. As will be discussed later, this algorithm optionally allows the eigenvalues and eigenvectors to be computed at a lower accuracy, with a reduction in the required computational time.

In order for computational chemists to solve larger, more complex problems, the time for the computation of the eigen-decomposition must be reduced significantly. One way to do this is by parallelizing the algorithm.

## 1.2 Objectives

The main objectives of this thesis are to improve computational performance of the Block Tridiagonal Divide and Conquer algorithm by modifying the algorithm to utilize parallel processing, and to investigate the resulting implementation issues. The algorithm will be implemented in a way suitable for distributed memory parallel architectures, such as a cluster of workstations or a parallel supercomputer. The goal is to be able to compute the eigen-decomposition of larger matrices that could not be contained in the main memory of a single workstation or that would otherwise require more time to compute serially than is desired.

## 1.3 Overview of Thesis

The next chapter will discuss the main details of the block tridiagonal divide-and-conquer algorithm. Special emphasis will be given to those aspects that have the most effect on parallelization. In Chapter 3, working details of the parallelization of the algorithm will be shown, along with the issues that arose and how they were addressed. Timings and other results from experimentation, along with analysis, will be given in Chapter 4. Finally, Chapter 5 will conclude the paper and discuss possible future work in relation to this thesis.

## 1.4 Definitions and Notations

This section discusses several notations and definitions used throughout this thesis. When stating that A and B are equal,

$$A = B$$

is used. When defining A to be equal to B,

$$A := B$$

is used.

The set of real numbers is denoted by $\mathbb{R}$.

A *matrix* $M \in \mathbb{R}^{m \times n}$ is an array of real numbers with $m$ rows and $n$ columns (usually called an $m \times n$ matrix). The element in the $i^{th}$ row and $j^{th}$ column of $M$ is referenced by the lower-case of the matrix name $m_{i,j}$ or by $[M]_{i,j}$.

The *transpose* of a matrix $M \in \mathbb{R}^{m \times n}$ is an $n \times m$ matrix denoted by $M^T$, and is defined by

$$[M^T]_{i,j} := [M]_{j,i} \quad \text{for} \quad i = 1, 2, \ldots, n \quad \text{and} \quad j = 1, 2, \ldots, m.$$

A matrix $M$ is said to be *symmetric* if and only if it is equal to its transpose, i.e.

$$M = M^T, \quad \text{or}$$

$$[M]_{i,j} = [M^T]_{i,j} \quad \text{for} \quad i = 1, 2, \ldots, m, \quad \text{and} \quad j = 1, 2, \ldots, n.$$

Other matrix operations, such as multiplication, are described in [6, §1.1].

The main algorithm discussed in this thesis (the Block Tridiagonal Divide and Conquer algorithm) will often be referred to as the BD&C algorithm.

# Chapter 2

# Divide and Conquer Algorithm for Symmetric Block Tridiagonal Matrices

## 2.1 Concept

Cuppen's popular divide and conquer algorithm for symmetric tridiagonal matrices [3] has been extended to a larger class of matrices [5], namely symmetric block tridiagonal matrices.

Let $T \in \mathbb{R}^{n\times n}$ be a symmetric block tridiagonal matrix,

$$T := \begin{pmatrix} B_1 & C_1^T & & & \\ C_1 & B_2 & C_2^T & & \\ & C_2 & B_3 & \ddots & \\ & & \ddots & \ddots & C_{p-1}^T \\ & & & C_{p-1} & B_p \end{pmatrix} \tag{2.1}$$

where $p > 1$, $B_i \in \mathbb{R}^{k_i\times k_i}$ is symmetric for $i = 1,2,\ldots p$, and $C_i \in \mathbb{R}^{k_{i+1}\times k_i}$ is arbitrary for $i = 1,2,\ldots p-1$. The block sizes $k_i$ must satisfy $1 \le k_i < n$ and $\sum_{i=1}^{p} k_i = n$.

The goal is to find the spectral decomposition

$$T = QDQ^T$$

where $Q \in \mathbb{R}^{n \times n}$ is the orthogonal eigenvector matrix, and $D \in \mathbb{R}^{n \times n}$ is a diagonal matrix containing the corresponding eigenvalues. The symmetry of $T$ guarantees that all eigenvalues are real and the eigenvector matrix is orthogonal [6, §8.1.1].

## 2.2 The Divide and Conquer Algorithm

The eigenvalues and eigenvectors of the matrix $T$ can be computed in a divide and conquer fashion, described in the following sections.

### 2.2.1 Subdivision

First, the singular value decomposition, or SVD [6, §2.5.3], of each subdiagonal block is computed:

$$C_i = U_i \Sigma_i V_i^T \tag{2.2}$$

Each matrix $\Sigma_i \in \mathbb{R}^{k_{i+1} \times k_i}$ for $i = 1, 2, \ldots p - 1$ is a diagonal matrix containing the singular values of $C_i$, and $U_i \in \mathbb{R}^{k_{i+1} \times k_{i+1}}$ and $V_i \in \mathbb{R}^{k_i \times k_i}$ contain (as columns) the left and right singular vectors, respectively. $C_i$ can then be written as a sum of rank-1 matrices,

$$C_i = \sum_{j=1}^{r_i} u_j \sigma_j v_j = \sum_{j=1}^{r_i} \sigma_j u_j v_j,$$

where $r_i = \mathrm{rank}(C_i)$ The diagonal blocks are modified using the SVDs of the subdiagonal blocks as follows:

$$\begin{aligned}\tilde{B}_1 &:= B_1 - V_1\Sigma_1 V_1^T \\ \tilde{B}_i &:= B_i - V_i\Sigma_i V_i^T - U_{i-1}\Sigma_{i-1}U_{i-1}^T \quad \text{for} \quad i = 2,3,\dots p-1 \\ \tilde{B}_p &:= B_p - U_{p-1}\Sigma_{p-1}U_{p-1}^T\end{aligned}$$

Then let $\tilde{T}$ be a block diagonal matrix containing the modified diagonal blocks from $T$,

$$\tilde{T} := \begin{pmatrix} \tilde{B}_1 & & & \\ & \tilde{B}_2 & & \\ & & \ddots & \\ & & & \tilde{B}_p \end{pmatrix}$$

Now we have

$$\begin{aligned} T &= \begin{pmatrix} B_1 & V_1\Sigma_1 U_1^T & & 0 \\ U_1\Sigma_1 V_1^T & B_2 & \ddots & \\ & \ddots & \ddots & V_{p-1}\Sigma_{p-1}U_{p-1}^T \\ 0 & & U_{p-1}\Sigma_{p-1}V_{p-1}^T & B_p \end{pmatrix} \\ &= \tilde{T} + \begin{pmatrix} V_1\Sigma_1 V_1^T & V_1\Sigma_1 U_1^T & 0 & \cdots \\ U_1\Sigma_1 V_1^T & U_1\Sigma_1 U_1^T & 0 & \cdots \\ 0 & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} + \cdots \\ &\qquad \cdots + \begin{pmatrix} \ddots & \vdots & \vdots & \vdots \\ \cdots & 0 & 0 & 0 \\ \cdots & 0 & V_{p-1}\Sigma_{p-1}V_{p-1}^T & V_{p-1}\Sigma_{p-1}U_{p-1}^T \\ \cdots & 0 & U_{p-1}\Sigma_{p-1}V_{p-1}^T & U_{p-1}\Sigma_{p-1}U_{p-1}^T \end{pmatrix}\end{aligned}$$

The sum involving the sparse block matrices can be written more simply as

$$\begin{aligned} T &= \tilde{T} + W_1W_1^T + \cdots + W_{p-1}W_{p-1}^T \\ &= \tilde{T} + \sum_{i=1}^{p-1} W_iW_i^T \end{aligned}$$

where

$$W_1 := \begin{pmatrix} V_1\Sigma_1^{1/2} \\ U_1\Sigma_1^{1/2} \\ 0 \\ \vdots \\ 0 \end{pmatrix}, W_2 := \begin{pmatrix} 0 \\ V_2\Sigma_2^{1/2} \\ U_2\Sigma_2^{1/2} \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \ldots W_{p-1} := \begin{pmatrix} 0 \\ \vdots \\ 0 \\ V_{p-1}\Sigma_{p-1}^{1/2} \\ U_{p-1}\Sigma_{p-1}^{1/2} \end{pmatrix}$$

So the original matrix $T$ can be written as

$$T = \tilde{T} + \sum_{i=1}^{p-1} W_iW_i^T \tag{2.3}$$

### 2.2.2 Solution of Subproblems

The next step of the algorithm is to find the spectral decompositions of the modified diagonal blocks $\tilde{B}_i$. Let $\tilde{B}_i = \tilde{Q}_i\tilde{D}_i\tilde{Q}_i^T$ be the decompositions for $i = 1, 2, \ldots p$ , and let

$$\tilde{D} := \begin{pmatrix} \tilde{D}_1 & & & \\ & \tilde{D}_2 & & \\ & & \ddots & \\ & & & \tilde{D}_p \end{pmatrix} \quad \text{and} \quad \tilde{Q} := \begin{pmatrix} \tilde{Q}_1 & & & \\ & \tilde{Q}_2 & & \\ & & \ddots & \\ & & & \tilde{Q}_p \end{pmatrix},$$

so that $\tilde{T} = \tilde{Q}\tilde{D}\tilde{Q}^T$. Note that $\tilde{D}$ is a diagonal real matrix and $\tilde{Q}$ is a block diagonal real matrix. Then from (2.3),

$$\begin{aligned} T &= \tilde{T} + \sum_{i=1}^{p-1} W_i W_i^T \\ &= \tilde{Q}\tilde{D}\tilde{Q}^T + \sum_{i=1}^{p-1} W_i W_i^T \\ &= \tilde{Q}\left(\tilde{D} + \sum_{i=1}^{p-1} \tilde{Q}^T W_i W_i^T \tilde{Q}\right)\tilde{Q}^T \\ &= \tilde{Q}\left(\tilde{D} + \sum_{i=1}^{p-1} Y_i Y_i^T\right)\tilde{Q}^T \end{aligned}$$

where $Y_i := \tilde{Q}^T W_i$.

### 2.2.3 Synthesis of Subproblems

The matrix

$$S := \tilde{D} + \sum_{i=1}^{p-1} Y_i Y_i^T \tag{2.4}$$

is called the *synthesis matrix*. The matrix $T$ is orthogonally similar to the synthesis matrix, and therefore has the same eigenvalues. The eigenvector matrix of $T$ can be found by multiplying $\tilde{Q}$ by the eigenvector matrix of $S$, as follows. Assume that the eigen-decomposition of the synthesis matrix is $S = ZDZ^T$. Then

$$\begin{aligned} T &= \tilde{Q} Z D Z^T \tilde{Q}^T \qquad (2.5) \\ &= QDQ^T, \end{aligned}$$

where $Q = \tilde{Q}Z$. The problem of finding the eigen-decomposition of $T$ is now reduced to finding the eigen-decomposition of $S$. Fortunately, there are good methods for doing this.

The right-hand side of (2.4) can be viewed as a series of rank-1 updates of $\tilde{D}$, a diagonal matrix. This is important because an accurate, efficient method for computing the eigensystem of a rank-1 updated diagonal matrix already exists, and can be separately applied to each rank-1 modification in (2.4). The mathematics of the method will be discussed in the next section. For each subdiagonal block $C_i$, $r_i$ rank-1 modifications are performed, where $r_i = rank(C_i)$. The computation of all rank-1 modifications corresponding to a single subdiagonal block is referred to as a *merge*, since it combines the eigensystems of the two related diagonal blocks.

### 2.2.4 Eigen-decomposition of the Synthesis Matrix

Given a matrix $D = diag(d_1, d_2, \dots d_n) \in \mathbb{R}^{n\times n}$ such that $d_1 > d_2 > \cdots > d_n$, $\rho \neq 0$, and a vector $z \in \mathbb{R}^n$ such that no element of $z$ is zero, the eigenvalues and eigenvectors of the matrix $(D + \rho z z^T)$ can be found as follows [6, §8.5.3]. The eigenvalues $\{\lambda_i\}_{i=1}^n$ are the roots of the secular equation

$$\begin{aligned} f(\lambda) &:= 1 + \rho z^T (D - \lambda I)^{-1} z \\ &= 1 + \rho \sum_{i=1}^{n} \frac{z_i^2}{d_i - \lambda} \end{aligned} \tag{2.6}$$

The eigenvalues can be computed by an efficient, specialized version of Newton's method [8]. The corresponding eigenvector $v_i$ is given by

$$v_i = (D - \lambda_i I)^{-1} z. \tag{2.7}$$

However, the direct implementation of (2.7) to compute the eigenvectors suffers from numerical instability, due to the matrix $D - \lambda_i I$ being close to singular. Gu and Eisenstat developed an algorithm to compute the eigenvectors which guarantees numerical stability and orthogonality [7].

## 2.3 Reduced Accuracy Ability

One advantage of the design of this algorithm is that it is possible to compute the entire spectrum of eigenvalues at an accuracy much less than machine precision and reducing the time required for these computations. As noted earlier, the SCF procedure is an iterative process in which one step of an iteration involves computing the eigenvalues and eigenvectors of a symmetric matrix. Earlier iterations of the procedure can accept approximations of the eigensystem without significantly affecting the convergence of the solution. Computing these eigen-decompositions at a reduced accuracy can eliminate many unnecessary floating point operations.

A reduced accuracy approximation of the eigensystem can be accomplished in several ways. For one, when computing the SVDs of the subdiagonal blocks (2.2), the rank-1 updates corresponding to singular values less than a certain parameter (directly related to the desired accuracy) can be omitted. This reduces the accuracy of the eigensystem resulting from the merging of the corresponding diagonal blocks. However, this also reduces the floating point operations required for the merging.

Another way to reduce accuracy is in the deflation step of the algorithm.

## 2.4 Deflation

An important aspect of the BD&C algorithm that can significantly reduce computation time is deflation. Deflation allows the algorithm to reduce the size of two subproblems that are being merged. There are two ways for deflation to occur.

Recall that Equation (2.6) requires that the eigenvalues of the diagonal matrix being updated must be ordered and unequal. If there are equal eigenvalues, a Givens rotation can be used to eliminate one of the values and reduce the number of eigenvalues and eigenvectors that need to be computed. Another requirement is that no element of the vector $z$ can be zero. If either of these occur, a permutation is used to move the eigenvalue and corresponding element of $z$ to the end of the array in order to reduce the size of the problem. If such deflations occur, it introduces a special block structure into the eigenvector matrix that allows more efficient computation.

In [5], deflation is extended further by considering “almost equal” elements instead of equal. If two eigenvalues are within a certain tolerance (greater than machine precision but based on the desired accuracy), then they are treated as if they were equal, and a Givens rotation is applied. This tolerance is called a *relaxed tolerance*. If an element of the $z$ vector is within this relaxed tolerance of zero (close to zero), then deflation of the element is performed. Experimentation has shown that relaxing the deflation tolerance significantly reduced floating point operations, and analysis has shown that the entire spectrum of eigenvalues and eigenvectors are found at a specified accuracy greater than machine precision. This often requires many permutations of the eigenvector matrices, which on a serial machine is compensated for by the savings in floating point operations. In the parallel implementation, however, this will be one of the major issues.

# Chapter 3

# Approach and Issues of Parallelization

The goals of this project are to create a coarse-grained parallelization of the Block Tridiagonal Divide and Conquer algorithm, and to investigate the implementation issues. The resulting algorithm will be suitable for distributed memory architectures, such as a parallel cluster environment or parallel supercomputer. The design of the algorithm and issues that had to be overcome are discussed in this chapter.

## 3.1 General Issues in Parallelization

Parallelization is generally utilized to overcome issues in two aspects of computation: data storage and processing time. As the size of a problem becomes large, there is a corresponding increase in the data storage needed and the processing cycles spent. However, it is these large problems that are of most interest to computational scientists. This holds true in problems where the SCF procedure is used, as well. More complex materials are represented by larger matrices, and as a result require more storage space and more processing power.

To overcome the limitations imposed by using a single processor, an obvious approach would be to use multiple processors working together to compute the solution of a problem. In general, these processors may have separate (distributed) memory storage or shared memory. For this project, only distributed memory designs are considered, although standard message-passing to shared-memory methods could be utilized to convert the resulting code to one for a shared-memory environment.

Ideally, the data for the problem and the work required to solve it would be divided among the processors. This distribution of data and work are the main considerations when parallelizing an algorithm. The even distribution of processing work is called *load balance*. It is important that all processors perform approximately the same amount of work on independent tasks, in order to minimize the overall running time of the algorithm. However, another important consideration (which does not arise in serial computing) is communication. In general, the processors may need to communicate for various reasons, such as data sharing or synchronization. Programmers generally try to minimize communication in a parallel algorithm, since it can be a major cost in terms of performance.

Parallel algorithms can generally be divided into two groups (with a gray area in between) according to data dependence. If all processors can work without knowledge of the other processors' work and results, and with very little or no synchronization, then the algorithm is said to be *pleasingly parallel* (formerly termed *embarrassingly parallel*) [4]. In this case, the communication between processors is very low. Unfortunately, eigenvalue problems in general are not pleasingly parallel. This can be seen in the fact that the entire decomposition can be greatly affected by a small change to only one element of the matrix being decomposed, and much synchronization is needed to efficiently divide the workload.

The last aspect of parallel computing that will be discussed in relation to this project is the concept of *fine-grain* versus *coarse-grain* parallelism. An algorithm is fine-grain if the solution to the problem can be broken into a large number of small pieces that are executed simultaneously. Increasing the number of processors allocated to the problem is typically straight-forward, since there are a larger number of tasks that can be divided among the processors. A coarse-grain parallel algorithm has a small number of pieces that cannot be (efficiently) divided further to exploit a large number of processors. This parallel implementation of BD&C is coarse-grained. The reasons will be discussed in Section 3.2.

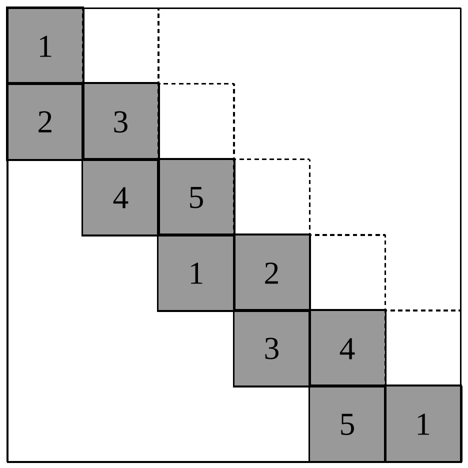

Figure 3.1: A simple distribution of equal size block matrices. Processor 1 stores the first diagonal block, and so on.

## 3.2 Data Distribution

Ideally, data should be distributed as evenly as possible across all processors for good load balance and efficient memory utilization. For this algorithm, the data that must be divided is the diagonal and sub-diagonal blocks of the input matrix and the eigenvectors to be computed. Several methods of distribution were considered. Initially, a simple distribution was considered in which the diagonal blocks and subdiagonal blocks themselves were distributed among the processors in a repeated fashion. Figure 3.1 shows what this distribution might look like for a matrix with six diagonal blocks, and using five processors. (Note that since the matrices being considered are symmetric, the upper diagonal blocks need not be stored, and thus are shown here and in later figures as outlines only.) Recall that the SVD for each subdiagonal block must be computed (§2.2.1), along with the eigen-decomposition of each modified diagonal block (§2.2.2). This distribution would seem to divide the storage and processing work almost evenly. However, this will not be the case for matrices with various, and sometimes widely ranging, block sizes, such as in Figure 3.2. It can be seen that processors 3 and 4 will store more of the data than the other processors, and will have to do more processing to compute the SVDs and eigen-decompositions of their

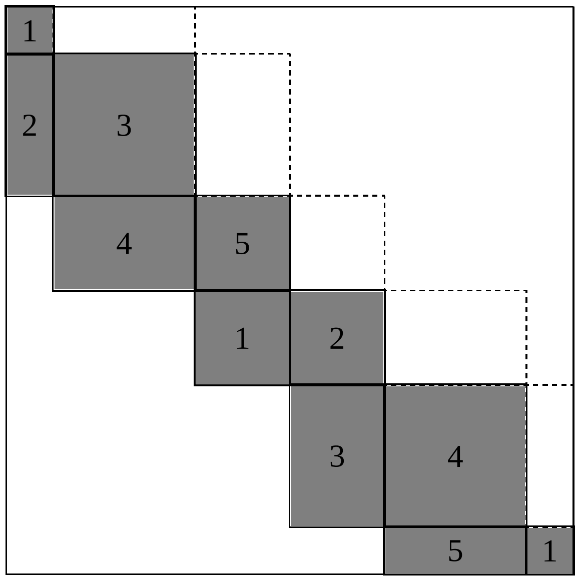

Figure 3.2: An unbalanced distribution of unequal block sizes. The same method of distribution (as Figure 3.1) for a matrix with varying block sizes may result in unbalanced data distribution.

blocks. This method would not generally provide good load balancing, especially since matrices with varying size blocks can occur frequently in practice. A specific example of this is the matrix representing an alkane-160 molecule for the SCF procedure. It can be approximated by a block tridiagonal matrix with block sizes 900, 62, 44, 75, and 853. This poor load balancing would be magnified during the eigenvector computation. Therefore this method of distribution was not used.

Other methods of data distribution that were considered include ones based on an "intelligent" routine to determine the distribution. Instead of a straight-forward, automatic distribution, this routine would consider block sizes and make processor assignments based on how much data had already been assigned to a processor in relation to the other processors. This routine would determine a "good" distribution of data that would (hopefully) balance the workload among the processors. This method, however, could not make use of quality, scalable, reusable software (such as ScaLAPACK), especially during the eigenvector computations, and would require a considerable effort. Since this parallel implementation is a first for the BD&C algorithm, it was decided that a simpler method would be attempted. A future project could be to explore some sort of “intelligent” method of distribution of the data and workload.

Instead, it was decided that a 2-dimensional block cyclic distribution would be used, as in the parallel library ScaLAPACK [2]. The ScaLAPACK library (from *Sca*lable *LAPACK*) is a free, parallel implementation of most of the routines in the LAPACK library [1]. This data distribution is defined as follows: given $p$ processors, an $m \times n$ *process grid* is formed, where $p = m \times n$. A dense matrix is uniformly divided into $k \times k$ sub-blocks, where $k$ is usually based on characteristics of the machine architecture being used (processor cache size, for example). The process grid is then cyclically superimposed over the sub-blocks of the matrix. Figure 3.3 shows an example with 6 processors forming a $2 \times 3$ process grid, and the sub-blocks of the matrix distributed in a block cyclic way.

For the BD&C algorithm, however, the matrix is not a general dense matrix, so the block cyclic distribution is not applied uniformly over the entire matrix. Instead, each diagonal block and subdiagonal block is individually distributed in this way. Figure 3.4 shows an example of how a block tridiagonal matrix is distributed.

Several issues are raised by choosing this type of data distribution. Smaller sub-block sizes tend to require a larger number of communications, therefore increasing communication overhead, while larger sub-block sizes may not make efficient use of memory caching on a single processor. Although the block tridiagonal matrix may be of large size, the number of processors that can be effectively utilized is limited by the size of the diagonal blocks, and the lower bound on sub-block distribution size. These limitations are what classifies this parallel version the of BD&C as coarse-grained.

Also, the matrix dimensions in general are not divisible by the sub-block size $k$, which results in partial sub-blocks on the “right” and the “bottom” of the matrix. However, ScaLAPACK routines are aware of this, and compensate appropriately internally. This distribution of data is known to be storage balancing and load balancing. All routines in

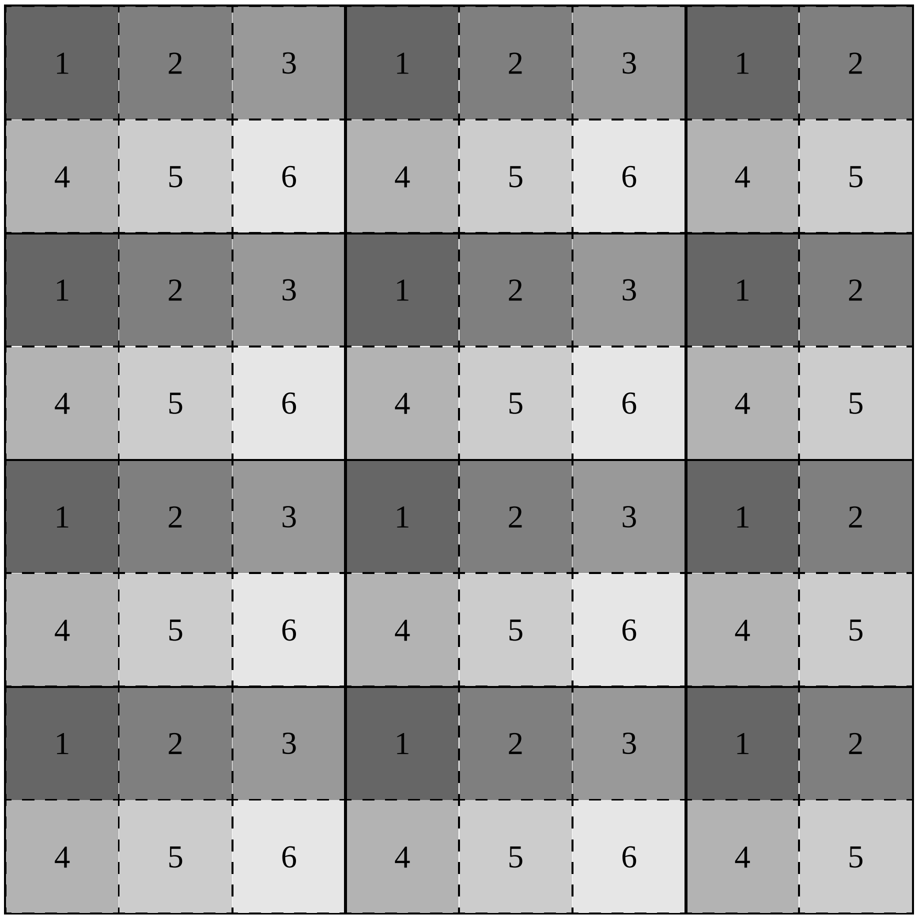

Figure 3.3: 2-dimensional block cyclic distribution of a matrix. In this example, all sub-blocks of this general dense matrix that are labeled with 1 are stored in processor 1, etc.

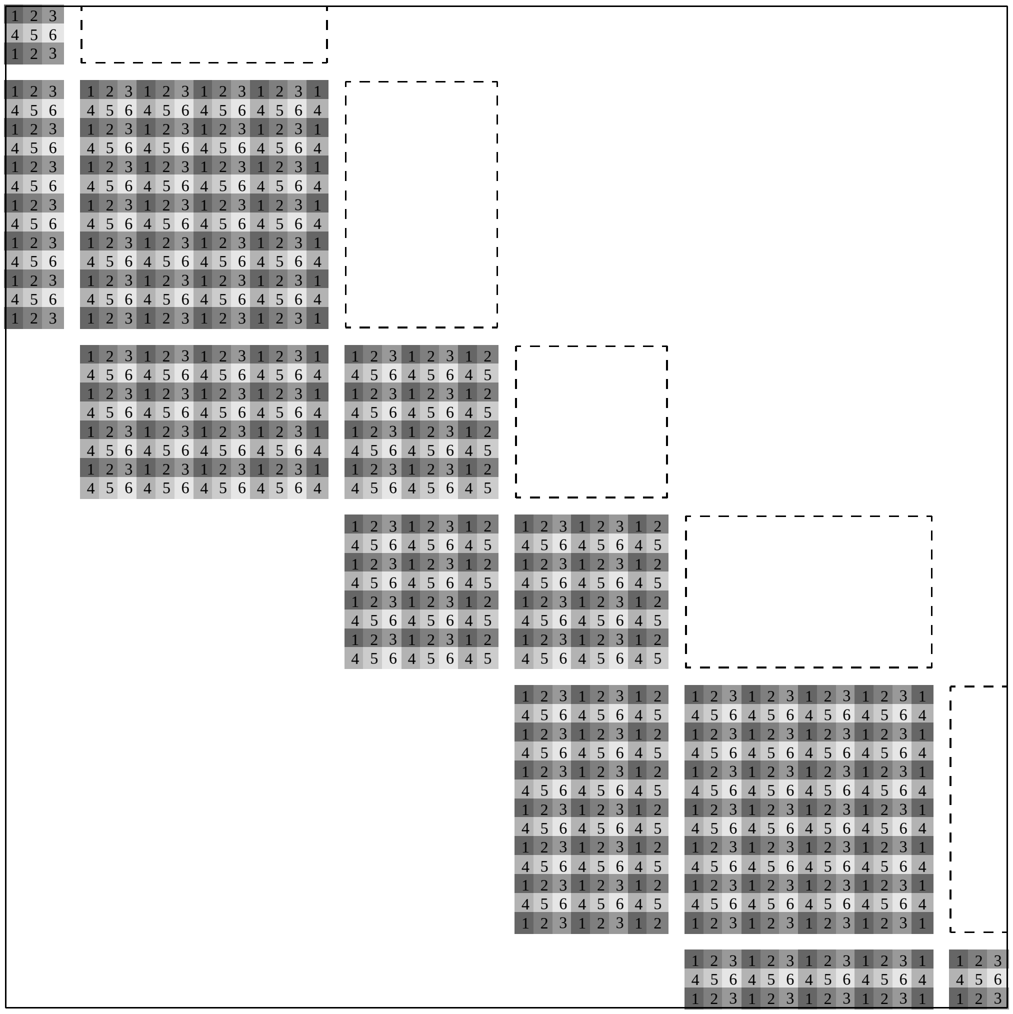

Figure 3.4: Individual block cyclic distribution of the blocks of a block tridiagonal matrix. Each diagonal block and subdiagonal block of a block tridiagonal matrix is individually distributed in the ScaLAPACK block cyclic fashion.

the ScaLAPACK library accept matrices that are stored in this format.

## 3.3 Processor Load Balancing

The block cyclic distribution of Section 3.2 solves the load balancing issues in two areas of the BD&C algorithm. Since the diagonal blocks and subdiagonal blocks of the matrix are distributed by this method, the ScaLAPACK routines for SVD and eigen-decomposition can be utilized for these aspects of the algorithm. In general, ScaLAPACK routines are well balanced, so this parallel implementation should benefit by using them.

Other areas of the algorithm that can be parallelized are the eigenvalue and eigenvector computation of the rank-1 updated matrix described in Section 2.2.3. All processes need to receive the newly computed eigenvalues. For the eigenvalues, a specialized, efficient root solver is used, so dividing up the work and then communicating the results will be more costly than having each processor individually compute all of the eigenvalues. This is also the preference in the parallel implementation of the Tridiagonal Divide and Conquer algorithm [10].

The computation of the eigenvectors can be parallelized very effectively. The routine that computes the eigenvalues also returns parameters used in the forming of the eigenvector corresponding to that eigenvalue. Each eigenvector, however, is stored as a column of the eigenvector matrix, and therefore only the $m$ processors in that process column (of the $m \times n$ process grid) that "own" the eigenvector need to retain those parameters. Each of those processors then works individually to compute its part of the eigenvector. Some communication between processors in that process column must take place because of the permutations performed as a result of deflation (§2.4). This communication is linear with respect to the number of eigenvalues. Meanwhile, other processors can simultaneously work on their eigenvectors.

## 3.4 Eigenvector Storage and Computation

The distributed storage scheme that has been chosen brings up a new issue. Since the diagonal and subdiagonal blocks are individually distributed, the eigenvector matrix that stores the initial and accumulating eigenvectors cannot be distributed in its entirety. The reason is that eigenvector results from a distributed eigen-decomposition cannot be written to an arbitrary $(i, j)^{th}$ element of a distributed matrix. The partitions of the block structure of the initial matrix do not generally coincide with the partition of the matrix imposed by the block cyclic distribution, so complex communication or redistribution would be needed to write all the data to its appropriate place in the eigenvector matrix.

Instead, a block structure directly corresponding to the block structure of the original block tridiagonal matrix must be imposed on the eigenvector matrix. Then each of these blocks is individually distributed block-cyclically, as it is with the initial block tridiagonal matrix. Corresponding blocks in the block tridiagonal matrix and eigenvector matrix will now have coinciding partitions, so eigenvectors can be easily stored without extra communication. Figure 3.5 shows an example of the eigenvector matrix that corresponds to the block tridiagonal matrix from Figure 3.4.

This decision now imposes a new problem. From Equation (2.5), the updating of the eigenvector matrix after each merge step requires a matrix multiply involving the current eigenvector matrix $\tilde{Q}$ and the eigenvector matrix $Z$ of the rank-1 updated system. Since these eigenvector matrices are divided into blocks imposed by the block tridiagonal structure, a “higher level” matrix multiply routine was written to deal with the special structure of these eigenvector matrices. This routine is essentially a triple-nested loop, with the inner loop being a call to the ScaLAPACK (PBLAS) matrix multiply routine. This introduces one more level above the kernel parallel matrix multiply routine, and an increase in communication overhead by a factor on the order of $n \times O(p^2)$ (where $p$ is the number of diagonal blocks, and $n$ is the size of the matrix) in comparison with using a single function call. The analysis follows.

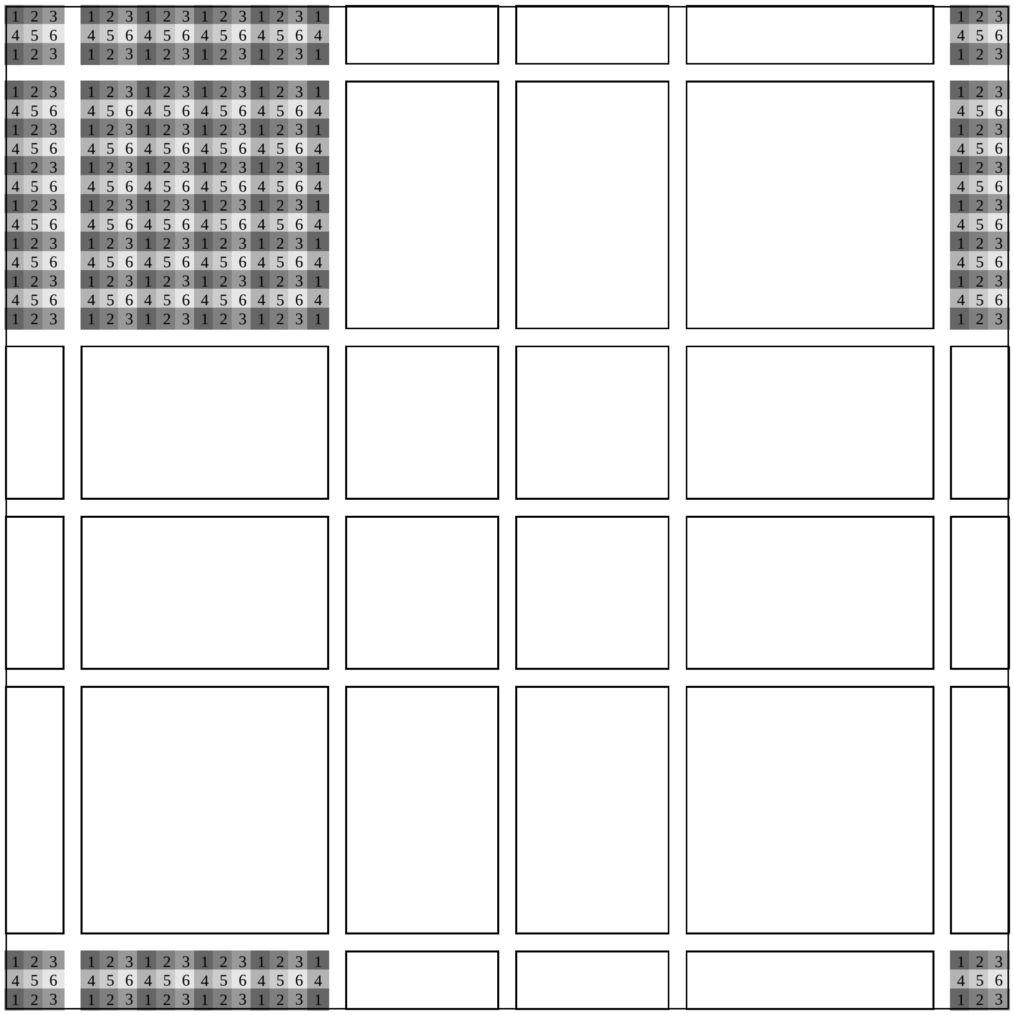

Figure 3.5: Independent block cyclic distribution of the imposed blocks of the eigenvector storage matrix. Each sub-block (imposed by the initial block tridiagonal matrix structure) is individually distributed in a block cyclic fashion, similar to the block tridiagonal matrix in Figure 3.4. To avoid redundancy, not all blocks are shown with the distribution scheme.

For simplicity, assume a block tridiagonal matrix with $p$ equal size diagonal blocks of size $k$, and assume that $p$ is a power of 2. Let a *stage* be a set of merges of $p$ blocks that combines the blocks into $\frac{p}{2}$ blocks. So the number of merges at each stage $i$ is $\frac{p}{2^i}$, where $i = 1, 2, \ldots \log_2 p$. Each merge requires $k$ rank-one modifications, assuming that no deflation is performed. In turn, each rank-one modification requires $(2^i)^3$ calls to the ScaLAPACK matrix multiply routine. The total for each stage $i$ is then $pk4^i$. Integrating over $i$ gives

$$\begin{aligned} pk\int_0^{\log_2 p} 4^i &= \frac{pk}{\ln 4}\left(p^2 - 1\right) \\ &= \frac{n}{\ln 4}\left(p^2 - 1\right) \end{aligned} \tag{3.1}$$

This indicates that $p$ is a major factor in performance and the algorithm may show better performance on the matrix with fewer blocks when comparing matrices of approximately the same size, depending on the overhead costs of a call to the ScaLAPACK matrix multiply. Of course, the matrix with fewer blocks in this case typically has the possibly offsetting property of requiring more rank-one modifications, since the block sizes are larger.

## 3.5 Deflation

A very important aspect of the serial BD&C algorithm is deflation, which was discussed in Section 2.4. Experiments have shown that deflation can significantly reduce floating point operations when computing eigen-decompositions to a relaxed tolerance (greater than machine precision). For this parallel version, implementing deflation in the same way resulted in extra communication costs, since deflation requires column permutations of the eigenvector matrix that will generally have to be done between processors. A subroutine was written to permute columns, using communication calls when the columns reside on different processors. Although it was understood that implementing full deflation may decrease performance, it was thought that the savings in floating point computations might offset the communication costs.

# Chapter 4

# Computational Results

## 4.1 Specifications

This parallel version of the Block Tridiagonal Divide and Conquer algorithm was implemented in FORTRAN. It uses the ScaLAPACK library, which includes the PBLAS low-level linear algebra routines and the BLACS communication routines. This version of BLACS is built on MPI, so MPICH version 1.2.2.3 was used. The ATLAS version of the core BLAS routines and LAPACK was used for the serial code. The compiler is the GNU g77 FORTRAN compiler (version 2.96). All timings were measured by PAPI (version 1.54).

The cluster used for these tests was the anakin cluster of the SInRG grid at the University of Tennessee, Knoxville. The anakin cluster is a group of 32 dual Intel Pentium III 500 MHz processors with 512 KB of cache and 512 MB of main memory. The network connections between nodes on the anakin cluster are Myrinet. The operating system is Red Hat Linux 7.2.

Recall that the BD&C algorithm offers the option of computing the entire set of eigenvalues and eigenvectors with reduced accuracy (and a corresponding savings in computational cost). The accuracy chosen for all of the results in this chapter was $10^{-4}$.

## 4.2 Matrices

Random matrices were created using a MATLAB script. Given diagonal block sizes, the diagonal and sub-diagonal blocks were filled in with random values in the range $[-1,1]$. The diagonal blocks were forced to be symmetric. Additionally, all values were divided by a power of 10 dependent on distance from the main diagonal. This forces the magnitude of the values to decrease from $10^0$ on the main diagonal to roughly $10^{-15}$ for the nonzero element farthest from the main diagonal. This simulates the type of matrices that can easily be approximated by block tridiagonal matrices.

## 4.3 Timings

When running the parallel routine, the serial routine was also run individually on each processor. Real time (wall time) was measured for both routines, separately on each processor, and speedup was computed as maximum real time for parallel for all processors divided by the average real time for serial for all processors.

### 4.3.1 Matrices with 10 Diagonal Blocks

Initially, experiments were run on matrices with 10 diagonal blocks. The block sizes used were 50, 100, 200, and 300, corresponding to matrix sizes of 500, 1000, 2000, and 3000. The program was run on a 2x2 processor grid (4 processors), and the block-cyclic distribution sizes used were 32, 64, and 128 (when possible). The initial results are shown in Table 4.1.

Poor performance was noted in the initial runs, so performance counters were placed in the code to measure communication calls in both numbers and time spent. This showed that the most time-consuming component of the algorithm was the block matrix multiplication discussed in Section 3.4. For matrix size $500 \times 500$, the time spent computing all matrix multiplications was approximately 27% of the total time. For the $1000 \times 1000$ matrix, it

Table 4.1: Timings on a 2x2 processor grid for matrices with 10 equal size diagonal blocks. *Sub-block sizes for block-cyclic distribution were 32, 64, and 128. Speedup is shown in parentheses. Hyphens represent where the sub-block size was too large to be effectively used for the given matrix.*

| Matrix Size | Time (seconds) | | |
|---|---|---|---|
| | 32 | 64 | 128 |
| 500 | 38.1 (0.16) | —- | —- |
| 1000 | 203.4 (0.44) | 192.5 (0.46) | —- |
| 2000 | 1316.8 (0.95) | 1537.9 (0.80) | 1515.9 (0.81) |
| 3000 | 4656.4 (1.19) | 4773.2 (1.16) | 4758.9 (1.16) |

Table 4.2: Matrix multiplication counts for matrices with 10 equal size diagonal blocks. *Data taken from the parallel run using a distribution block size of 32. In parentheses is time as a percentage of total algorithm time.*

| Matrix size | Predicted Count | Actual Count | Time (secs) |
|---|---|---|---|
| 500 | 35707 | 4318 | 10.5 (27%) |
| 1000 | 71413 | 11332 | 98.0 (48%) |
| 2000 | 141113 | 24489 | 822.3 (62%) |
| 3000 | 214240 | 36318 | 3092.5 (66%) |

was approximately 48%. For $2000 \times 2000$, 62%, and for $3000 \times 3000$, 66%. The total number of calls to the ScaLAPACK (PBLAS) matrix multiply routine was approximately 12% to 17% of the number predicted by Equation 3.1. However, it does follow the same linear pattern. The overestimates are most likely due to deflation, which makes accurate approximations difficult. This data is shown in Table 4.2.

For reference, runs were also completed for matrices with 10 unequal size diagonal blocks. The size of the diagonal blocks are as follows:

| | |
|---|---|
| 500 : | 56, 74, 57, 37, 43, 34, 50, 45, 48, 56 |
| 1000 : | 84, 99, 120, 96, 126, 75, 74, 126, 88, 112 |
| 1976 : | 131, 210, 222, 131, 141, 243, 226, 203, 217, 252 |
| 3000 : | 272, 297, 283, 272, 340, 362, 311, 344, 253, 266 |

The speedup results are comparable to the runs with equal diagonal blocks (see Table 4.3).

Table 4.3: Timings on a 2x2 processor grid for matrices with 10 unequal size diagonal blocks. *Sub-block sizes for block-cyclic distribution were 32, 64, and 128. Speedup is shown in parentheses.*

| Matrix Size | Time (seconds) | | |
|---|---|---|---|
| | 32 | 64 | 128 |
| 1000 | 161.9 (0.48) | —- | —- |
| 1976 | 6070.7 (0.92) | 6359.4 (0.88) | —- |
| 3000 | 4838.6 (1.19) | 4795.6 (1.20) | 5003.2 (1.15) |

Table 4.4: Timings on a 2x2 processor grid for matrices with 4 equal size diagonal blocks. *Sub-block sizes for block-cyclic distribution were 32, 64, and 128. Speedup is shown in parentheses. The* $4000 \times 4000$ *case could not be run with the serial version.*

| Matrix Size | Time (seconds) | | |
|---|---|---|---|
| | 32 | 64 | 128 |
| 1000 | 430.1 (0.99) | 409.9 (1.04) | 408.2 (1.05) |
| 2000 | 3257.5 (1.48) | 3137.6 (1.54) | 3151.2 (1.53) |
| 3000 | 10477.1 (1.73) | 10075.6 (1.79) | 10004.5 (1.81) |
| 4000 | 23443.4 | 22527.8 | 22370.1 |

### 4.3.2 Matrices with 4 Diagonal Blocks

Given the poor speedup for the 10 diagonal block runs, and the analysis of overhead costs due to the number matrix multiplication calls, it was decided to check performance on matrices with 4 diagonal blocks. This also allows the larger blocks to be divided over a larger number of processors. The processor grids in these experiments are $2 \times 2$ and $4 \times 4$. The results are shown in Table 4.4 and Table 4.5. When using a $2 \times 2$ processor grid, speedup approached 1.8 (and possibly 2.0 for the $4000 \times 4000$ case, which was too large for the serial version of BD&C). For the $4 \times 4$ processor grid, speedup approached 3.0. See Table 4.6 for matrix multiplication counts for these cases.

In all cases, the matrix multiplications are the dominant computational cost of the parallel algorithm, making up from 50% to 70% of the total computational time.

Table 4.5: Timings on a 4x4 processor grid for matrices with 4 equal size diagonal blocks. *Sub-block sizes for block-cyclic distribution were 32, 64, and 128. Speedup is shown in parentheses.*

| Matrix Size | Time (seconds) | | |
|---|---|---|---|
| | 32 | 64 | 128 |
| 1000 | 363.8 (1.18) | 348.4 (1.23) | —- |
| 2000 | 2244.9 (2.15) | 2120.1 (2.27) | 2133.8 (2.26) |
| 3000 | 6461.3 (2.79) | 6151.1 (2.94) | 7215.8 (2.50) |

Table 4.6: Matrix multiplication counts for matrices with 4 equal size diagonal blocks. *Data taken from the parallel runs using a distribution block size of 32. In parentheses is time as a percentage of total algorithm time.*

| Size | Predicted | $2 \times 2$ grid | | $4 \times 4$ grid | |
|---|---|---|---|---|---|
| | | Actual | Time | Actual | Time |
| 1000 | 10820 | 4716 | 244.0 (57%) | 4716 | 194.3 (53%) |
| 2000 | 21640 | 8026 | 2113.7 (65%) | 8026 | 1305.2 (58%) |
| 3000 | 32460 | 9290 | 7161.4 (68%) | 9290 | 3926.4 (61%) |

### 4.3.3 Reduced Rank Subdiagonal Block Experiments

To further examine the characteristics of the parallel BD&C algorithm, matrices with reduced rank subdiagonal blocks were considered. Reduced rank subdiagonals will decrease the total number of matrix multiplications needed to merge sub-blocks of the eigenvector matrix. Test matrices were produced with 4 diagonal blocks, and having subdiagonal blocks with ranks one-half of their size and one-tenth of their size. Results are shown in Table 4.7. Although times did decrease greatly as expected, speedup remained consistent, due to the fact that reduced rank causes a corresponding decrease in run time for the serial algorithm.

### 4.3.4 Comparison with ScaLAPACK Eigenvalue Routine

As a final test, this parallel BD&C algorithm (PDSBTDC) is tested against the dense symmetric eigenvalue routine from ScaLAPACK (PDSYEV). The reduced rank subdiagonal block test matrices from Section 4.3.3 were used, since they best represent the matrix prob-

Table 4.7: Timings on a 2x2 processor grid for matrices with 4 equal size diagonal blocks and reduced rank subdiagonal blocks. *Data taken from the parallel runs using a distribution block size of 64.*

| Size | Times (seconds) | | |
|---|---|---|---|
| | Full rank | 50% rank | 10% rank |
| 1000 | 409.9 (1.09) | 278.1 (0.92) | 140.6 (0.97) |
| 2000 | 3137.6 (1.54) | 1215.1 (1.30) | 822.1 (1.41) |
| 3000 | 10075.6 (1.79) | 2560.9 (1.50) | 1989.7 (1.62) |

Table 4.8: Timings compared with ScaLAPACK PDSYEV routine. *Times are in seconds. Sub-block sizes for block-cyclic distribution were 64. The approximate factor of increase is shown in parentheses with the PDSBTDC timings.*

| Matrix Size | 50% rank | | 10% rank | |
|---|---|---|---|---|
| | PDSYEV | PDSBTDC | PDSYEV | PDSBTDC |
| 500 | 5.4 | 63.7 (12x) | —- | |
| 1000 | 28.5 | 281.4 (10x) | 29.1 | 140.8 (5x) |
| 2000 | 196.7 | 1217.3 (6x) | 205.0 | 816.3 (4x) |
| 3000 | 624.9 | 2562.5 (4x) | 634.6 | 1982.7 (3x) |

lems from expected applications. The results are shown in Table 4.8. Although PDSBTDC takes advantage of the lower ranks of the subdiagonal blocks, it does not provide better performance than the ScaLAPACK routine. Note, however, that the pattern in Table 4.8 indicates that PDSBTDC will eventually perform better on larger matrices. This is expected, since PDSBTDC takes advantage of the banded structure of block tridiagonal matrices, while the ScaLAPACK routine assumes a dense matrix. In addition, PDSBTDC has the advantage of requiring less memory to store the matrix. This parallel implementation of BD&C would not be expected to have these advantages over a banded parallel eigenvalue routine, however.

## 4.4 Other Results

Timings and counters were embedded in the code for all communication calls needed for column permutations and eigenvector computations. Besides the matrix multiplication, all of these communications were not a large cost. For example, for a $3000 \times 3000$ matrix with 10 equal size diagonal blocks, on a $2 \times 2$ processor grid, the highest communication costs were calls relating to eigenvector computation of the rank-one updated matrix. The total average time spent for each processor was 369 seconds. This is small compared with the time spent on matrix multiplications, which was 3350 seconds. The corresponding numbers for the identical case with 4 diagonal blocks are 727 seconds and 7213 seconds. When using a $4 \times 4$ processor grid, the same times for the 4 diagonal block matrix are 602 seconds and 15865 seconds.

There are a few areas in the code where some of these communication costs can be reduced. One way is to combine several related communication calls into one, and then separate the data after receiving it. This can reduce the overhead costs associated with the communication routines. However, there is currently no reason for doing this, until the major cost of the matrix multiplications can be greatly reduced.

# Chapter 5

# Conclusion

## 5.1 Summary

This parallel implementation of the Block Tridiagonal Divide and Conquer algorithm was able to produce speedups over the serial version from 2 to 3 on larger matrices (2000 to 3000) using 4 and 16 processors. However, even with matrices suitable for the parallel algorithm, this speedup is not adequate to compete with current parallel eigenvalue routines from ScaLAPACK, even though the serial version showed good competition with corresponding routines from LAPACK [5].

The major cost in the parallel algorithm was found to be the repeated matrix multiplications required in updating the eigenvector matrix. The storage method used for the eigenvector matrix causes a large increase in overhead for communication.

## 5.2 Future Work

The storage scheme for the eigenvector matrix must be revisited. One option is to implement a specialized multiplication routine, which does not rely on ScaLAPACK's PBLAS routines. This specialized routine may be able to greatly reduce overhead. Another option is to utilize a different storage scheme altogether. A new implementation of the parallel

BD&C algorithm is currently being investigated in which the eigenvector matrix is not restricted by the form of the block tridiagonal matrix. This implementation should be more fine-grain and show better performance.

# Bibliography

# Bibliography

[1] E. Anderson et al. *LAPACK User Guide*. SIAM, February 2000.

[2] L. S. Blackford et al. *ScaLAPACK Users' Guide*. SIAM, July 1997.

[3] J. J. M. Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numerical Mathematics*, 36:177–195, 1981.

[4] Jack Dongarra et al., editors. *Sourcebook of Parallel Computing*. Morgan Kaufmann, 2003.

[5] W. N. Gansterer, R. C. Ward, R. P. Muller, and W. A. Goddard, III. Computing approximate eigenpairs of symmetric block tridiagonal matrices. *SIAM J. Sci. Comput.*, 2003 (to appear).

[6] G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins, Baltimore, third edition, 1996.

[7] M. Gu and C. Eisenstat. A stable and efficient algorithm for the rank-one modification of the symmetric eigenproblem. *SIAM J. Matrix Anal. Appl.*, 15:1266–1276, 1994.

[8] J. D. Rutter. A serial implementation of Cuppen's divide and conquer algorithm for the symmetric eigenvalue problem. Technical Report UCB/CSD 94/799, University of California, Berkeley, CA, February 1994. LAPACK Working Note 69.

[9] A. Szabo and N. S. Ostlund. *Modern Quantum Chemistry*. Dover Publications, Mineola, NY, 1996.

[10] F. Tissuer and J. Dongarra. Parallelizing the divide and conquer algorithm for the symmetric tridiagonal eigenvalue problem on distributed memory architecures. Technical Report UT–CS–98–382, University of Tennessee, Knoxville, TN, February 1998.

## Vita

Robert M. Day was born in Knoxville, TN on May 10, 1976. He was raised in Heiskell, TN, and graduated valedictorian from Anderson County High School in 1994. Afterward, he attended Lincoln Memorial University and graduated *magna cum laude* and salutatorian with a Bachelor of Science degree in Mathematics.

He is currently pursuing a Master of Science degree in Computer Science at the University of Tennessee, Knoxville.