universität wien

Masterarbeit

Titel der Masterarbeit

„Investigation of Strategies for the Parallel Computation of Eigenvectors of Block Tridiagonal Matrices – Parallel Twisted Block Factorizations“

Verfasser

Michael Moldaschl, BSc

angestrebter akademischer Grad

Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2011

Studienkennzahl lt. Studienblatt:	A 066 940
Studienrichtung lt. Studienblatt:	Scientific Computing UG2002
Betreuer:	Univ.-Ass. Privatdoz. Dr. Wilfried Gansterer

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe.

Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Wien, am September 16, 2011

Michael Moldaschl

Abstract

Today the quality of an algorithm is not only defined by the numerical accuracy and the sequential runtime (or complexity), it is also essential to be able to parallelize the computation on many computing units and to use new hardware and their new features efficiently. The multi- and many-core architecture changes the structure of work and data distribution to optimize the use of the given hardware. The algorithms used for example in LAPACK are constructed for sequential process and are limited in their parallelization and furthermore ScaLAPACK is not optimized for shared memory systems like we can also see in the results of this master thesis. Therefore it is necessary to look for other algorithms where the work can be easier distributed on large systems based on a multi-core architecture. Furthermore it is important to exploit the structure of specific problems, like LAPACK which has special methods for band and/or symmetric matrices. The twisted block factorization can be used to compute the eigenvectors of a block tridiagonal matrix, which can be seen as a generalization of band matrices. For all this reasons this algorithm, with the possibility of many independent operations, is parallelized on state-of-the-art hardware to mention also the aspects of new hardware architectures.
Different parallelization strategies are investigated and implemented to evaluate how the twisted block factorization can be parallelized most efficiently. The evaluation shows that the parallelization strategies are more efficient than the tested ScaLAPACK-Routine and new CPU features are able to strongly influence the speedup of parallel programs. Furthermore the detailed analysis of the runtime shows the new challenge of creating an efficient function on a multi-core system. The numerical accuracy is not the main aspect of this work, but a very interesting correlation between the distance of the eigenvalues and the orthogonality of the eigenvectors was found. This knowledge could be used in further researches to optimize the accuracy of the twisted block factorization.

Zusammenfassung

Heutzutage wird die Qualität eines Algorithmus nicht nur durch die numerische Genauigkeit oder die sequentielle Laufzeit (oder Komplexität) definiert, es ist auch essenziell das Berechnen auf viele Recheneinheiten parallelisieren und neue Hardware und deren neuen Merkmale effizient verwenden zu können. Die multi- und many-core Architektur verändert die Struktur der Arbeits- und Datenverteilung um die Verwendung der gegebenen Hardware zu optimieren. Die Algorithmen, welche zum Beispiel in LAPACK verwendet werden, sind für den sequentiellen Ablauf konstruiert und sind in deren Parallelisierung limitiert. Außerdem ist ScaLAPACK nicht für gemeinsam genutzten Speicher, wie auch in den Ergebnissen dieser Masterarbeit zu sehen ist, optimiert. Deswegen ist es notwendig andere Algorithmen zu finden bei denen die Arbeit einfacher auf großen Systemen basierend auf einer multi-core Architektur verteilt werden können. Weiters ist es wichtig die Struktur spezieller Probleme auszunutzen, wie es auch LAPACK durch spezielle Methoden für Band- und/oder symmetrische Matrizen tut. Die twisted block Faktorisierung kann verwendet werden um Eigenvektoren von Block-Tridiagonalen-Matrizen, welche als Verallgemeinerung von Bandmatrizen angesehen werden können, zu berechnen. Aus all diesen Gründen wird dieser Algorithmus, mit der Möglichkeit auf viele unabhängige Operationen, auf aktueller Harware parallelisiert um auch Aspekte von neuen Hardwarearchitekturen zu analysieren.

Verschiedene Parallelisierungsstrategien sind entwickelt und implementiert worden um zu evaluieren wie die twisted block Faktorisierung am effizientesten parallelisiert werden kann. Die Evaluierung zeigt das die Parallelisierungsstrategien effizienter sind als die getestete ScaLAPACK-Routine und neue Prozessoreigenschaften können den Speedup eines parallelen Programms stark beeinflussen. Außerdem zeigt die detaillierte Analyse der Laufzeit die neue Aufgabe effiziente Funktionen für multi-core Systeme erstellen zu müssen. Die numerische Genauigkeit ist nicht der Hauptaspekt der Arbeit, aber eine sehr interessante Korrelation zwischen dem Abstand der Eigenwerte und der Orthogonalität der Eigenvektoren wurde gefunden. Diese Erkenntnis könnte in weiteren Untersuchungen verwendet werden um die Genauigkeit der twisted block Faktorisierung zu verbessern.

Contents

Chapter 1

Introduction

1.1 Objective

The objective of the master thesis is the construction and implementation of parallelization strategies of the in [1] described and as a sequential program implemented algorithm. It is not the only aspect to create and describe the fastest implementation (for specific multi-core and multiprocessor systems) but also to analyze in detail all possible parallelizations of the algorithm and the influence of different aspects of state-of-the-art multi-cores. The evaluation of the different implementations should build a basis for the parallelization of future algorithms based on the twisted block factorization, because the actual version ignores some aspects of the accuracy which need to be mentioned in the future. This will change the process of the algorithm and will also influence the parallel computation.

1.2 Motivation

This master thesis concentrates on the parallelization of a new algorithm based on twisted block factorizations to calculate eigenvectors of a block tridiagonal matrix. [2] showed that the sequential implementation can be very fast and theoretically eigenvector computation based on reverse iteration (like the twisted block factorization) can have a high parallelization potential by the independent computation of the eigenvectors. That is why this method sounds very promising and furthermore the twisted block factorization consists of partly independent computations that can be used for more parallelization. Very important are possible applications where the matrix is of this kind of special structure, but there are three other possibilities to use this algorithm to calculate eigenvectors. The first one is the use of band matrices as a special case

of block tridiagonal matrices. The second strategy is the band reduction of the given matrix (band reduction can also be used for full matrices) to calculate the eigenvectors of the transformed problem and finally transform the eigenvector matrix in the reverse way. The third and last possibility is the transformation of the full matrix directly to a block tridiagonal matrix (described in [3]).
We can see that there are many applications possible for the calculation of the eigenvectors of a block tridiagonal matrix. Of course the approach based on twisted block factorizations only calculates the eigenvectors, so the eigenproblem cannot be solved alone by this algorithm. But this aspect is excluded in this master thesis. Other studies (actually in process) concentrate on the combination of different algorithms to calculate only eigenvalues of the block tridiagonal matrix and use this result for the twisted block factorization to calculate the corresponding eigenvectors.
In the next section (1.3) an overview of other works, that are related to the parallelization of eigenproblem computations or to the twisted block factorization, is given. In Section 1.4 the twisted block factorization algorithm is reviewed. In Chapter 2 improvements of the sequential implementation (implemented in [4]) are described. In Chapter 3 a short overview of its sequential implementation and different parallelization strategies for the parallel program are given. Furthermore, a detailed evaluation of all possible parallelization strategies is done by creating a directed cyclic graph (DCG) in Section 3.8.
In Chapter 4 an evaluation of the runtime of the different programs and a comparison with the well known library ScaLAPACK on different test systems is given and the accuracy is analyzed. This master thesis is focused on the parallel runtime, because the accuracy of the sequential program is already discussed in [2] and the parallelization does not change the results. Chapter 5 is the summary and conclusion describing the feasibility and the efficiency of parallelizing the twisted block factorization followed by important aspects of the algorithm and the parallel implementation which need to be further discussed.

1.3 Related Work

The related works can be split into three different areas:

1. The twisted block factorization of block tridiagonal matrices (described in [2] and [1])

2. The parallelization of eigensolver (described in [5], [6], [7], [8] and [9]) and

3. the parallelization of algorithms for block tridiagonal matrices (described in [10])

The first area is the basic for this master thesis and is important to understand what is done by the program and how the calculation of the eigenvectors work (and obviously how efficient the algorithm can be). In [2] we can see that the performance of this algorithm can be faster than LAPACK-Routines. Of course this algorithm only calculates the eigenvectors while the LAPACK-Routines calculate both (eigenvalues and eigenvectors) or only eigenvalues. But this is also an important fact: There is no option to calculate only eigenvectors with standard methods. So this algorithm can be used in combination with a method which calculates only the eigenvalues or if a problem is given where the eigenvalues are known (or easily calculated). The functionality of the twisted block factorization is detailed described in Section 1.4.

The second point is the best known and best developed area and is therefore interesting for ideas how parallelization can be done most efficiently, which limitations exists for parallelization and which compromise must be accepted to reach a good speedup also for massive parallelization. It is also very important to compare the results of this thesis with the best developed programs to be able to evaluate the results. The ScaLAPACK-Routine PDSYEVR described in [5] is based on the standard LAPACK algorithm, which transform the given matrix to a tridiagonal matrix, calculate the eigenvalues and eigenvectors and transform the eigenvectors to get the solution for the original problem. An important aspect of this parallel implementation is, that the whole tridiagonal matrix is broadcasted to all processes to calculate the eigenvalues and eigenvectors. This is a proceeding which can also be used for the parallelization of the twisted block factorization, even though the amount of data, representing the block tridiagonal problem, is even higher. The amount of non zero values of a tridiagonal matrix is equal $3n - 2$ and of a block tridiagonal matrix equal $\left(3\frac{n}{b} - 2\right) b^2 = 3nb - 2b^2$. If we ignore the subtrahend (because $b << n$) the ratio of the data of both matrix types is b.

In PDSYEVR a representation tree is constructed to distribute the computation of the different eigenvectors. The problem is, that not all eigenvectors can be calculated independently from all other and therefore crossover must be considered in the parallelization. In Figure 1.1 a simple example is illustrated how the tree is constructed/distributed. In this example processor 1 is responsible for the eigenvalues 4 to 6, but to check all crossover the local representation tree consists of further parts of the complete tree. So it is redundantly distributed to allow all processes the independent calculation (no need of data of other processes during the calculation). This distributed calculation could be interesting in the context of the improvement of the orthogonality

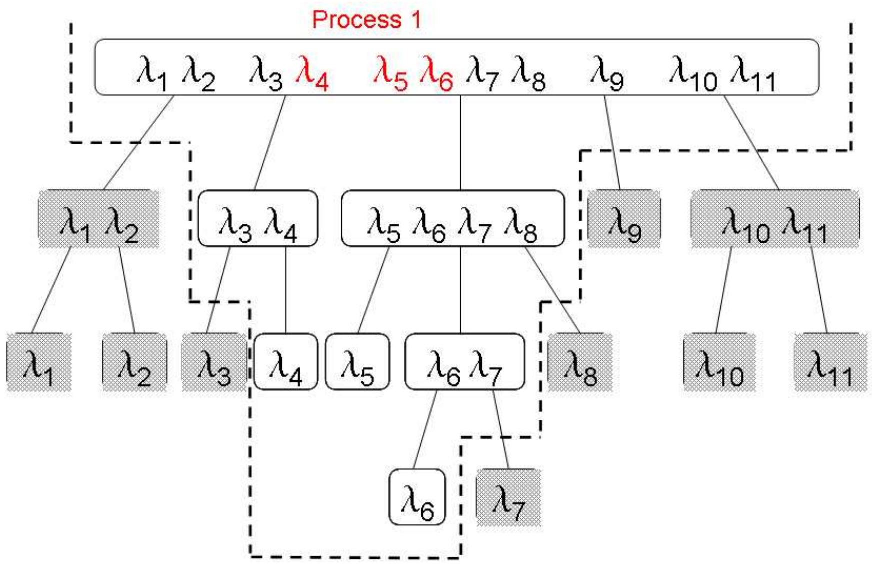


Figure 1.1: An example of the local representation tree for eigenvector calculation (based on [5]). Processor 1 is responsible for the eigenvalues 4-6, but uses information about the eigenvalues 3-8 to ensure the orthogonality of the eigenvectors. Therefore a larger part of the whole representation tree is locally used.

in the twisted block factorization and the parallelization of this improved version. The actual version of the twisted block factorization can have problems with the automatic orthogonality of the calculated eigenvectors for specific matrix types (see Section 4.2).

The last area is most compatible, because it handles the same type of problem (or rather matrix type) it just uses another algorithm to solve the eigenproblem. The block divide and conquer algorithm (BDC) described in [10] is based on the idea to transform the block tridiagonal matrix into a block diagonal matrix plus updating vectors. As an initial step the eigenpairs of each diagonal block are calculated and all updating vectors are transformed to get the following equation: $W = Q(D + \sum_i y_i y_i^\top)Q^\top$
This equation is a sequence of rank-one modifications that are sequential applied on the diagonal matrix. In each step the new values of the diagonal matrix are calculated and the matrix which must be multiplied with Q and each remaining vector y. After all rank-one modifications are done, the matrix Q represents the eigenvectors and D the eigenvalues. Each y has at the beginning only entries at two blocks and is therefore not changed in all steps of the computation. In each step two blocks are merged and all other blocks are not influenced. The parallelized implementation distributes the computation of the steps by building a tree of all merge operations and all merging operations in one level of this tree can be computed independently of all others. In the first step the parallel tasks can be up to the half amount of blocks, in the second only half of this and so on. So the amount of parallel tasks decrease in each step exponential.

Therefore a second parallelization is involved. In each step two blocks are merged, so the blocks and the cost of the operations grow in each step. The blocks are cyclically distributed among the processes and the operations are parallelized. While the blocks are small only a few processes are involved in one operation, but many parallel operations are available. When the blocks get greater the amount of parallel operations decrease, but more processes are involved in one operation. So the parallel calculation is guaranteed over the whole computation.

All these areas try to solve a whole or a part of a eigenproblem, but they do this in different ways. The first area computes the eigenvectors of a block tridiagonal matrix sequentially, the second solves the whole eigenproblem in parallel but for full matrices and the third solves the whole eigenproblem in parallel for a block tridiagonal matrix but uses a very different algorithm. Interesting is how these concepts can be combined to compute the eigenvectors of a block tridiagonal matrix in parallel. The second area includes optimizations for the accuracy which are actually not part of the twisted block factorization. Therefore this area could be more interesting for the future, but in this master thesis the concept of a representation tree is excluded. Nevertheless this master thesis tries to give a basic idea for the improvement of the orthogonality, but further concepts and implementations for the improvement are explicitly excluded (this topic is very complex and need to be analysed in a separate work).

The sequential implementation of the twisted block factorization is actually well optimized by using state-of-the-art libraries like Blas or LAPACK and algorithmic optimizations (described in 1.4) for the computation of the different twisted block factorizations are also included, but further improvements are never tried. Therefore this master thesis analyses in the progress of the parallelization which sequential optimizations can be done. The third area uses the same matrix type, therefore it is analysed if the used concept could also be used for the twisted block factorization in the combination with small and during the algorithm not growing block sizes.

1.4 Twisted Block Factorization and Inverse Iteration

In this section a short description of the twisted block factorization is given and how the eigenvectors are calculated by using an inverse iteration. First of all a definition of

a block tridiagonal matrix is given: $M = \begin{pmatrix} B_1 & C_1 & & & \\ A_2 & B_2 & C_2 & & \\ & A_3 & \ddots & \ddots & \\ & & \ddots & \ddots & C_{p-1} \\ & & & A_p & B_p \end{pmatrix}$
In the symmetric case the blocks have the following structure: $B_1 = B_1^\top, A_{i+1} = C_i^\top$
Like in the implementation in [4] the size of each block is equal b (generally the algorithm would work for different sizes for each diagonal block, but the sequential implementation supports only one size for all blocks). The number of blocks p multiplied with the block size is equal the matrix size n. The eigenproblem which is defined by the matrix M can be written as

$$(M - \lambda I)\, x = Wx = 0. \tag{1.1}$$

To solve this equation and compute the eigenvector x, a starting vector x_0 can be chosen and inserted. The following inverse iteration can be constructed ($\hat{\lambda}$ is the approximation of λ which is used in the twisted block factorization) :

1. choose x_0 with $\|x_0\|_2 = 1$ and set $i = 0$
2. solve $\left(M - \hat{\lambda} I\right) x_{i+1} = x_i$
3. normalize vector x_{i+1}
4. increase i by one and continue with point 2

The computation of the new vector can be generally done by creating a LU-Factorization of the shifted matrix W and then solve the equation. Similar to that, a LU-Factorization is constructed by exploiting the special structure of the matrix. The result of the LU-Factorization of W can have the following structure:

$$W = \begin{pmatrix} P_1^+ & & & \\ & P_2^+ & & \\ & & \ddots & \\ & & & P_p^+ \end{pmatrix} \begin{pmatrix} L_1^+ & & & \\ M_2^+ & L_2^+ & & \\ & \ddots & \ddots & \\ & & M_p^+ & L_p^+ \end{pmatrix} \begin{pmatrix} U_1^+ & N_1^+ & & \\ & \ddots & \ddots & \\ & & U_{p-1}^+ & N_{p-1}^+ \\ & & & U_p^+ \end{pmatrix} \tag{1.2}$$

In Equation (1.2) each entry is a block of size $b \times b$. The plus in the equations define that this blocks are parts of the forward factorization. The other possibility is the minus which is used for the backward factorization which is defined in Equation (1.3).

$$W = \begin{pmatrix} P_1^- & & & \\ & P_2^- & & \\ & & \ddots & \\ & & & P_p^- \end{pmatrix} \begin{pmatrix} L_1^- & M_1^- & & \\ & \ddots & \ddots & \\ & & L_{p-1}^- & M_{p-1}^- \\ & & & L_p^- \end{pmatrix} \begin{pmatrix} U_1^- & & & \\ N_2^- & U_2^- & & \\ & \ddots & \ddots & \\ & & N_p^- & U_p^- \end{pmatrix} \tag{1.3}$$

When we multiply the matrices of the forward and backward factorization we get the following two illustrations of W:

$$W = \begin{pmatrix} P_1^+L_1^+U_1^+ & P_1^+L_1^+N_1^+ & & \\ P_2^+M_2^+U_1^+ & P_2^+L_2^+U_2^+ + P_2^+M_2^+N_1^+ & \ddots & \\ & \ddots & \ddots & P_{p-1}^+L_{p-1}^+N_{p-1}^+ \\ & & P_p^+M_p^+U_{p-1}^+ & P_p^+L_p^+U_p^+ + P_p^+M_p^+N_{p-1}^+ \end{pmatrix} = \tag{1.4}$$

$$\begin{pmatrix} P_1^-L_1^-U_1^- + P_1^-M_1^-N_2^- & P_1^-M_1^-U_2^- & & \\ P_2^-L_2^-N_2^- & \ddots & \ddots & \\ & \ddots & P_{p-1}^-L_{p-1}^-U_{p-1}^- + P_{p-1}^-M_{p-1}^-N_4^- & P_{p-1}^-M_{p-1}^-U_p^- \\ & & P_p^-L_p^-N_p^- & P_p^-L_p^-U_p^- \end{pmatrix} \tag{1.5}$$

The first matrix can be calculated by starting at the first block. A LU-Factorization of B_1 is calculated, then the result can be used to solve the two systems $P_1^+L_1^+N_1^+ = C_1$ and $P_2^+M_2^+U_1^+ = A_2$. One result is directly L_1^+ while the other one is $P_2^+M_2^+$, but M_2 could be calculated when P_2 is known after the next step (this is not necessary for the algorithm). The second summand of the next block is totally known and can be subtracted from B_2 to get the next equation: $B_2 - P_2^+M_2^+N_1^+ = P_2^+L_2^+U_2^+$. This can also be solved by a LU-Factorization. These steps can be repeated down to the last block to calculate the whole forward factorization.
The backward factorization works identical, we only start at the last block with a LU-Factorization and go up to the first block.
Now we can combine both factorizations to build a twisted factorization (see Equation (1.6)). The variable i defines where the forward and backward factorization come together. There are p different possibilities where this can happen, so p different twisted

factorizations are possible. In this algorithm all of them are calculated (this can be done efficiently by calculating the complete forward and backward factorization and solve the following p equations: $B_i - P_i^+ M_i^+ N_{i-1}^+ - P_{i+1}^- M_{i+1}^- N_i^- = P_i L_i U_i \ \forall i \in [1,p]$).

$$W = P \begin{pmatrix} L_1^+ & & & & & & \\ M_2^+ & \ddots & & & & & \\ & \ddots & L_{i-1}^+ & & & & \\ & & M_i^+ & L_i & M_{i+1}^- & & \\ & & & & L_{i+1}^- & \ddots & \\ & & & & & \ddots & M_p^- \\ & & & & & & L_p^- \end{pmatrix} \begin{pmatrix} U_1^+ & N_1^+ & & & & & \\ & \ddots & \ddots & & & & \\ & & U_{i-1}^+ & N_{i-1}^+ & & & \\ & & & U_i & & & \\ & & & N_i^- & U_{i+1}^- & & \\ & & & & \ddots & \ddots & \\ & & & & & N_{p-1}^- & U_p^- \end{pmatrix} \tag{1.6}$$

This result is not only used to solve the equation in the inverse iteration it is primarily interesting for the definition of the starting vector (this and also other strategies for the definition of the starting vector are described and compared in [2]) . The important part of the twisted factorizations are the U_i or rather the diagonal entries in this blocks. To define the starting vector for the inverse iteration the minimal diagonal entry in all U_i is searched. The position of this value defines the position in the starting vector which is not null (this means only one field in the starting vector is not null).
The inverse iteration uses the fact that the eigenvalues of the inverse matrix are the reciprocal values and that subtract each diagonal element by the same scalar changes the eigenvalues in the same way. The combination of these two facts can transform any eigenvalue to the largest of the inverse problem. Therefore an approximation of the eigenvalue $\hat{\lambda}$ is used to shift the matrix, the eigenvalue λ becomes nearly zero and the inverse eigenvalue becomes very large (or rather the largest).
Each starting vector would converge to the eigenvector of the largest eigenvalue (as long as the starting vector is not orthogonal to this eigenvector[1]), but for a fast convergence the starting vector is very important. In [2] we can see, that a good approximation of the searched eigenvectors can be reached by using only one iteration. Therefore in Chapter 4 all tests are done with one iteration.

[1]If we consider numerical errors in the computation also in this case the starting vector could converge to the eigenvector of the largest eigenvalue, because the perfect orthogonality could be destroyed

Chapter 2

Optimization of the sequential implementation

Before we start to parallelize the given algorithm respectively the given program, we are looking for possible improvements. This can include better performance, but also simplified code (more readable, fewer lines of code) or a better memory usage.
In the original code three possible cases are distinguished for solving the equation $((M - \hat{\lambda})x_{i+1} = x_i)$, depending on where the minimal entry in U was found.

- The first case is that the minimal value was found in the first block (the twisted factorization is simplified to a forward factorization).
- The second case is that the minimal value was in the last block (the twisted factorization is simplified to a backward factorization).
- The third and most common case is that the minimal value was found in one of the middle blocks (the twisted factorization is a combination of the forward and backward factorization, while the position of the starting block defines which part of the forward and backward factorization is used)

We can easily see, that the first two cases are only special cases of the third one. For example, if the starting block is the first one, the twisted factorization uses all iterations of the forward factorization and zero iterations of the backward factorization. This improvement reduces the lines of code and makes the code more readable, but does not influence the performance of the program.

A second possible improvement needs less amount of memory, but can slow down the program by not saving all possible twisted factorizations to find the minimal entry in all U. When one twisted factorization is calculated all entries can be compared with the so far smallest value. If a smaller value is found the value and the whole twisted factorization block is saved. Only two blocks would be necessary (instead of number of blocks). One to store the block with the smallest value and the other to temporally compute the next twisted factorization. The disadvantage is, that whenever a smaller value is found the corresponding block must be copied. This could result in a higher amount of operations if the smallest value often changes.

The pivoting function uses memory which can already be used by the program to store a later needed result. This is an error which has only small influence to the result, but this error must be changed to ensure that no failure can occur.

The last change of the sequential program is mentioned in Section 3.8 where two pivoting operations are identified which are not necessary for the calculation. This operations would pivot the matrices M, but the explicit construction is not necessary for the calculation. The remove of this operation improves the runtime up to 8%.

Chapter 3

Parallelization Strategies

In this chapter the parallelization strategies of the twisted block factorization and their implementations are described. We assume that the serial code is known along general lines (description is given in another master thesis [4]) or at least the mathematical procedure is known (see Section 1.4). Like in Figure 1.1 it could be necessary to consider the attributes of the eigenvalues to ensure correct results. This would be important to ensure orthogonal eigenvectors, but this aspect is excluded in this master thesis. So all parallelization strategies ignore possible dependencies between the computation of different eigenvectors.

3.1 The first and simple parallel version (version0)

The first try to get a parallel version of the block twisted factorization is the use of the independent calculation of each eigenvector. The eigenvalues and the matrix are distributed. Each process has some of the eigenvalues and the whole matrix to compute the corresponding eigenvectors. After the computation, the eigenvectors are merged to get the whole eigenvector matrix (one process gets all eigenvectors computed by the other processes).

It is important that every process gets nearly the same amount of eigenvalues to do nearly the same amount of work. The work distribution is very important to get a good speedup by using more processors. A very easy distribution which will produce the best work balance which is possible for this simple parallelization is a cyclic distribution of the eigenvalues. The problem is that the eigenvectors calculated by one process are not in one series in the eigenvector matrix. This would cause a more complex merge process (see Figure 3.1). Therefore an improved version would be a block distribution where the first blocks are one element greater than the last blocks and the merge operation of all eigenvectors becomes easy (see Figure 3.2). The number of the blocks which are one greater is equal the rest of the matrix size n divided by the number of processes p. So the first processes has more work than the others, but this is only $1/n$ of the whole work.

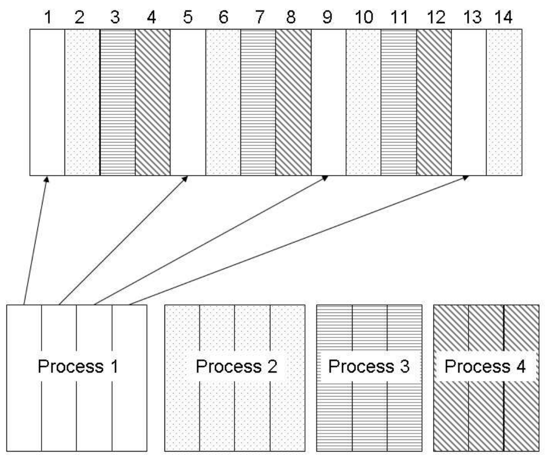


Figure 3.1: The merge/gather operation of all eigenvectors if the data are cyclically distributed among all processes

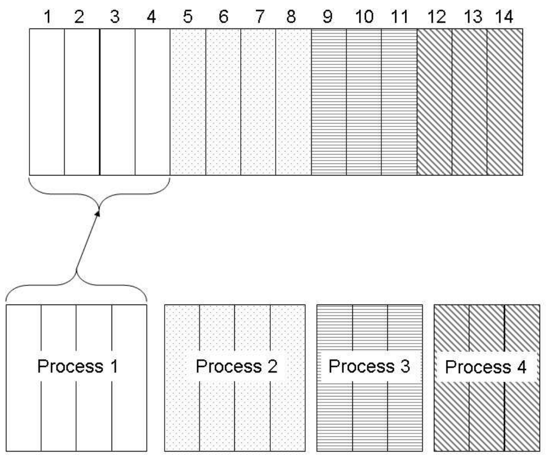


Figure 3.2: The merge/gather operation of all eigenvectors if the data are blocked distributed among all processes

3.2 More parallel version (version1)

The previous version (version0) can be improved by splitting the forward and backward factorization into two nearly independent calculations. The forward factorization is calculated up to the half of the matrix while the backward factorization is also calculated up to the half. When the middle is reached, the last calculated block of the forward and backward factorization is exchanged with the other process. Then the other process can continue calculating the backward or forward factorization (the process which calculated the first part of the forward factorization, calculates the last part of the backward factorization and vice versa). After that, one process has the data of the forward and backward factorization of the first half of the matrix and the other process has the data of the second half. These data can be used to calculate all possible twisted factorizations without need of the other data (except two blocks which where calculated in the first or second half but are necessary for the other part). After the computation of all twisted factorizations the minimal diagonal entry in all U are searched in parallel by two processes to create the starting vector. The resulting equation is then calculated in parallel by the two processes. Because of the data distribution, the upper part of the equation is solved by the first process and the lower part by the second. The computation starts at the point where the twisted factorization is chosen. From this position the calculation moves parallel up and down. If the starting point is in the upper half, the calculation is first moving down to reach the lower half fastest possible to enable the second process the calculation. Only one block must be send from the first to the second process. The program works equivalent if the starting point is in the lower half, only the computation direction is reverse.

3.3 Easier alternative to the second version (version2)

The first implementation of the previous idea (version1) is done with MPI, so the communication is really done by sending the data to the other process. Normally MPI is that efficient implemented that it uses the shared memory (if available) to communicate and should be very fast [11]. But to compare this solution with a easier implementation for a multi-core processor, the second version is also implemented with OpenMP. The forward and backward factorization is defined as (parallel OpenMP) sections and the twisted factorization is parallelized by splitting the loop. So one thread calculates the whole forward and the other thread calculates the whole backward factorization, then the twisted factorization is parallel calculated by them. The solving of the equation or the searching of the minimal entry is not parallelized, because this would be much more complicated and the runtime for this parts is much less than the calculation of the factorizations. This implementation is much easier and the resulting code is much more readable than the MPI-Version. So it has already two improvements, but much more important will be the performance (see Chapter 4.1). Another disadvantage of OpenMP is, that it is not as flexible as MPI. If no shared memory is available between two cores (if single core CPUs are given), the OpenMP implementation will get very slow (if multiple threads are running on one CPU). On the other hand the MPI would use two single core CPUs in the same way like a dual core, only the communication would need more time.

3.4 Exploiting shared memory for easier implementation (version3)

Another idea is to parallelize version0 partly by OpenMP. This means that the eigenvalues are distributed between the different MPI-Processes and then the loop over all eigenvectors, that must be calculated by one MPI-Process, is parallelized with OpenMP. The improvement of this variant is, that the given Matrix need not to be copied for each core because all OpenMP-Threads assigned to one MPI-Process can access the same. It can also help to avoid unnecessary communication overhead (although the MPI-Broadcast and the MPI-Gather methods should be that intelligent implemented to automatically avoid this overhead by using the shared and fast memory of the processors).

3.5 Combination of the OpenMP Implementations (version4)

Version2 has the disadvantage that only two OpenMP-Threads can be used for parallelization. This is a very strong constraint and therefore another parallelization strategy is constructed which combines the parallelization of the forward and backward factorization and the distribution of the calculation of different eigenvectors through OpenMP. A very important problem in the implementation could be the need of nested OpenMP. We need to distribute with OpenMP two different times, which means to create threads and each thread creates later new threads. Normally this must be explicitly activated (with the command OMP_SET_NESTED) but this is not everywhere supported[1] so the implementation could get problems on different systems or by using different compiler (the used compiler are defined in Table 4.1).

3.6 Inverse use of OpenMP and MPI (version5)

The parallelization with OpenMP is normally very easy and could be used to further distribute the computation of version1. The improvement of OpenMP is the use of the shared memory, therefore OpenMP is normally used for the inner parallelization (each MPI-Process creates threads which work together on a specific problem). This strategy will use OpenMP in another way. MPI parallelize the calculation of

[1]eg. not supported in: IBM®XL C for AIX®[12] or Sun™ONE Studio 8 compilers[13]

one eigenvector and OpenMP distributes the different eigenvectors. In more detail, MPI is first used to distribute the calculation of all eigenvectors among different processes (two processes solve the same eigenvectors), then each process creates threads (by using OpenMP) to further distribute the calculation of all eigenvectors that are calculated by one MPI-Process. The distributed calculation of the factorization is then calculated among two threads among two different MPI-Processes. To be able to use MPI-Communication in OpenMP-Threads, MPI must be initialized in another way. Instead of using the *MPI_Init*-Function, in this case *MPI_Init_thread* is called with the parameter *MPI_THREAD_MULTIPLE* which defines that multiple threads can use MPI-Communication[2].

3.7 Improved second version (version6)

Version2 distributes only the forward, backward and twisted factorization with OpenMP. The other calculations are not relevant for the runtime and are therefore ignored, but the use of OpenMP for all eigenvectors cause the creation and deletion of many threads which could cost much time. The improvement of this would be the creation of the threads outside the loop, but only one thread is responsible for the calculation of the rest. So the distribution of the calculation is nearly the same as in version2 but the threads are only created one time.

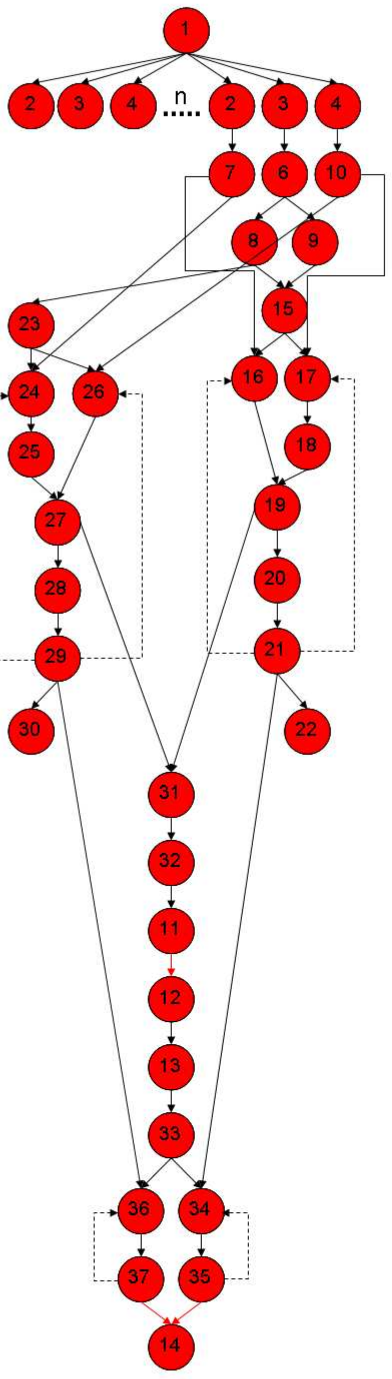
3.8 Directed Cyclic Graph (DCG)

Traditionally a directed acyclic graph (DAG) of all dependencies is created to find the critical path to define priority of operations and to identify all possible parallel operations. An easier illustration is the use of a directed cyclic graph to analyze the program for any problem size (see Figure 3.3). The illustration contains in detail only the calculation of one eigenvector. The subgraph starting with the nodes 2, 3 and 4 defines this computation (the other nodes with the same number show only the other eigenvectors). A directed arrow defines which operation must be calculated before the next can be done (the required nodes point at the followed nodes). The dashed lines define a loop and allow the computation of nodes multiple times. All nodes below the beginning of the loop are also executed more often, but a red arrow defines that the next node can only be executed if all multiple executions of the above node are

[2]all possibilities for the parameter are *MPI_THREAD_SINGLE*, *MPI_THREAD_FUNNELED*, *MPI_THREAD_SERIALIZED* and *MPI_THREAD_MULTIPLE* [14]

finished. A node can only be executed if all required nodes are finished. The structure of the graph implies that only one node is allowed to have incoming but no outgoing edges (the end node). But in this example we can see that the nodes 15, 22 and 30 have no outgoing edges. This means for the program that calculations are done which are never used. So the nodes 22 and 30 can be deleted, like it is also mentioned before at the beginning of this chapter.

In the right part of Figure 3.3 a list of all operations is given to identify which node represents which function. Furthermore a simple syntax is used to define loops. A row where a number is written in a square bracket defines how often the loop is executed and all operations which are included in this loop are indented. In the graph we can see the large parallel blocks which are already used in the different implemented methods. The basic parallelization is done by splitting the nodes 2, 3, 4 and all below. More parallelized methods additionally split the nodes 15 to 21 and 23 to 29, which are the forward and backward factorization. Furthermore the twisted factorization (and the search of the minimum) could be used as a third parallel task, but this would depend on both other processes (much synchronization would be needed). Normally the number of processes are a multiple of 2 (except triple core CPUs would be used) and the program would get inhomogeneous (e.g. the three processes that work together could be on the same quad-core or two are on the same but one is on another). The nodes 24 and 26 can also be calculated by two processes, but most of the loop cannot be parallelized (except the basic operations itself). So we can see that for the calculation of one eigenvector no further efficient parallelizations can be found. The last possibility would be the calculation of more than one eigenvector among many processes to distribute all possible parallel tasks and if at one time the parallel calculation for one eigenvector is not possible the remaining processes can calculate the others. The problem of this method would be, that for each eigenvector a separate memory is necessary (the amount of needed memory would grow with the number of parallel calculated eigenvectors) and the used memory must be synchronized (more communication occurs).



1. Broadcast
   [n]
2. a(1) = a_tmp(1)
3. b(1) = b_tmp(1)
4. c(1) = c_tmp(1)
   DSYBTEV
5. Gather

DSYBTEV:
6. Shift
7. "a2=a1"
8. "b2=b1"
9. "b3=b1"
10. "c2=c1"
    DSYBTTWF
11. Min
12. ANTIPIVOTINGVEC
13. DTRSM
    DSYBTBS
14. Norm-ev

DSYBTTWF:
15. DGETRF
    [count-1]
16. DTRSM
17. pivoting
18. DTRSM
19. DGEMM
20. b(k,1) -= a(j,3)
21. DGETRF
22. PIVOTING
23. DGETRF
    [count-1]
24. pivoting
25. DTRSM
26. DTRSM
27. DGEMM
28. b(k,2) -= c(k,3)
29. DGETRF
30. PIVOTING
    [count-2]
31. b(k,3) -= a(k-1,3) - c(k,3)
32. DGETRF

DSYBTBS:
33. DTRSM
    [blocks-1]
34. DGEMV
35. DTRSM
    [count-blocks-1]
36. DGEMV
37. DTRSM

Figure 3.3: The directed cyclic graph of the twisted block factorization. Mainly only the computation of one eigenvector is illustrated. The nodes 2 to 4 define the start of the computation of one eigenvector, therefore this operations can be done for n eigenvectors simultaneously. Node 5 would be the collection of all eigenvectors to one process to create the whole eigenvector matrix. This operation can only be started after all eigenvectors are computed.

Chapter 4

Evaluation

The evaluation of the different parallelization strategies and their implementation is divided into the performance (defined by the *parallel efficiency* in Section 4.1) and the accuracy (defined by the *residual* and the *orthogonality* in Section 4.2). The performance is much more important, because the accuracy is already discussed in [2] and should not change in parallel. So the accuracy is mainly used to specify the correctness of the parallelization. Nevertheless a short but detailed evaluation of the accuracy is given in Section 4.2 to show the advantages and disadvantages of the used algorithm. In Table 4.1 the hardware of the test systems is specified, which are used to measure the runtime of the implementations.

	System 1	System 2
Name	Standard1	Orestis
Type	-	Sun Fire X4600 M2
Processor-Type	Intel i7-860	AMD Opteron 8356
Frequency	2.8 GHz	2.3 GHz
Processor Amount	1 (=4 Cores)	8 (=32 Cores)
Memory	8GB	32GB
Compiler	GNU Fortran 4.4.3	GNU Fortran 4.4.3

Table 4.1: Test Systems

4.1 Performance

The hardware is not the exclusive aspect which is changed in the evaluation of the performance. Furthermore features of new processors, the automatic overclocking (e.g.

Intel Turbo Boost Technology) and hyper-threading (e.g. *Intel Hyper-Threading Technology*), are included in the evaluation, because it can de- or increase the benefit of the parallelization. This new features are only supported on the test system *Standard1*. In Section 4.1.1 the results of the different implementations on the smaller test system *Standard1* are illustrated. It is separated into the evaluation of the speedup with disabled features and then it is compared with the results of the same system but with enabled features (this contains four possible combinations: no hyper-threading and no turbo boost, no hyper-threading but turbo boost, hyper-threading but no turbo boost and hyper-threading and turbo boost). In Section 4.1.2 the results on the larger test system *Orestis* are illustrated. In this section the processor does not support any of these features, therefore only one type of result exists. In both sections the speedup is only tested for two different block sizes ($b = 5$ and $b = 10$) but for many different matrix sizes. In Section 4.1.3 the matrix size is constant but the block size varies. The results are very similar to the others and therefore both test systems are compared in one section.

4.1.1 Performance evaluation on system *Standard1*

The system *Standard1* has only four cores on one processor, therefore the scalability cannot be analyzed in detail. In this section the efficiency for up to four cores and the influence of new CPU-features is evaluated.
A very informative way to illustrate the efficiency of the different methods is the *parallel efficiency* which is the runtime of the sequential program divided by the runtime of the parallel program and furthermore divided by the number of used cores. In this master thesis this metric is simply called efficiency. The advantage of the used illustration is that in all cases the axis can have the same range (which makes it easier to compare all figures/tests).

Evaluation of scalability for two and four cores

Not all implementations can be used with two cores. Only version0 to version3 (version3 only with two threads) and version6 can be used because of the minimal number of processes that are needed for the other implementations.
In Figure 4.1 the parallel efficiency of different matrix sizes for block size 5 is illustrated for two processes. We can see that the methods have very different efficiency and that it is almost independent of the problem size. The two best methods in this case are the simplest version (version0) and the improvement of this implementation with OpenMP

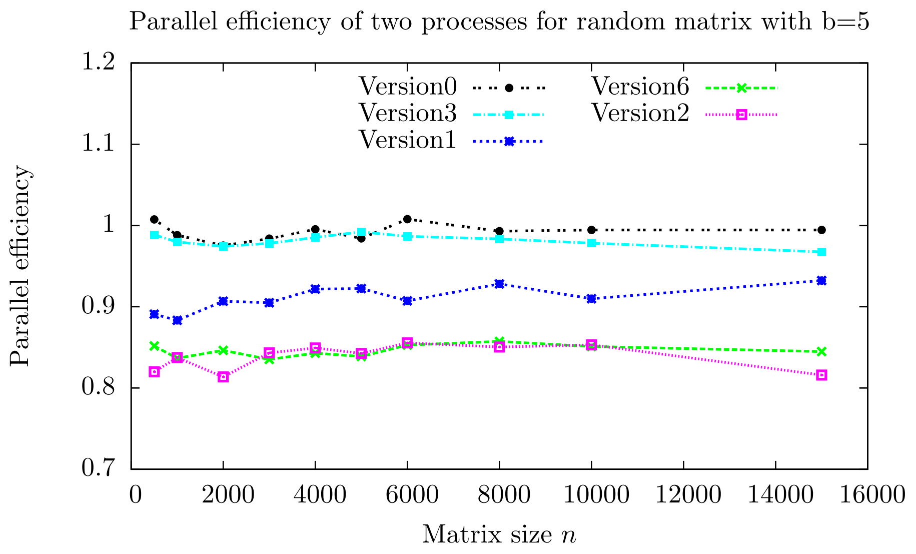


Figure 4.1: Efficiency of the different methods for different n with $b = 5$ using 2 processes on test system *Standard1*

(version3). Version1 can also reach a good speedup, although both processes are involved in the calculation of all eigenvectors. Version2 (which uses OpenMP instead of MPI to parallel calculate one eigenvector like version1) and version6 (which is only a improved implementation of version2) are significantly slower. This first figure is not very significant, because two processors are not enough to test all methods and the influence of the block size could change the results.

In Figure 4.2 we are using four processes and are able to use all implementations and constellations of thread and processes based parallelizations. The best implementations are the same as before (version0 and version3). Version3 is tested with two and four threads and it seems that two threads are a little bit faster. But it is important to consider, that version3 with four threads uses totally no MPI communication. So it will be interesting how this implementation will work for more than four processes when MPI must be used. The speedup of the different methods are nearly constant over all matrix sizes. Apparently because of the small number of processes used in this test, the need of a higher amount of data to efficiently distribute the computation is not necessary. Version1 reaches also a good efficiency and in opposition to version0 and version3, the speedup has nearly the same level as for two processes. Another interesting result is, that version5 is nearly as good as version1, although MPI and OpenMP is unconventionally used (MPI is used in all Threads of OpenMP, see Section 3.6). Till now the other methods (version2, version4 and version6) are not able to reach the good speedup of the best versions, independently of a higher matrix size.

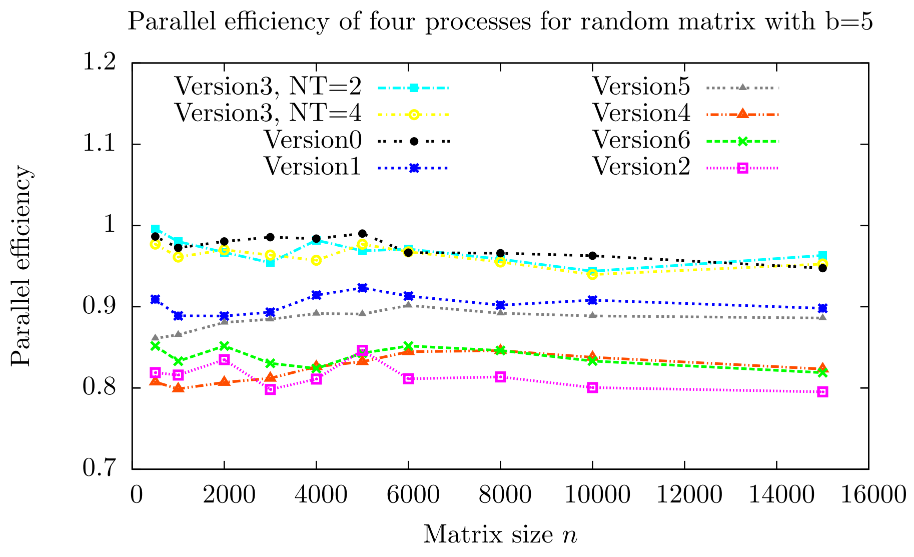


Figure 4.2: Efficiency of the different methods for different n with $b = 5$ using 4 processes on test system *Standard1*

The next step is the evaluation of a higher block size ($b = 10$) to see the influence of this factor on the speedup. In Figure 4.3 we see that the order of the implementations is very similar to block size 5, but the efficiency of version1 and version6 increased significantly. Therefore this methods could get interesting for higher block sizes. The speedup of version0 and version3 becomes quite the same, but both are slightly worse than for the smaller block size.

Figure 4.4 shows the speedup of four processors for different matrix sizes with block size 10. In this case the different methods are not clearly separated, but version3 with two threads is generally the best method. The speedup of version0 and version3 strongly decreased from block size 5 to 10, but increases again with greater matrix size. The efficiency of nearly all versions decreases from matrix size 3000 up to 10000 before it increases again. Version1 became the best method for $n = 4000$ to $n = 5000$, but then decreases faster than the other methods. Generally all methods have a lower efficiency then in all other cases. This is very interesting, because the efficiency of version1 increased from block size 5 to 10 by using two processors, but stronger decreased when four processors are used.

Evaluation of the influence of new CPU-features

The new CPU-features that are analyzed are the automatic overclocking of the cores if higher performance is needed and hyper-threading which should provide the possibility to run two threads on one core at the same time.

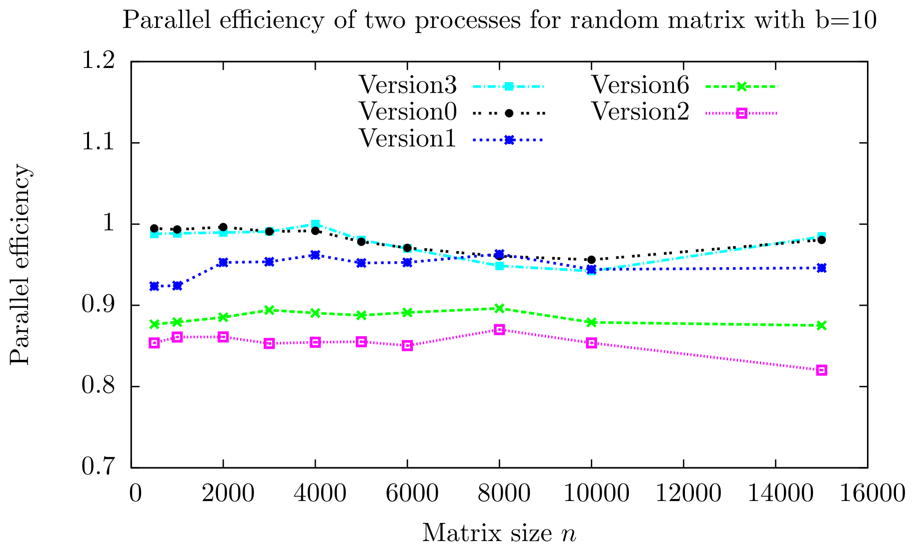


Figure 4.3: Efficiency of the different methods for different n with $b = 10$ using 2 processes on test system *Standard1*

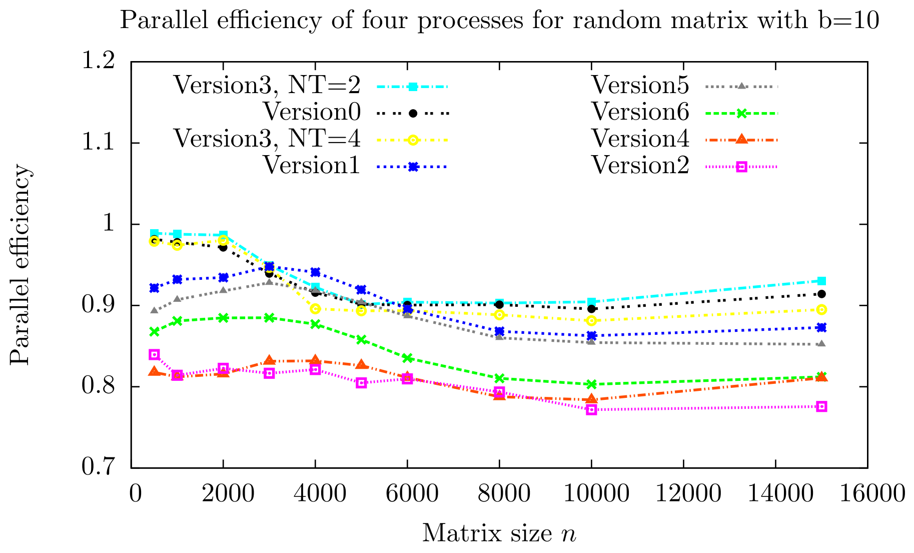


Figure 4.4: Efficiency of the different methods for different n with $b = 10$ by using 4 processes on test system *Standard1*

"Intel Turbo Boost Technology is supported in Intel Core i7 processors [...] to dynamically increase processor performance for single-threaded and multi-threaded/multi-tasking environment." [15]

How much the frequency of the cores can be increased depends on the number of cores which will be overclocked. If only one core should give more performance the frequence can be increased by 0.66ghz, each of two cores could be increased by 0.53ghz and three or four cores could be increased by only 0.13ghz (see [16]). This decreasing amount of higher frequency when the multi-core architecture is used, decreases the efficiency of parallel programs. Theoretically if we want to get the same efficiency as before each core must increase the performance by 0.66ghz. We will now calculate the decrease of the speedup because of the lower turbo boost for more cores:

2 Cores:

$$c2 = \frac{2.8 + 0.660}{2.8 + 0.530} \approx 1.0464 \tag{4.1}$$

4 Cores:

$$c4 = \frac{2.8 + 0.660}{2.8 + 0.130} \approx 1.2357 \tag{4.2}$$

The difference for two cores is quite too small to definitely evaluate the calculation, therefore only four cores are compared. The relative difference of the efficiency of all methods with turbo multiplied by $c4$ and of all methods without turbo are illustrated for block size 5 in Figure 4.5 and for block size 10 in Figure 4.6. The y axis describes the efficiency reached with activated turbo boost (Efficiency_t) multiplied with $c4$ minus the efficiency without turbo boost (Efficiency) and both divided by Efficiency. The result is the *relative Error of turbo boost model* $= \frac{\text{Efficiency}_t \cdot c4 - \text{Efficiency}}{\text{Efficiency}}$.
We can see that the inaccuracy is about $\pm 7\%$. So it is obvious that two cores cannot be analyzed, because the difference of the efficiency is (based on the Equation (4.1)) only 4.6%.
These figures show not only the accuracy of the theoretical model, it also confirms how high the loss of efficiency on new processors with the *Intel Turbo Boost Technology* is. If we compare the runtime of one and four cores with turbo boost the speedup for the four cores is about 19% ($= 1 - 1/1.2357$) lower than the speedup which would be calculated if the runtime of one and four cores without turbo boost are compared.

"Intel Hyper-Threading Technology and Intel multi-core technology are extensions to Intel 64 and IA-32 architectures that enable a single physical processor to execute two or more separate code streams (called threads)

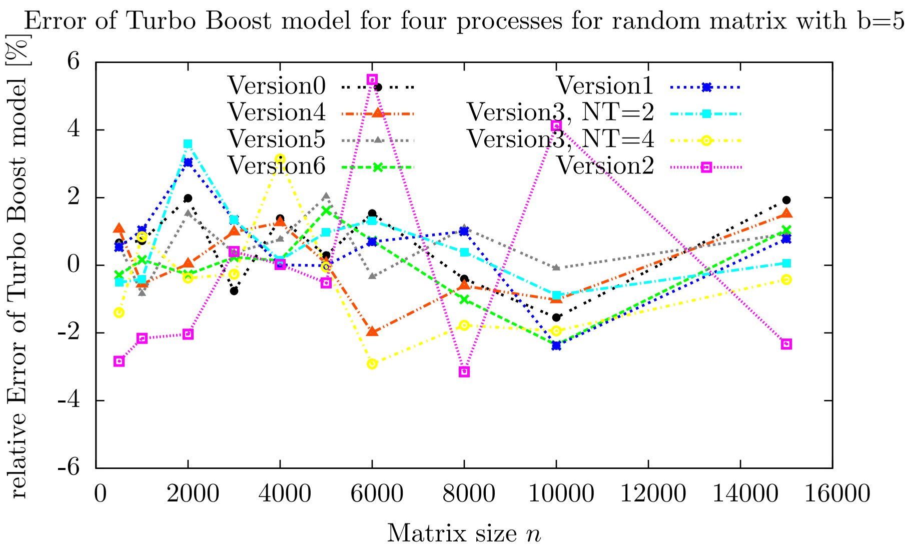


Figure 4.5: Relative Error of Equation (4.2) and the real results measured for the different methods and different n with $b = 5$ using 4 processes with activated turbo boost on test system *Standard1*

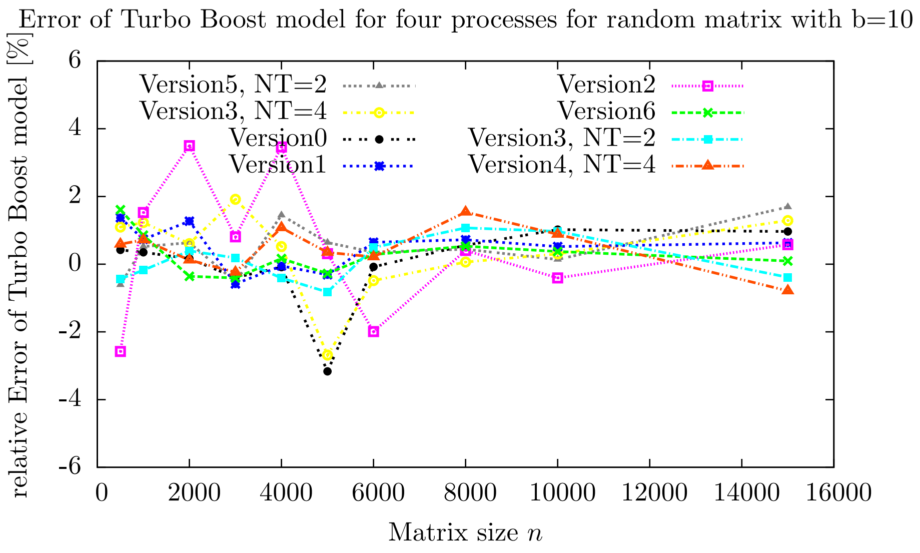


Figure 4.6: Relative Error of Equation (4.2) and the real results measured for the different methods and different n with $b = 10$ using 4 processes with activated turbo boost on test system *Standard1*

concurrently. In Intel Hyper-Threading Technology, a single processor core provides two logical processors that share execution resources" [15]

The hyper-threading technology is tested for all four cores by enabling the feature. 8 cores are simulated and used by the different methods in the same way like they would be real. Furthermore the simultaneous use of the turbo boost is possible. It would be possible to use less cores[1], but it does not seem to be a realistic test to use hyper-threading on a quad-core processor and not using all real cores. The difference between using hyper-threading or not is illustrated in Figure 4.7 without and in Figure 4.8 with turbo boost. A new variable S_h is used to define the relative improve of the efficiency or speedup (the relative change of both metrics is equal) by using hyper-threading. S_h is the speedup reached with p cores (S_p) minus the speedup reached with p cores with activated hyper-threading (S_{hp}) and both divided by S_p (see Equation 4.3). The maximal number of available cores defines the variable p and the variable S_{hp} is defined by the runtime of the sequential program (absolutely no parallelization, which implies that no hyper-threading, is used) divided by the runtime of the parallel program using $2 \cdot p$ processes on p hardware but $2 \cdot p$ logical cores.

$$S_h = \frac{S_p - S_{hp}}{S_p} \tag{4.3}$$

We would expect that the hyper-threading technology cannot improve the efficiency of the implementations, because they use LAPACK and Blas routines and are therefore very efficient. The synchronization time of the different processes or threads should also be not that long. But we can see that hyper-threading strongly optimizes the runtime of the methods. Some of them are more influenced than others. For block size 5 version2, version4 and version6 were using the additional threads in the best way. When we compare this results with the efficiency without hyper-threading in Figure 4.2, we can see that the implementations with the worst efficiency got the highest speedup from the hyper-threading. This is a coherent result, because all methods are doing the same amount of work, but they need not the same time. So some are using the hardware more efficient than others, and those who use the hardware less efficient can be better optimized by hyper-threading which uses the idle time of one process to compute another. The improvement with hyper-threading is for block size 5 at least 10% of the runtime (for all methods) and the best is over 25%.

For block size 10 we can generally see different improvements of using hyper-threading, but the best improvement was also reached by version4, which is very similar

[1]Up to 3 Cores can be disabled in the BIOS

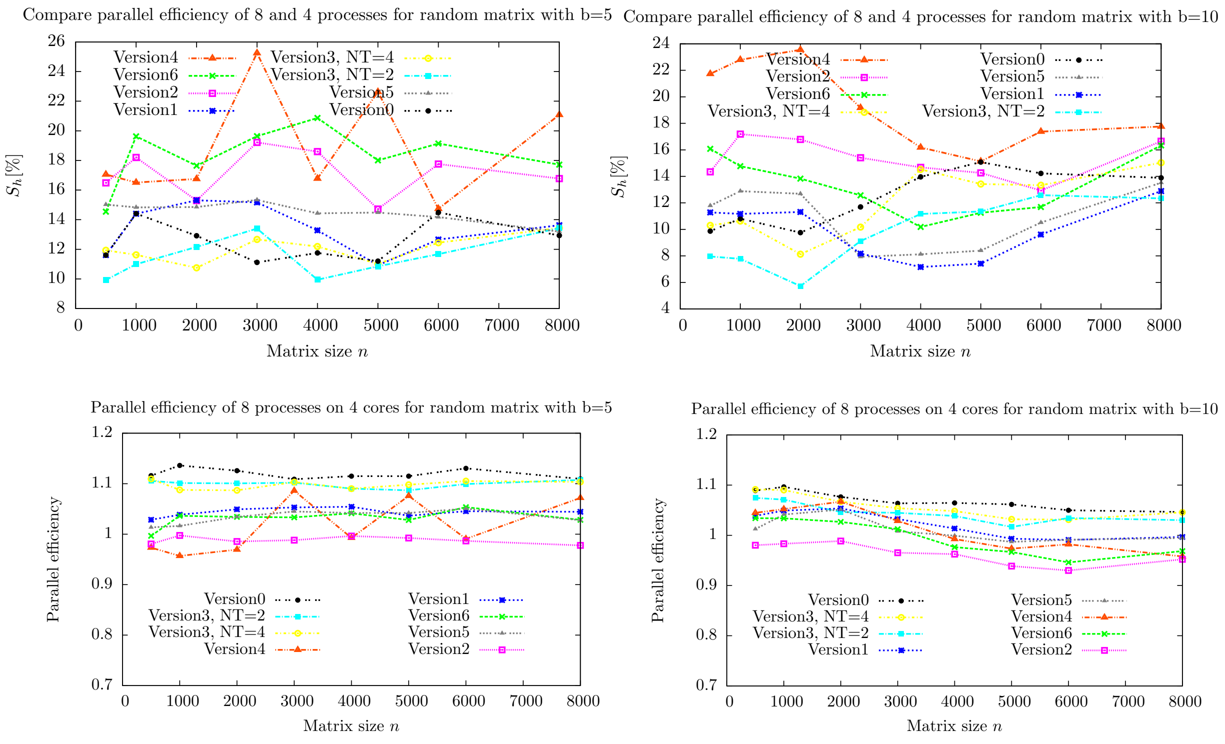


Figure 4.7: Efficiency of the different methods for different n using hyper-threading with 8 processes and 4 cores on test system *Standard1*, left $b = 5$ and right $b = 10$

to block size 5. Interesting is the good improvement of version0 and version3 with four threads, although they were also very efficient without hyper-threading. On the other hand version6 got nearly only half of the improvement as before.

Because of the strong improvement and the large differences between the different methods the efficiency of all methods are illustrated in Figure 4.7 for block size 5 and for block size 10.
For block size 5 version0 becomes clearly the best version. version4 sometimes reached a very high efficiency, but never as good as version0 or both version3. The most interesting result of this test is, that version0, version1, version3 and version5 always reached an efficiency of over 1. This means that the speedup of this methods is always higher than 4 for 4 cores. This shows obviously, that a core can be more efficiently used by a parallel program with hyper-threading than by a sequential program.
The turbo boost which was already tested for this methods can also be combined with the hyper-threading. This combination is also tested with all cores, which results in a very similar speedup (like discussed before, the overclocking of 4 cores is minimal). For block size 5 version2, version4 and version6 get the best improvement (see Figure 4.8). The efficiency of version4 increased nearly to the same level as the best methods (version0 and version3). The efficiency is low because the turbo boost of one core is

that high that this results cannot be reached by more processes.
For block size 10 version4 has got clearly the best improvement, but like in all other cases the efficiency is nevertheless not the highest. The minimal improvement decreases in the case of activated turbo boost and in the combination with block size 10 and matrix size 500 the efficiency of version1 got worse with activated hyper-threading. But this is the only constellation where activated hyper-threading is less efficient.

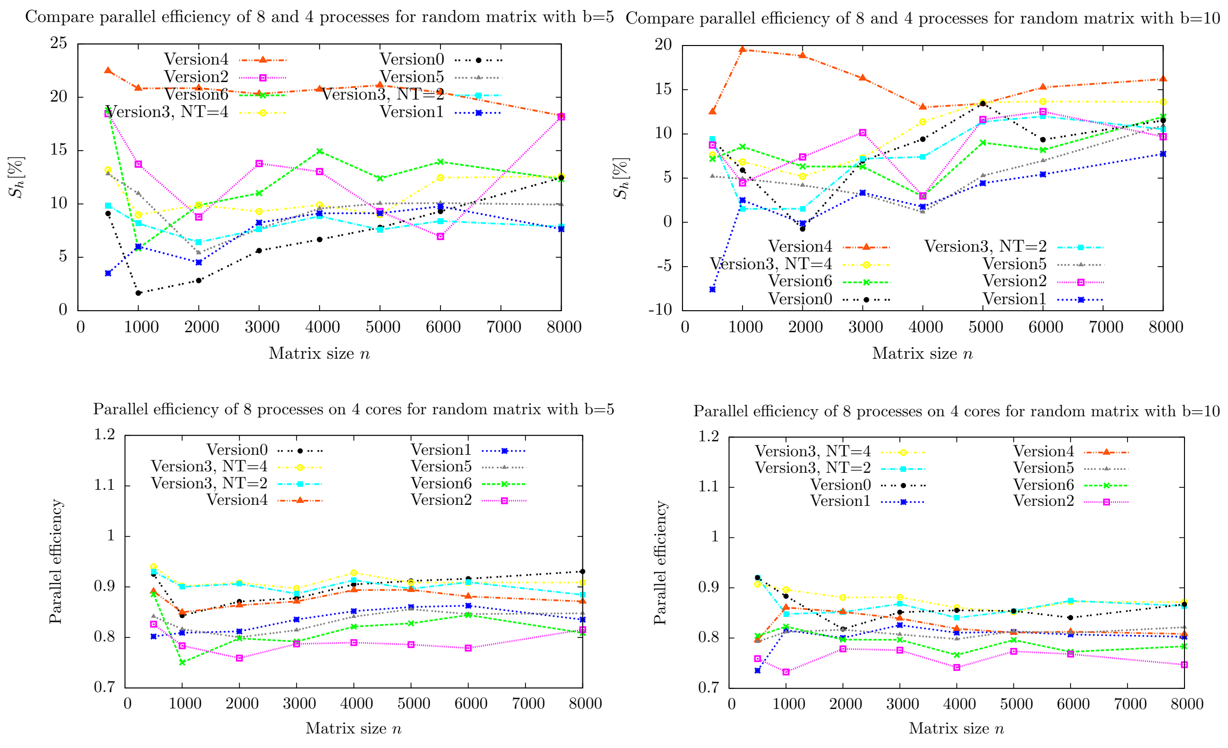


Figure 4.8: Efficiency of the different methods for different n using hyper-threading and turbo boost with 8 processes and 4 cores on test system *Standard1*, left $b = 5$ and right $b = 10$

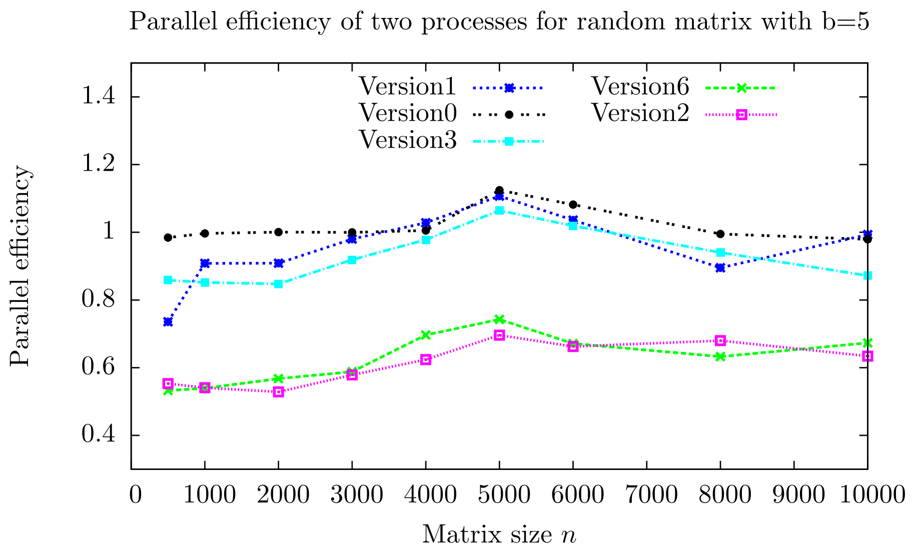


Figure 4.9: Efficiency of the different methods for different n with $b = 5$ using 2 processes on test system *Orestis*

4.1.2 Performance evaluation on system *Orestis*

After a detailed evaluation of the test system *Standard1* and two new features of processors, the different implementations are tested on a higher number of cores. Although the efficiency of the parallelization running on a quad-core is important, the implementation needs a good scalability for far more cores. In this section the performance is evaluated for up to 32 cores. We start with 2 processes and increase them to analyze how the efficiency of the different implementations changes for different matrix and block sizes and number of processes.
The first figures (4.9 and 4.10) can be compared with the results in Section 4.1.1 to evaluate the influence of the used system on the efficiency of the different versions. In Figure 4.9 the parallel efficiency of two processes for different matrices with block size 5 is illustrated. Like in Figure 4.1 version0 and verion3 are very good, but in this case also version1 reaches a similar or rather a little bit better performance than version3. Another interesting result is that the efficiency reached a value higher than 1, although 1 should be the optimal (best) value. A so far unidentified effect causes less cache misses for the parallel version (see Table 4.5 in Section 4.1.4). The worst value of version2 and version6 reached nearly 0.5, which means that using two cores can be almost as fast as using one. This is far worse than the result on the test system *Standard1*.

In Figure 4.10 four processes, and so also all different implementations, are evaluated for block size 5. In this case four versions reached for some matrices a better than optimal speedup. While version0 is for smaller matrices far the best implementation,

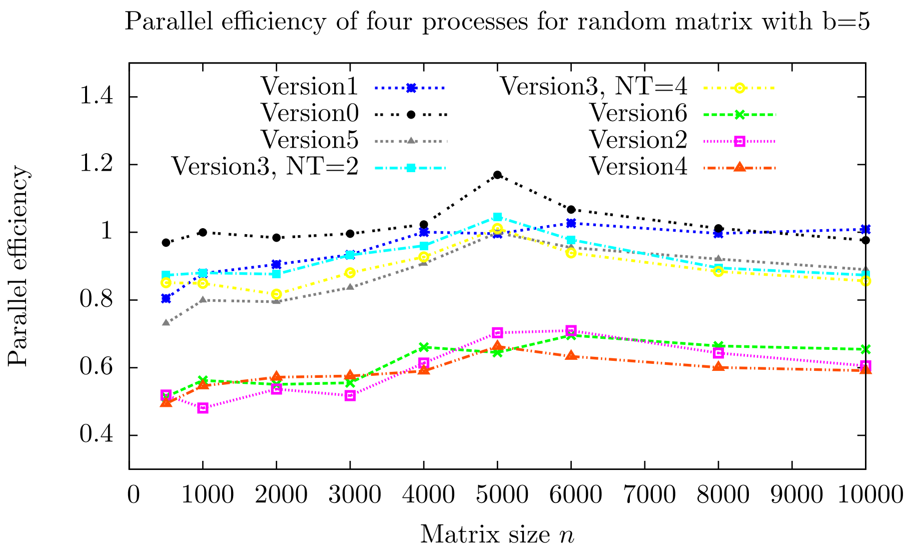


Figure 4.10: Efficiency of the different methods for different n with $b = 5$ using 4 processes on test system *Orestis*

version1 got a better result for the largest tested matrix ($n = 10000$) and furthermore a higher efficiency than 1. It seems that besides version1 also version5 is better compatible with the second system and slightly outperforms version3. Version2, version4 and version6 reached in the worst case an efficiency around 0.5. This is the same value as before, but this means that the speedup increased by a factor 2.

There is no figure for the number of processes between 4 and 16, because the results are very similar to Figure 4.10. The efficiency only decreases slightly and uniformly over all methods. In Figure 4.11 we see that also 16 processes are very similar. The only difference we can expect is that version3 with 2 threads loses a little bit less than the other methods and version0 becomes better for larger matrices than version1.

The two expected changes for 16 processes further changed for more cores. In Figure 4.12 we can see that version0 becomes definitely the best and version3 becomes as good as version1. Interesting would be how the efficiency would further change when more processors are used.

All this tests are also done with block size 10 for up to 32 cores, but they are not illustrated because there are only a few differences. Generally all versions lose more or less of the efficiency but the order of the methods are equal. Only version1 can get as fast as version0 and becomes one of the best methods. It seems that version1 could get very good for larger block sizes but we will see in Section 4.1.3 that this is not true for block sizes greater than 10.
In the previous figures the efficiency was used to illustrate the quality of the parallel programs, because this metric can be used to easily compare the use of different number

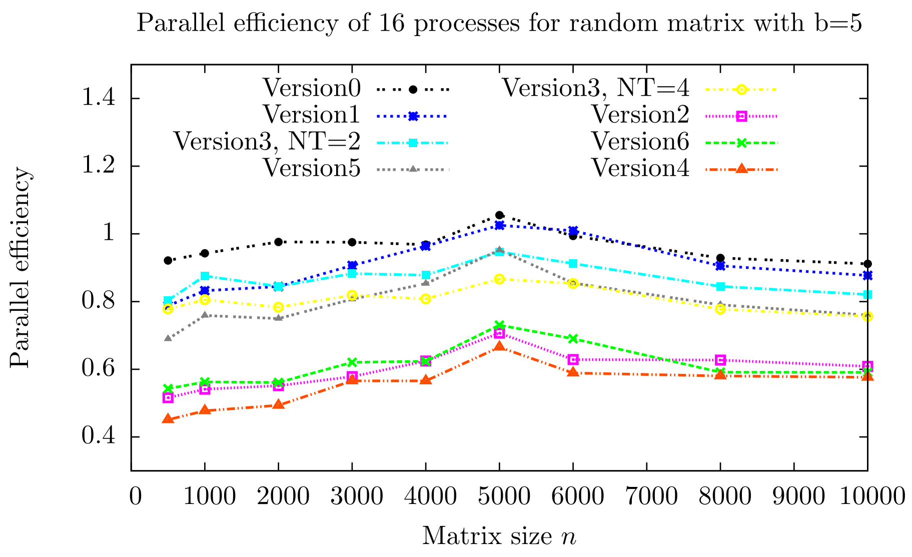


Figure 4.11: Efficiency of the different methods for different n with $b = 5$ using 16 processes on test system *Orestis*

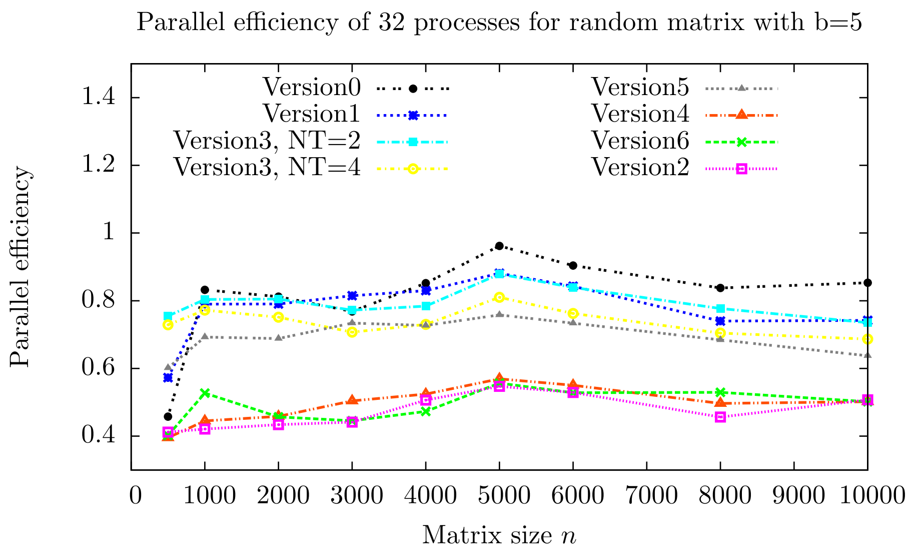


Figure 4.12: Efficiency of the different methods for different n with $b = 5$ using 32 processes on test system *Orestis*

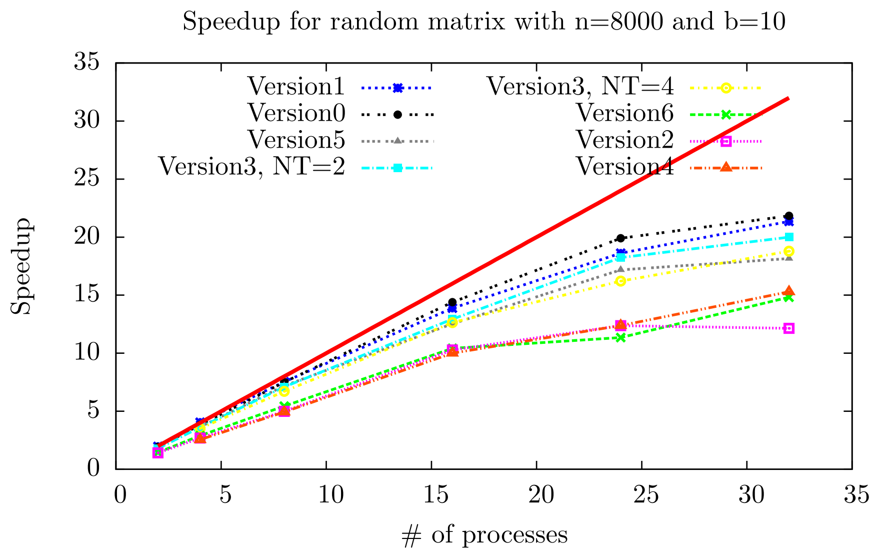


Figure 4.13: Speedup of the different methods for different n with $b = 10$ and different number of processes on test system *Orestis*

of processes. The disadvantage of this metric is that it does not illustrate how the (absolute) speedup changes and how fast it grows with increasing number of cores. Therefore in Figure 4.13 the speedup for different number of processes is illustrated. We can see that the speedup is up to 16 nearly optimal and between 24 and 32 processes the speedup is nearly the same (version2 is the only method which loses speedup by using 32 instead of 24 processes). In Section 4.1.4 it is analyzed in detail why the speedup is changing in this way.

4.1.3 Performance evaluation for different block sizes

In the previous sections we saw that the efficiency of the different methods vary for the two different tested block sizes. Therefore in this section the efficiency of the methods are tested for different block sizes on both test systems. On the system *Standard1* the changes of the speedup is tested for the smallest significant matrix size where the efficiency of all methods reaches a value that does not change for larger matrices. For block size 5 it seems to be always quite constant (see Figure 4.2), but for block size 10 it reaches the final value at matrix size 8000 (see Figure 4.4).
Because of the limitation that the size of all blocks of the matrix must be equal (see Section 1.4), the possible different block sizes are strongly restricted [2].
In Figure 4.14 we see that the efficiency of version0, version1 and version3 with two processes does not change for block sizes greater than 10. Only for smaller blocks the

[2]For matrix size 8000 all possible block sizes are: 2, 4, 5, 8, 10, 16, 20, 25, 32, 40, 50, 64, 80

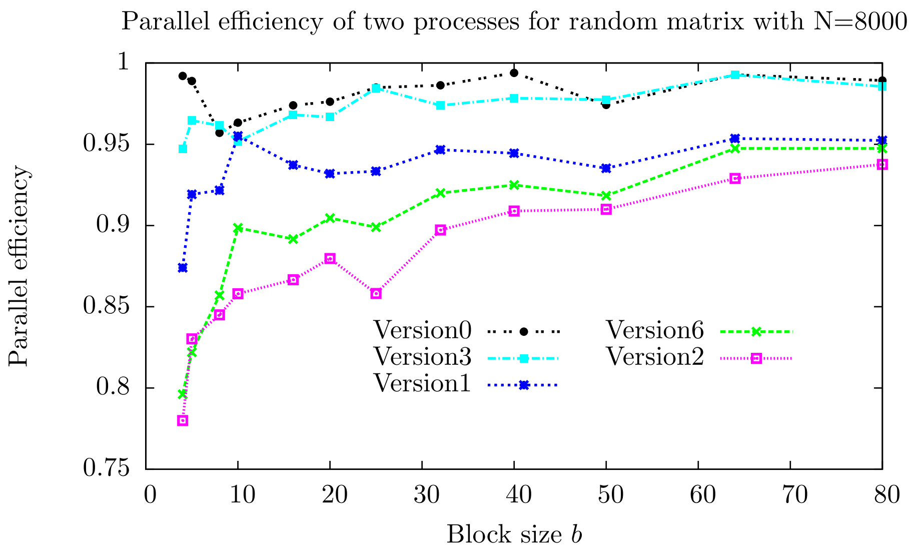


Figure 4.14: Efficiency of the different methods for different b with $n = 8000$ using 2 processes on test system *Standard1*

speedup is significantly lower. The efficiency of version2 and version6 increases steadily for larger block sizes and at block size 64 version6, which is always a little bit better than version2, reaches the the runtime of version1.

In Figure 4.15 the efficiency for the same matrices but with four processes are illustrated. Ignoring the smallest block size the speedup of all methods decrease up to block size 16. For greater blocks the efficiency of all versions increases, version0 and version3 reach the same level as for block size 5, while the speedup of the other methods further increases. But both version3 (with two and four threads) are the best methods and version0 is close behind. Like in the previous case, version6 seems to profit most of greater block sizes. Theoretically it would be interesting how the efficiency further increases, but too large block sizes make the twisted block factorization very slow. Therefore it is not useful to further analyze the method in this context.

Testing only one matrix size (although the size is well chosen) cannot show the whole evolution of all methods. Therefore the same (or rather nearly the same[3]) block sizes are tested with matrices of size 6000. There are only two differences to the previous results. The first is that for two processes the efficiency of version0 and version3 slightly decreases from block size 8 to 16 and then increases, but both are always the best methods. The second difference is much more significant. Version5 strongly decreases and becomes definitely the worst version (efficiency about 0.65, while the other methods are around 0.9). The other methods are very similar to Figure 4.15.

[3]For matrix size 6000 the block sizes 32 and 64 cannot be used, they are replaced by 30 and 60

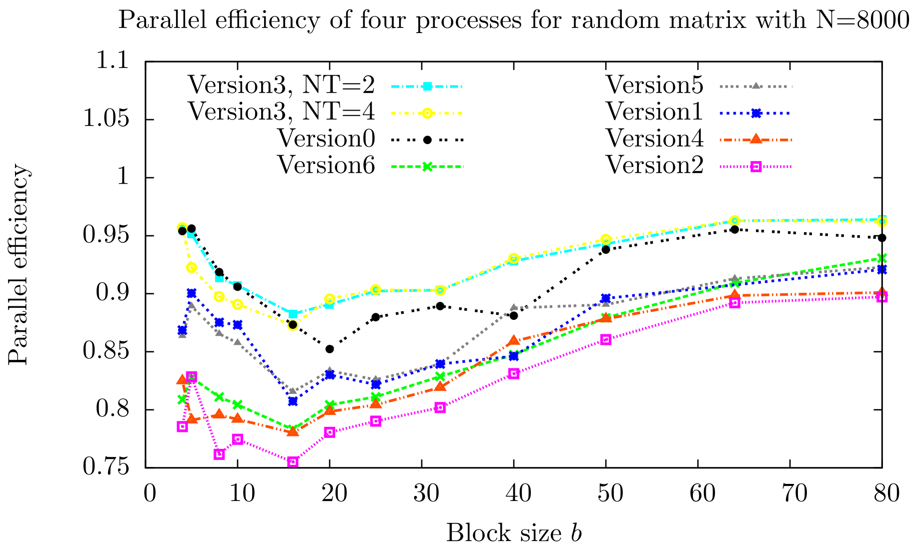


Figure 4.15: Efficiency of the different methods for different b with $n = 8000$ using 4 processes on test system *Standard1*

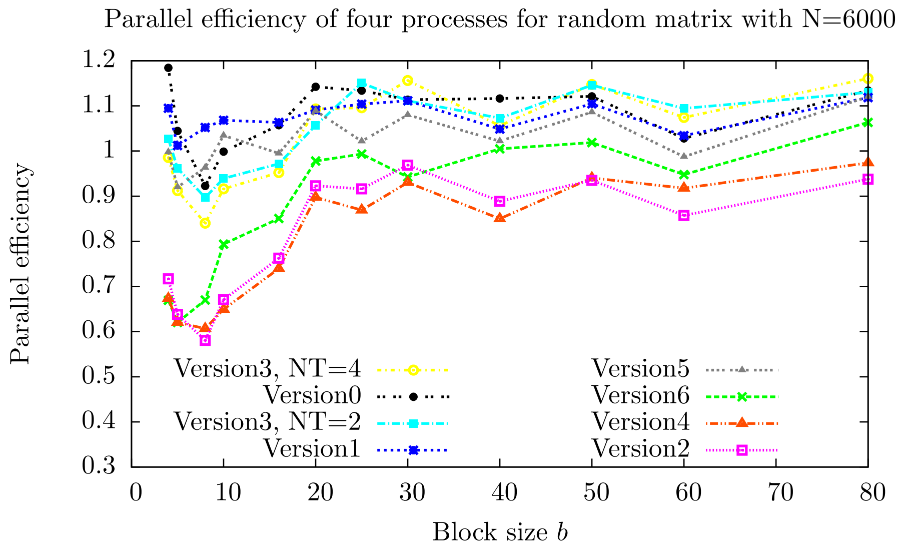


Figure 4.16: Efficiency of the different methods for different b with $n = 6000$ using 4 processes on test system *Orestis*

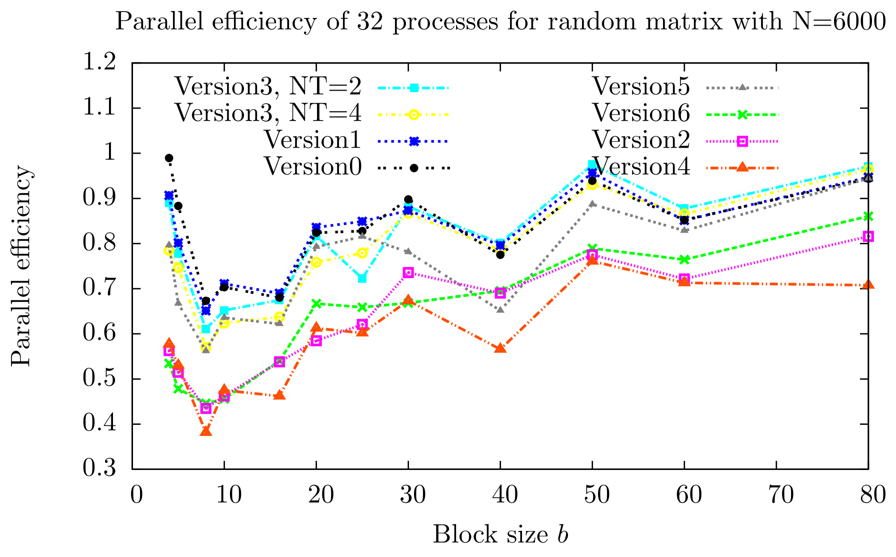


Figure 4.17: Efficiency of the different methods for different b with $n = 6000$ using 32 processes on test system *Orestis*

In Figure 4.16 the efficiency for different block sizes on the test system *Orestis* for four cores is illustrated. For block sizes between 4 and 8 the efficiency of all methods decreases, then up to a block size of 20 it increases and is then nearly constant. Only version6 slightly increases for greater block sizes and the efficiency of version1 is very constant also for small block sizes. Version0 is one of the best methods up to a block size of 40 and then both version3 becomes the best for all greater sizes. Version2, version4 and version6 are also in these tests definitely the worst implementations. The efficiency of the other methods are nearly always over, the theoretical best value, 1.
In Figure 4.17 the efficiency for 32 processes is illustrated. The results are very similar besides the general decrease of the efficiency because of the use of all cores (discussed in Section 4.1.4). The speedup of version1 for small block sizes also decreases in the same way like it happens for the other methods in this and the previous test case. For larger block sizes the efficiency of version0, version1 and both version3 are very similar and so no definite best method can be determined. Version5 becomes as fast as the best implementations for the largest tested block size 80.

4.1.4 The Performance-Model for a parallel program

In this section three different things are mentioned. First of all a model for the runtime of the algorithm on different systems is created, then the quality of the prediction is evaluated and at the end it is also analyzed which influences decrease the speedup of the implementations by taking a detailed look on the work distribution (in the other sections of this chapter it is evaluated how efficient the different methods are for different matrix and block sizes on different test systems, but it was never mentioned why they are that fast).

Theoretical construction of the Performance-Model

An interesting theoretical analysis of the performance is described in [17]. There a model is created that can be used to predict the runtime of a parallel program on different computer systems. The model parameter for defining a formula describing the performance of a parallel algorithm are the following five variables:

- τ_{DGEMM}: Time the BLAS-Routine DGEMM needs (divided by n^3)
- τ_{DGEMV}: Time the BLAS-Routine DGEMV needs (divided by n^2)
- $\tau_{\div}$: Time the division operation needs
- τ_{lat}: Time which is minimal needed to communicate with another process (latency)
- τ_{band}: The inverse of the amount of data that can be transferred to another process per second

The formula constructed for PDSYEVX in [17] is given in Equation (4.4). The model is created by splitting the ScaLAPACK-Routine in four different parts (the tridiagonalization, the bisection, the inverse iteration and the back-transformation). Then the dominating operations in each part are defined (these factors define the needed variables to describe the runtime of the routine) and how often and with which problem size the operations are called. This information allows the description of the runtime of PDSYEVX by using the 5 defined variables. In [17] it is also mentioned that the model for the bisection and the inverse iteration needed to be improved empirically, because these two operations are very compiler dependent.

$$\frac{8}{3}\frac{n^3}{p}\tau_{DGEMM}+\left(\frac{2}{3}\frac{n^3}{p}+520\frac{n^2}{p}\right)\tau_{DGEMV}+71\frac{n^2}{p}\tau_{\div}+21n\tau_{lat}lg\left(p\right)+\left(7\frac{n^2lg\left(p\right)}{\sqrt{p}}+4\frac{n^2}{p}\right)\tau_{band} \tag{4.4}$$

This model will also be created and tested for the twisted block factorization on the different test systems. For the given algorithm additional variables are defined to construct the model. These new parameters are τ_{DGETRF} and τ_{DTRSM} which represent the runtime for the LAPACK-Routine DGETRF and the Blas-Routine DTRSM (for these two variables the runtime is also divided by n^3). The parameter $\tau_{\div}$ is not necessary. The performance model is constructed for version0, thread based parallelizations would be more complicated and for this method the formula is easier to define.

The computation of each eigenvector needs almost the same operations, uses the same problem size and the same number of operation calls. Therefore it is only necessary to look on the computation of one eigenvector and multiply the result with n. The twisted block factorization consists of a forward, backward and twisted factorization. In the forward factorization a DGETRF is called for each block (n/b is the number of blocks and the size of each block is equal $b \times b$), two DTRSM and one DGEMM are called for each block except the first one. In the backward factorization the same operations are called. In the twisted factorization one DGETRF is called for each block except the first and the last. The last part is the inverse iteration which consists of DGEMV and DTRSM calls, but DTRSM is in this case only called to solve one row. Therefore we cannot use the variable τ_{DTRSM}, but to avoid an additional parameter the same value as for DGEMV is used. The variables τ_{DGEMM} and τ_{DTRSM} are for the same size very equal, so the time of DGEMV should generally be a good approximation for the time of DTRSM solving one row. Furthermore both runtimes are much lower than the rest, so an error in this part would not occur a significant failure. In the inverse iteration one DTRSM and DGEMV is called for each block, except the block where the forward and backward factorization of the chosen twisted block factorization meet. For this block two DTRSM are called.

In Equation (4.5) the formula of the sequential program is defined. Furthermore the communication overhead must be defined. The factor for the broadcast is described in [17] as $lg(p) * (\tau_{lat} + message\_size * \tau_{band})$. The message_size is equal the number of elements of the block tridiagonal matrix which is $3bn - 2b^2$ plus the number of eigenvalues n (to make the communication more easy all data are distributed in one step by a broadcast). The gather-method, used to get all eigenvectors, is the sending of n^2 data to one process. The parallel aspect is defined in Equation (4.6) and in Equation (4.7) the whole model is constructed for a parallel twisted block factorization of a $n \times n$ matrix with block size b which is computed on p processes. This formula is compared with the real runtime in Figure 4.20.

$$\tau_{serial} = \Big(\tau_{DTRSM} * \Big(\frac{n}{b} - 1\Big) * 4 + \tau_{DGETRF} * \Big(3 * \frac{n}{b} - 2\Big) + \tau_{DGEMM} * \Big(\frac{n}{b} - 1\Big) * 2\Big) * b^3 * n$$

$$+2 * \frac{n}{b} * \tau_{DGEMV} * b^2 * n \tag{4.5}$$

$$\tau_{comm} = lg(p) * \Big(\tau_{lat} + \big(3nb - 2b^2 + n\big) * \tau_{band}\Big) + n^2 * \tau_{band} \tag{4.6}$$

$$\tau_{version0} = \frac{\tau_{serial}}{p} + \tau_{comm} \tag{4.7}$$

Evaluation of the Performance-Model

In this section the performance model which is previously created is compared with the results on the test systems. In the first step the parameters of the model are determined. Therefore a program is written which does the necessary operations and measures the time.
The first variable which is examined is τ_{DGEMM}. It seems to be easy to get the correct value, but which matrix size should be used for the reference value? In [17] it is not clearly defined how the performance of DGEMM is evaluated. It is only specified that the size is *"large enough to allow acceptable DGEMM (BLAS 3 matrix-matrix multiply) and DGEMV (BLAS 2 matrix-vector multiply) performance"*. One possible procedure would be the use of growing matrices up to a size where the efficiency of DGEMM does not further increase. The maximal value could be used as a reference. This would be very easy to test on different systems, but in the twisted block factorization the DGEMM is only used for small blocks and will not be able to reach the optimal efficiency. Therefore in the first step the evolution of DGEMM is illustrated for different matrix sizes in Figure 4.18.
The next value which is defined is τ_{DGEMV}. There is the same problem like before and thats why the results for different matrix sizes are also illustrated in the same figure. The evolution of the two new values, τ_{DGETRF} and τ_{DTRSM}, are also illustrated in Figure 4.18.

The last two variables are defining the time which is needed for communication. The latency is easy measured by sending a message to another process and the other process sends back a message to the first one (the time is divided by 2).
For the bandwidth growing messages are send to see when the amount of data per

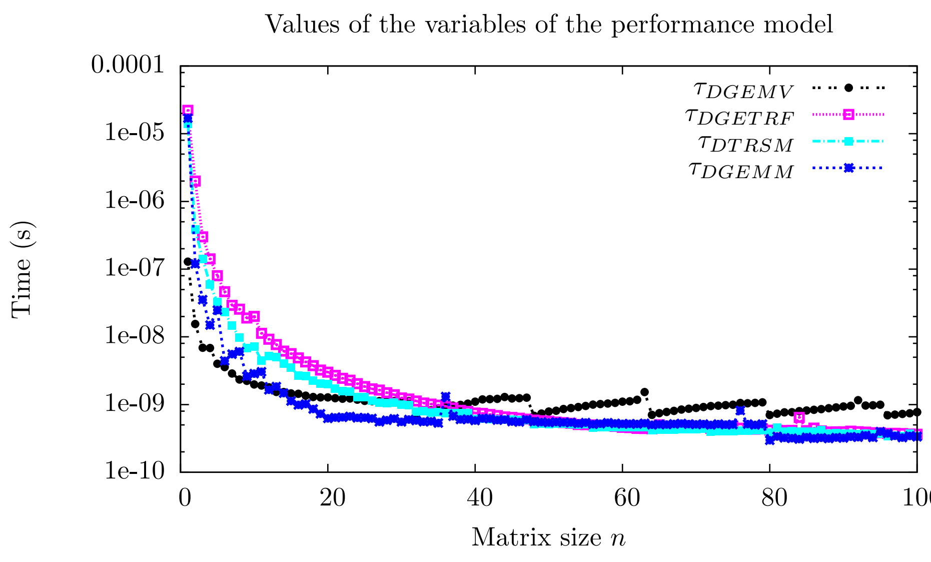


Figure 4.18: The value for the different variables depending on the used size used for the operations

second does not further increase. The problem is, that in the algorithm the amount of data which is communicated need not to be that high that this asymptotic value can be reached in the program. So the question is how good the use of the highest measured value is.
The best efficiency of DGEMM and DGEMV is nearly reached at size 20 (in logarithmic scale, the improvement up to size 100 is although factor 2). The runtime divided by n^3 of DGETRF and DTRSM converge slower and decreases significantly longer (see Figure 4.19). For smaller problems these methods also seems to need more time than DGEMM, but between matrix size 100 and 200 the methods become faster. DGEMV cannot be directly compared with the other methods, because the complexity order is only $O(n^2)$ instead of $O(n^3)$. Therefore it is always much faster than the other three routines (in the figures the runtime of DGEMV is only divided by n^2).

In the twisted block factorization the operations are only used for small problems, therefore the maximum value would cause a large error in the performance model. The problem is, that all functions change their efficiency up to 20 very fast and this sizes are the most common for this algorithm.
We will use the correct value for the specific block size to be able to get an accurate result and evaluate the performance model by reducing the influence of the measurement method as good as possible. The results for the different variables for block size 10 on both test systems are given in Table 4.2. The variables are all defined in nanoseconds. We can see that the values are very different for the systems. While on test system

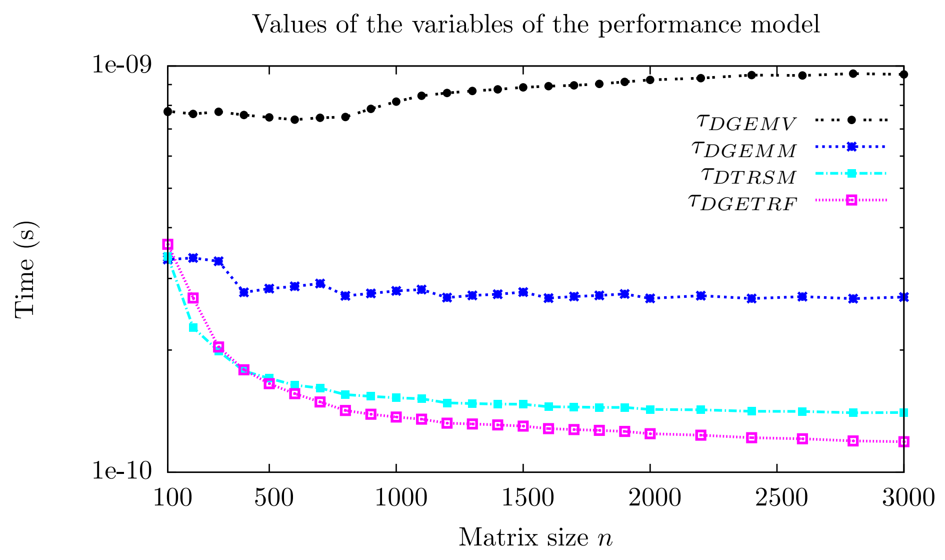


Figure 4.19: The value for the different variables depending on the used size used for the operations. The same illustration as in Figure 4.18 but for much larger problem sizes.

Orestis DGEMM needs only about 2.2 times longer than on *Standard1*, DGEMV needs about 3.7 times longer. In the third column of the table all ratios are given to be able to compare how the different variables change on both systems.

	System 1 [ns]	System 2 [ns]	Ratio
τ_{DGEMM}	0.9108	2.0313	2.2302
τ_{DGEMV}	2.7895	10.395	3.7264
τ_{lat}	214.59	702.50	3.2736
τ_{band}	1.6397	6.6380	4.0483
τ_{DGETRF}	2.3794	3.5620	1.4970
τ_{DTRSM}	0.8798	1.5306	1.7397

Table 4.2: Values of the variables for the performance model for block size 10

The values of the variables can be used in Equation (4.5) to predict the needed runtime for different matrix and block sizes. In Figure 4.20 the difference between the real runtime and the prediction divided by the real runtime (relative error of the prediction) is illustrated for different matrix sizes. The error for test system *Standard1* is for all problems nearly the same and about 17%. The prediction for test system *Orestis* is not that constant and the error is up to 30%. The quality of the results are very similar to [17].

The error in the prediction comes from other functions which are used but not mentioned in the performance model and the other problem is the data reuse. The problem size in the twisted block factorization is very small, therefore the use of the same data

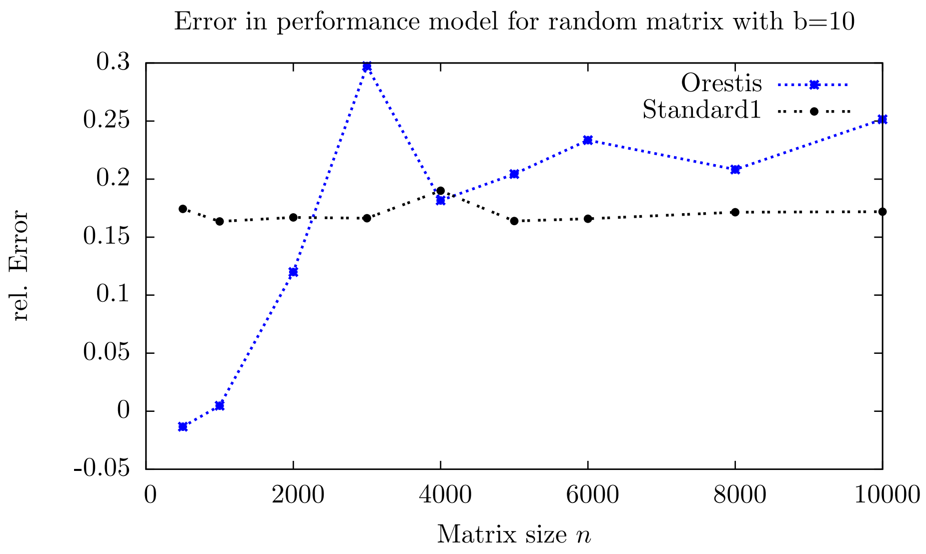


Figure 4.20: Error of the performance model for both test systems

for the same or different operations can be faster.

Evaluation of the Work-Distribution

Possibly a version could be created where the whole work is not automatically distributed, so processes which finished faster can get more work. The most important aspect for this solution is the different runtimes of the processes. If all processes need nearly the same time, better work distribution would not be able to improve the efficiency. Therefore in this section it is evaluated how efficient a dynamic scheduling could be and how good the static scheduling in the different implementations already are.

An easy example would be the dynamic distribution of the eigenvalues. The improvement could be, that not all eigenvalues are distributed at the beginning. For example, 90% are computed and when one process completes the calculation it could get further work. So the 10% are distributed between the fastest processes.

Therefore in the first step the runtime of the broadcast, the calculation and the gather method for sending the eigenvectors distributed among all processes to the root process is compared for different number of processes. In Table 4.3 the times on the system *Standard1* for 1, 2 and 4 processes of version0 (for the work-distribution only this implementation is mentioned) are illustrated. We can see that the needed time for the communication is not significant for the complete runtime, but the time needed for the calculation increases with the number of used processes although the calculation is done completely independent. This happens because some processes share the cache memory, which results in more cache misses[4]. At the end of this section a quantitative evaluation of this phenomenon is given by using PAPI[5]. For the work-distribution the difference of the calculation time between the processes is the most important aspect. We can see that in this example, the runtime for two processes is nearly equal. So a better work-distribution would not be able to improve the runtime. The difference of the highest and the average time of four cores is about 1.6%, so the best work-distribution would only be able to improve up to this value (ignoring the overhead which would be produced by the dynamic work-distribution). Based on the highest time ($40.899s$) the calculation of one eigenvector would need $0.002s = 40.899s/5000/4$ and the slowest process should calculate 82 eigenvectors less to reach the average runtime of all four processes (which is the optimal runtime), while the fastest process should calculate these 82 eigenvectors.

Cores (process)	Broadcast (s)	Calculation × p (s)	Gather (s)
1 (1)	3.8147E-006	36.994	8.9898E-002
2 (1)	6.4802E-004	37.558	8.3605E-002
2 (2)	6.4802E-004	37.608	8.3511E-002
4 (1)	1.1420E-003	39.413	8.1405E-002
4 (2)	1.1430E-003	40.852	6.1627E-002
4 (3)	1.1389E-003	40.899	4.2129E-002
4 (4)	1.1389E-003	39.745	8.1314E-002

Table 4.3: Runtime of the broadcast, the calculation and the gather-method for matrix size 5000 and block size 10 of version0 on test system *Standard1*

The next tests are done on the larger test system *Orestis* to see the changes for 1 to 32 processes. In Table 4.4 instead of illustrating the runtime of all processes, the sum of the calculation times of all processes and the maximum of all times multiplied by the number of processes are shown. The last column is the best theoretical optimization which can be reached with better work distribution. We can see that the calculation time is nearly constant for up to 8 processes. The reason for this is that 8 cores of different processors can be used and so no conflict in the caches can occur. For 16 processes at least 2 cores of each processor must be used and the shared cache significantly increases the sum of the calculation time. Using 24 or 32 processes the efficiency of each process strongly decreases because of the shared memory.
We can see that the best work distribution would only be able to improve the runtime in the most cases about 3%, only the result for 24 processes could be increased up to 4.8%. An interesting result is the runtime of the Gather-Method. It needs nearly

[4]Data which are currently not used can be replaced by data used by another process.
[5]Performance Application Programming Interface: http://icl.cs.utk.edu/papi/

the same time independently of the number of processes. This happens because the amount of data is always the same, only the amount of messages increases.

Cores	Broadcast (s)	Calculation Time (s)		Gather (s)	Max. Optimization (%)
		Sum	Maximum × p		
1	6.9141E-006	70.420	70.420	0.24143	0.0000
2	2.2271E-003	69.229	71.382	0.23474	3.0162
4	3.7260E-003	70.442	72.759	0.26383	3.1845
8	5.4801E-003	70.006	72.660	0.23839	3.6526
16	7.9391E-003	76.021	78.739	0.26641	3.4519
24	1.4227E-002	80.616	84.723	0.27406	4.8476
32	1.4291E-002	94.225	97.827	0.26337	3.6820

Table 4.4: Runtime of the broadcast, the calculation and the gather-method for matrix size 5000 and block size 10 of version0 on test system *Orestis*

Cores	L1 cache misses	L2 cache misses	FP-Operations	Total Cycles
1	6561323	160059	293440001	1317657944
2	5856744	163443	293440001	1295261590
4	6309719	204369	293440001	1281330751
8	6038785	234659	293440001	1275810079
16	6300249	301661	293440001	1263427341
24	6033947	362003	293440001	1256580641
32	6216261	451790	293440001	1257494754

Table 4.5: Cache-misses, floating point operations and total cycles for matrix size 500 and block size 10 on test system *Orestis*, the sum over all processes is shown in each cell

The efficiency on both systems strongly decreases with the number of processes, although the communication overhead is not significant. In Table 4.5 different values measured with Papi on the test system *Orestis* or calculated with them are shown. We can see for different number of processes how many *L1 and L2 cache misses* totally occur over all cores. The *L1 cache misses* are constant (except the normal fluctuation), because each core has his own L1 cache while the *L2 cache misses* strongly increases. It also matches with the runtime where the first significant increase happens with 16 processes and we can now see the same for the cache misses. The chosen matrix size for these measurements is 500 because the given hardware of the test system *Orestis* has three cache levels, but no event counter for the L3 cache is supported[6]. A larger matrix would result in more L3 and not L2 cache misses, but this effect cannot be shown on this system.

[6]All supported Papi-Events are checked with the program *papi_avail*

4.1.5 Performance comparison with ScaLAPACK

After the detailed evaluation of the performance of the different parallelization strategies and the analysation why the efficiency is changing in that way, in this section the results of the parallel twisted block factorization are compared with the ScaLAPACK implementation of the Divide-and-Conquer algorithm (PDSYEVD[7]). In the last section we saw that the communication overhead is not significant. This could imply that other implementations cannot be more efficient, but a better data distribution or rather data usage could reduce the overhead caused by the shared caches. ScaLAPACK is unfortunately not made for multi-core architecture and will therefore be not very efficient. A better library would be *Parallel Linear Algebra for Scalable Multi-core Architectures* (PLASMA) which is constructed for the new architectures, but actually there is no eigensolver implemented [18]. The comparison of the parallel twisted block factorization and PDSYEVD is quite difficult, because the amount of data used in this functions is extremely different and therefore also the amount of memory needed to solve the eigenproblem. For a good comparison it is necessary to use the ScaLAPACK routine in the most efficient way. Therefore different parameters are changed to find the best configuration for PDSYEVD. The parameters that can be changed are the *Process Grid*[8] or the blocking of the rows and columns of the data (this parameters are different for the two test systems and depend on the amount of processes).

PDSYEVD computes the eigenvalues and eigenvectors of the given matrix, while the twisted block factorization computes only eigenvectors. For a complete comparison a optimized parallel program should be used to calculate only the eigenvalues of the block tridiagonal matrix, but actually this is not available. Therefore this is excluded in these tests, but would be interesting for the future. In Figure 4.21 the efficiency of PDSYEVD is compared with all parallel twisted block factorization implementations on test systems *Standard1* using 4 processes.

For small matrix sizes the efficiency of PDSYVED is very low, but is strongly increasing up to a matrix size of 6000. For matrices greater than 8000 the efficiency is slightly decreasing. In all cases the ScaLAPACK-Routine is not able to reach the speedup of the worst parallel twisted block factorization. This result confirms the good quality of the parallelization strategies.

In Figure 4.22 the speedup of PDSYEVD is compared with the other implementa-

[7]It would be better to compare the twisted block factorization with an implementation for banded matrices, but in ScaLAPACK no eigenproblem solver for banded matrices is available (List of functions on http://www.netlib.org/scalapack/double/)

[8]http://netlib.org/scalapack/slug/node70.html

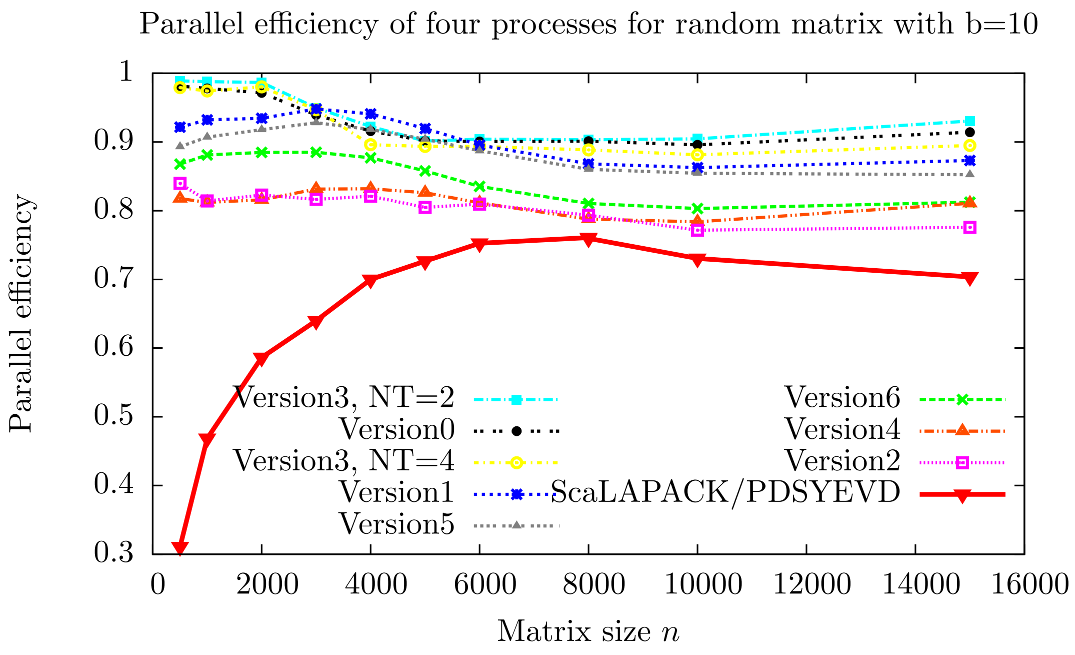


Figure 4.21: Efficiency of the different methods and the ScaLAPACK-Routine PDSYEVD for different n with $b = 10$ using 4 processes on test system *Standard1*

tions on test system *Orestis*. In this case the efficiency is illustrated for different number of processes because there are more cores available and it is interesting to see how the value decreases. For a smaller number of processes (2,4 and 8) the ScaLAPACK-Routine reaches a quite good efficiency, nearly as good as the best parallel twisted block factorization. For 16 processes the speedup strongly decreases (this is the first case where more than one core of each processor is used). We can see that the multi-core architecture has a very large influence on the performance of PDSYEVD. The speedup for 16, 24 and 32 processes is nearly the same and therefore the efficiency is strongly decreasing to a very low value.

4.2 Accuracy

Basically the accuracy is of course an essential aspect of an algorithm, but for the sequential program the accuracy is already discussed in [2]. Therefore this section will analyze shortly the accuracy of the parallel implementations to give an complete view of the twisted block factorization and how efficient this algorithm can be used for eigenproblems with different attributes.

The following matrix types are used to test the twisted block factorization:

0. Random matrices with values uniformly distributed in [0,1]

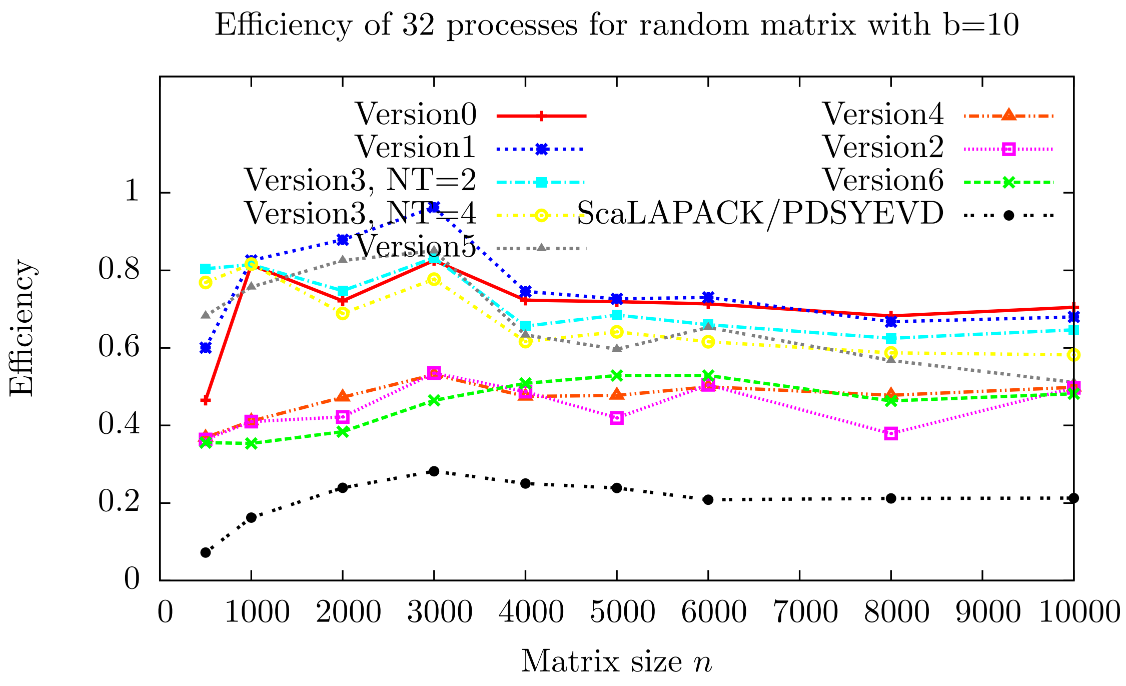


Figure 4.22: Efficiency of the different methods and the ScaLAPACK-Routine PDSYEVD for different number of processes with $b = 10$ and $n = 8000$ on test system *Orestis*

1. Eigenvalues clustered around $\pm\varepsilon$[9]

2. Eigenvalues clustered around ± 1

3. Eigenvalues geometrically distributed in $[-1, -\varepsilon] \cup [\varepsilon, 1]$

4. Eigenvalues arithmetically distributed in $[-1, -\varepsilon] \cup [\varepsilon, 1]$

5. Eigenvalues whose logarithms are uniformly distributed in $[-1, -\varepsilon] \cup [\varepsilon, 1]$

6. Eigenvalues uniformly distributed in $[-1, 1]$

The following aspects could be mentioned to analyze and define the accuracy of the twisted block factorization:

- **Residual:** $\mathfrak{R}_i = \left(\frac{\|(A-\lambda_i I)x_i\|_1}{\|A\|_1 \|x_i\|_1}\right)$
  From the mathematical point of view, the matrix A shifted by one of his eigenvalues multiplied by the corresponding eigenvector must result in the null-vector. So the error in the calculation is given by the norm 1 of the resulting vector

[9]ε defines the machine epsilon which is the largest value for a double precision variable where $1.0 + \varepsilon \equiv 1.0$

normalized by the norm 1 of the matrix A and the norm 1 of the corresponding eigenvector (which is in this case equal 1). $\mathfrak{R}_i$ is the residual for the i-th eigenvalue and eigenvector.

- **Orthogonality** $\mathfrak{O}_i = \left\| \left(X^\top X - I \right) (:, i) \right\|_\infty$
  Another attribute is that all eigenvectors are orthogonal to each other (scalar product of x_i and x_j is 0 $\forall i, j$ with $i \neq j$) and each eigenvector is normalized (scalar product of x_i with itself is 1 $\forall i$). The given metric for the orthogonality considers both attributes. The transposed of the eigenvector matrix X, composed of the eigenvectors x_i, multiplied by X should result in the identity matrix. Each diagonal element represents the scalar product of each eigenvector with itself and all the off-diagonal elements represent the scalar product of all eigenvectors with each other. Subtract the resulting matrix by the identity matrix should result in a matrix with all elements equal zero. Splitting this matrix into vectors and calculating the maximum norm of each calculates the quality for each eigenvector separately.

This master thesis illustrates the quality of the accuracy in a new way by creating two tables (see tables 4.6, 4.7) where the residual and the orthogonality of each eigenvector is used to calculate the mean, the standard deviation, the minimum, the maximum and the percentage of values which are smaller or equal than $n \cdot \varepsilon$ or smaller or equal than $n \cdot \sqrt{\varepsilon}$. In the context of accuracy the average of all values can be strongly dominated by only one (or a few) values. Therefore the mean and the standard deviation of the logarithm of the values are created to look on the number of correct digits. This metric is less influenced by outliers.
For the residual we can see in Table 4.6 that the twisted block factorization gets a good approximation with only one inverse iteration in nearly all cases. Only one outlier, which is also larger than the greater threshold $n \cdot \sqrt{\varepsilon}$, in matrix type 3 was found during these tests[10]. The results for matrix type 0, 1, 2 and 4 are perfect, the residual is for all eigenvectors smaller than $n \cdot \varepsilon$. In matrix type 6 one eigenvector has a little bit larger residual, but it is even though very good. In matrix type 5 some eigenvectors have a larger residual, but also all of them are smaller than the greater threshold.

[10]Further researches are necessary to find out in which cases the residual can get that bad

matrix type	$\leq n \cdot \varepsilon$ [%]	$\leq n \cdot \sqrt{\varepsilon}$ [%]	average	deviation	maximum	minimum
0	47.6	100	-12.7	0.58	-10.82	-16.0
1	0.06	0.06	-0.06	0.46	0	-14.8
2	0.06	0.06	-0.08	0.50	0	-15.0
3	12.6	67.5	-7.22	4.64	0	-16.0
4	73.9	100	-13.0	0.49	-10.9	-16.0
5	43.4	62.1	-8.20	6.44	0	-16.0
6	92.6	100	-13.8	0.72	-10.5	-16.0

Table 4.7: The table consists of different information about the orthogonality of the eigenvectors for matrices of different matrix types. The first column defines the type, the second and third columns are the percentage of all orthogonalities that are smaller than the two thresholds, the other columns are the average exponent, the deviation of the exponent, the maximal and the minimal exponent of the orthogonality, this table is an advancement of the illustration in [2]. If the exponent is less than 10^{-15} the value is rounded down to zero, which means that the orthogonality is almost one.

matrix type	$\leq n \cdot \varepsilon$ [%]	$\leq n \cdot \sqrt{\varepsilon}$ [%]	average	deviation	maximum	minimum
0	100	100	-15.9	0.22	-14.0	-16.4
1	100	100	-28.7	4.38	-15.3	-33.1
2	100	100	-16.1	0.44	-13.9	-18.9
3	99.9	99.9	-18.6	1.73	-0.97	-23.6
4	100	100	-16.3	0.41	-13.7	-18.1
5	84.5	100	-17.6	4.10	-8.60	-28.0
6	99.9	100	-15.8	0.53	-12.1	-17.4

Table 4.6: The table consists of different information about the residual of the eigenvectors for matrices of different matrix types. The first column defines the type, the second and third columns are the percentage of all residuals that are smaller than the two thresholds, the other columns are the average exponent, the deviation of the exponent, the maximal and the minimal exponent of the residuals, this table is an advancement of the illustration in [2]

In Table 4.7 we can see that the twisted block factorization has problems with the orthogonality of the different eigenvectors. In the matrix types with no clustered eigenvalues (0, 4, 6) the orthogonality of each eigenvector is smaller than the greater threshold and in matrix type 6 it is also in nearly all cases smaller than the other threshold. In matrix type 3 and 5 some eigenvectors are absolutely not orthogonal while others are. Completely not useful in the context of orthogonal eigenvectors is the twisted block factorization without any further orthogonalization strategies for the matrix types 1 and 2. There is only one eigenvector in each of the two matrices which is orthogonal to all others (these are the two eigenvalues which are not in the clusters).

In Figure 4.23 the orthogonality for each eigenvector of 50 different matrices of each

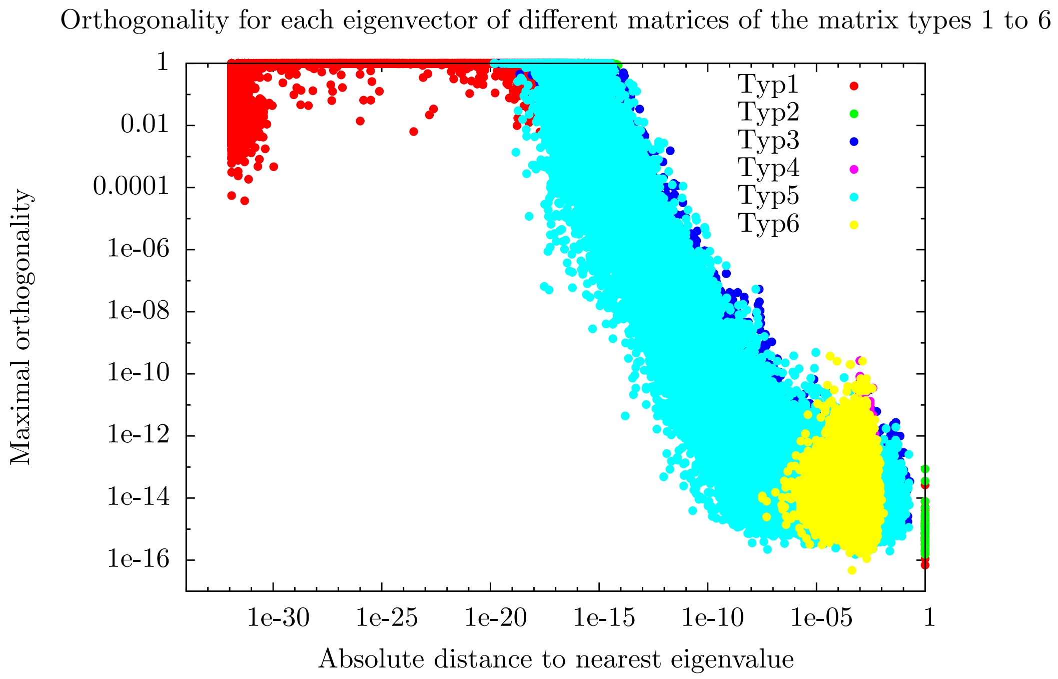


Figure 4.23: The distribution of the orthogonality depending on the minimal absolute gap between the eigenvalues, each illustrated point is the maximal orthogonality of one eigenvector to all others of the same matrix, 50 matrices with $n = 1000$ of each matrix type are used

constructed matrix type (1 to 6) related to the absolute distance to the nearest eigenvalue is illustrated (the matrix size is always equal 1000, this are $6 \cdot 50 \cdot 1000 = 300000$ eigenvectors which are all illustrated in the figure). For the smallest gaps ($< 10^{-30}$) the orthogonality is between 1 and 10^{-4}, for very small gaps up to 10^{-18} the orthogonality is nearly always 1. For larger gaps the orthogonality is inversely proportional to the smallest distance of the nearest eigenvalue. This is a very interesting result, because it seems to be possible to estimate the quality of the orthogonality only with the distance between the eigenvalues (which is very easy to calculate). This information can be used to reorthogonalize only the eigenvectors corresponding to eigenvalues with a very small distance (even though how many eigenvectors must be orthogonalized depends on the needed accuracy). Another result is that the orthogonality cannot be guaranteed to be smaller than 10^{-12}, independent on the distribution of the eigenvalues[11].

[11]sole exception was found for eigenvectors if the distance for the eigenvalues is nearly 1, then the orthogonality was always smaller than 10^{-13}

Chapter 5

Conclusion

5.1 Results

Different aspects of the parallelization of the given sequential twisted block factorization were mentioned in this master thesis. Therefore different parallelization strategies are investigated. We saw that version0, version1 and version3 (in two different constellations of MPI-Processes and OpenMP-Threads) are the best implementations. On the smaller system *System1* the efficiency that is reached suggested a bad scalability for a larger number of processes. The first conclusion of this is normally, that the communication overhead strongly increases with the number of processes and that this would be the reason for the decreasing efficiency. But a detailed analyzes of the runtime showed that the communication needs no significant time. The needed time for the complete independent calculation of the eigenvectors increases with the number of used cores per processor. This is very good illustrated on the larger system *Orestis* were this phenomenon occurs not until using more processes than processors are available. So the number of processes is not the reason for the increasing runtime (for the computation of one eigenvector) it only depends on the amount of cores, with a shared cache, which are used. So the problem of the parallel implementation is the need of different data for each eigenvector, which can occur conflicts between the, apart from that, independent processes. A very interesting aspect of new architectures is the *Hyper-Threading-Technology*, which can produce a significant improvement of parallel programs. We saw that in all cases and for all versions a better efficiency was achieved by using this new feature. Further tests in this direction would be very interesting on a much larger system which also supports hyper-threading. To find out if any other possibly parallel operations exist in the algorithm, a directed cyclic graph is constructed. More commonly than this is a directed acyclic graph (DAG), but it cannot be constructed for any matrix and block size. A DAG must be constructed individually for each given problem and the resulting graph would be that big that it cannot be illustrated on any normal sized paper. The used graph is based on the operations which are defined by different Blas- and LAPACK-Routines. So further parallelization would be possible if we would look inside the basic operations. One simple idea would be the use of thread parallel Blas, PBlas and ScaLAPACK. The problem of this is the very small runtime of one operation. This was also mentioned in this master thesis by comparing the efficiency of the basic operations for different sizes. The runtime of the basic operations were analysed while a performance model was created which should predict the runtime of the algorithm on different systems by using a few parameters. There we saw that the efficiency of the basic operations strongly increases with growing problem size, especially for small problem sizes. This implies that the parallelization of this functions will not be efficient. The performance model is able to predict the runtime on both systems if a fault tolerance of up to 30% is acceptable.

5.2 Future Work

The scalability of the parallel twisted block factorization is mentioned in the previous chapter, but the important factor accuracy or rather the orthogonality must be mentioned in further researches. The orthogonality can be guaranteed by a reorthogonalisation of the calculated eigenvectors. An improvement could be the correlation between bad orthogonality of the eigenvectors and a small gap between the corresponding eigenvalues, which could result in a reorthogonalisation of small subsets of the eigenvectors. If this would work in all cases, the distribution of the work (eigenvectors) through the processes must be changed adequate to the clusters of the eigenvalues. Another possibility to get orthogonal eigenvectors could be the choice of other starting vectors. This could be very efficient, because it would replace an expensive orthogonalization.
Important aspects of the performance, which are not mentioned in this master thesis, are the efficiency of the different parallelization strategies for a much larger system. There the influence of the multi-core architecture could get smaller, because the communication overhead increases and the idle caused by this could reduce the conflict in the shared cache. Furthermore a larger modern system would be interesting to find out how good the hyper-threading technology is in massive parallel applications and when the overhead caused by the doubled number of processes is larger than the improvement of the CPU feature. On the other hand the turbo boost could result in a

situation where the use of less cores of each processor is more efficient than use all of them.
There is much more to do to get a general useable parallel eigenvector solver which guarantees always accurate results with high efficiency on large systems, but the actual results show that this algorithm is already better than normally used methods for specific problems.

Bibliography

[1] W. N. Gansterer and G. König, "On twisted factorizations of block tridiagonal matrices," *Procedia Computer Science*, vol. 1, no. 1, pp. 279 – 287, 2010, iCCS 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050910000323

[2] G. König, M. Moldaschl, and W. N. Gansterer, "Computing eigenvectors of block tridiagonal matrices based on twisted block factorizations," *Journal of Computational and Applied Mathematics*, vol. In Press, Corrected Proof, pp. –, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0377042711003967

[3] Y. Bai and R. C. Ward, "Parallel block tridiagonalization of real symmetric matrices," *J. Parallel Distrib. Comput.*, vol. 68, pp. 703–715, May 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1360708.1360738

[4] G. König, "Computation of Eigenvectors of Block Tridiagonal Matrices Based on Twisted Factorizations," 2009, masterthesis.

[5] C. Vömel, "ScaLAPACK's MRRR algorithm," *ACM Trans. Math. Softw.*, vol. 37, pp. 1:1–1:35, January 2010. [Online]. Available: http://doi.acm.org/10.1145/1644001.1644002

[6] P. Bientinesi, I. S. Dhillon, and R. A. van de Geijn, "A Parallel Eigensolver for Dense Symmetric Matrices Based on Multiple Relatively Robust Representations," *SIAM J. Sci. Comput.*, vol. 27, pp. 43–66, July 2005. [Online]. Available: http://portal.acm.org/citation.cfm?id=1081198.1081222

[7] F. Tisseur, J. Dongarra, M. Eprint, F. Tisseur, and J. Dongarra, "A parallel divide and conquer algorithm for the symmetric eigenvalue problem on distributed memory architectures," *SIAM J. Sci. Comput*, vol. 20, pp. 2223–2236, 1999.

[8] I. S. Dhillon, B. N. Parlett, and C. Vömel, "The design and implementation of the MRRR algorithm," *ACM Trans. Math. Softw.*, vol. 32, pp. 533–560, December 2006. [Online]. Available: http://doi.acm.org/10.1145/1186785.1186788

[9] J. W. Demmel and I. Dhillon, "On The Correctness Of Some Bisection-Like Parallel Eigenvalue Algorithms In Floating Point Arithmetic," *Electronic Trans. Num. Anal*, vol. 3, pp. 116–149, 1995.

[10] Y. Bai and R. C. Ward, "A parallel symmetric block-tridiagonal divide-and-conquer algorithm," *ACM Trans. Math. Softw.*, vol. 33, August 2007. [Online]. Available: http://doi.acm.org/10.1145/1268776.1268780

[11] Argonne National Laboratory, "Frequently Asked Questions," May 14, 2011. [Online]. Available: http://wiki.mcs.anl.gov/mpich2/index.php/Frequently_Asked_Questions

[12] Sun Microsystems, Inc., "IBM XL C/C++ Enterprise Edition V8.0 for AIX, Compiler Reference," May, 2003, http://www.bluefern.canterbury.ac.nz/UCSC%20userdocs/ForUCSCWebsite/C /AIX/compiler.pdf.

[13] ——, "OpenMP API User's Guide, Sun ONE Studio 8," May, 2003. [Online]. Available: http://www.filibeto.org/sun/lib/development/studio_8/817-0933.pdf

[14] MPI Forum, "MPI: A Message-Passing Interface Standard, Version 2.2," September 4, 2009. [Online]. Available: http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

[15] Intel Corporation, "Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1," May 2011. [Online]. Available: http://download.intel.com/design/processor/manuals/253668.pdf

[16] ——, "Intel Core i7-860 and Core i5-750 Processors for Embedded Computing," 2009. [Online]. Available: http://download.intel.com/newsroom/kits/embedded/pdfs/Core_i7-860_Core_i5-750.pdf

[17] D. Stanley and K. Stanley, "The Performance of Finding Eigenvalues and Eigenvectors of Dense Symmetric Matrices on Distributed Memory Computers," in *In Proceedings of the Seventh SIAM Conference on Parallel Proceesing for Scientific Computing. SIAM.* SIAM, 1994, pp. 528–533.

[18] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan, “PLASMA Users’ Guide, Version 2.0,” November 10, 2009. [Online]. Available: http://icl.cs.utk.edu/projectsfiles/plasma/pdf/users_guide.pdf

Michael Moldaschl, BSc
Eipeldauerstraße 21-25/43/6
1220 Wien
+43 650 8894418
a0607892@unet.univie.ac.at

Curriculum Vitae

Geburtsdatum	14.07.1987
Geburtsort	Wien
Staatsbürgerschaft	Österreich

Schulbildung

- π Master “Scientific Computing” auf der Universität Wien (seit 2009)
- π Bachelor “Physik” auf der Universität Wien (seit 2006)
- π Bachelor of Science auf der Universität Wien in Informatik mit Auszeichnung abgeschlossen (2009)
- π HTL Donaustadt, Abteilung EDV, Matura mit Auszeichnung bestanden (2006)
- π AHS Franklinstraße 26 (1997 – 2001)
- π Volkschule Brioschiweg 3 (1993 – 1997)

Tätigkeiten

since 10/2010	**Projektmitarbeiter**, *Divide and Conquer Algorithm for Eigenvalue computation of block tri-diagonal matrices* und *Twisted Block Factorization for Eigenvector computation of block tri-diagonal matrices* *Universität Wien*
03/2011 – 07/2011	**Tutor,** Algorithmen und Programmierung in Scientific Computing
10/2010 – 02/2011	**Tutor**, Introduction in Scientific Computing - Applications and Algorithms *Universität Wien*
03/2010 – 07/2010	**Tutor**, Algorithms and Programming in Scientific Computing *Universität Wien*
07/2008 – 07/2009	**Projektmitarbeiter**, Exploiting Structure in Complex Symmetric Eigenproblems - Applications in the Numerical Solution of Maxwell's Equations in Optoelectronics Simulations (resulting in Bachelor Thesis) *Universität Wien*
03/2008 – 07/2008	**Tutor**, Algorithms and Data structures *Universität Wien*

Michael Moldaschl, BSc
Eipeldauerstraße 21-25/43/6
1220 Wien
+43 650 8894418
a0607892@unet.univie.ac.at

Auszeichnungen und Zertifikate

- π 2. Platz in „Best of the Best" 2011 in der Kategorie „Bester Bachelor Abschluss Informatik" (Auszeichnung für die besten Studenten)
- π 2. Platz in "Best of the Best" 2007 in der Kategorie "Bachelor Informatik"
- π Best Paper Award 2008 in Informatik auf der Universität Wien für die Arbeit "Quantum Computing"
- π Microsoft Certified Professional (2004): Developing and Implementing Windows-based Applications with Microsoft Visual Basic .Net and Microsoft Visual Studio .Net

Publikationen

- π *Computing eigenvectors of block tridiagonal matrices based on twisted block factorizations*, Journal of Computational and Applied Mathematics, 2011. (http://www.sciencedirect.com/science/article/pii/S0377042711003967)
- π Towards Parallel twisted block factrization, Parallel Numerics, 2011

IT Skills

Operating systems	Linux, Windows
Programming skills	JAVA, MATLAB, C/C++, Visual Basic, VB.NET, SQL, Fortran, XML, PHP, HTML Multicore Programming (MPI, OpenMP) OpenCL (Computing on graphical unit) Software Libraries (Blas, Lapack, Slepc, SBR-Toolbox)
Others	UML, Latex

Wien, September 2011