Computing eigenvalues in parallel

Daniel Kleine-Albers

13.04.2009

Matriculation no. 3218819, TU München.

Written after participation in the Joint Advanced Student School 2009 (JASS09) in St. Petersburg.

Contents

1 Introduction and Motivation

Eigenproblems are of great importance in many application areas. The most important application are the physics problems, quantum mechanics as well as "classic" physics problems in stiffness calculations, load analysis or the calculation of the eigenfrequency for example. But there are also applications in other scientific disciplines, for example efficient compression algorithms in computer science. In this report the focus lies on the calculation of eigenvalues of full, not neccessarily symmetric matrices. However a lot of optimizations can be applied on symmetric matrices which is why they will also be of great interest.

Ever expanding matrix sizes and the fact that single core processor clock speeds are close to their physical limits require the parallelization of the algorithms. Also multi-core systems are standard nowadays not only in the supercomputer area, but also on desktop and laptop computers. Even mobile phones will become multi-core processors soon [2, 1]. Therefore parallel algorithms can be used everywhere. Last but not least having a parallel algorithm also yields greater flexibility - running a parallel algorithm on a single core is no problem, but running a serial algorithm efficiently on a parallel computer is.

This report accompanies the talk held on the same topic at the Joint Advanced Student School 2009 (JASS09) in St. Petersburg on the same topic. First a short introduction into parallelization is given in section 2 describing the basic parallel architectures and the criteria which are required to parallelize an algorithm efficiently. Thereafter the main part (section 3) reviews the algorithms QR iteration, Divide and Conquer as well as the relatively new MRRR algorithm. Also the transformation into compact matrix forms will be shown.

2 Parallelization

2.1 Basic parallel architectures

In principal there are two forms of parallel architectures. In a shared memory system all processors have access to the same memory. Therefore the interprocess communication can be realized easily and efficiently by accessing the same memory areas.

In a distributed memory system every processors (or a group of processors) has its own memory and is interconnected to each other by some kind of communication link, e.g. a bus or a network interface. In these architectures the communication overhead between processors often is the main limiting factor of the parallelization efficiency. Fig. 1 shows the differentiation.

Of course, also hybrid forms are in use nowadays, e.g. a network of multi-core machines.[4]

2.2 Criteria for parallelization

For parallelizing an algorithm some factors have to be considered.

Data locality means the amount of data that is required to perform a certain operation and how close together this data is stored. If the data is stored in adjacent memory positions it can usually be accessed very efficiently as most

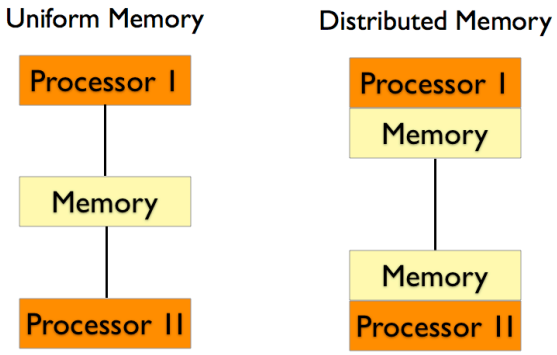

Figure 1: Parallel architectures differentiated by type of memory

architectures are optimized for this type of access. For example a problem where only a limited number of input data is required to calculate a certain amount of output and this data is stored closely together has good data locality. On parallel architectures it is also of importance if an input data set is only used on one processor (so that the data can be local to this processor) or on more of them. In case it is used by a lot of processors the data has to be distributed, thus increasing communication overhead.

Data dependence relates to the fact that in most algorithms there are some operations that require the output of the presceding operations. If a great number of operations can be performed without the need of the previous results it can usually be parallelized quite easily. On the other hand if one can only execute the next step when the step before has finished parallelization is much harder.

Communication overhead is often the limiting factor of parallel efficiency. From a technical point of view the communication overhead is the time required to communicate with other processors or memory locations. Usually communication gets extremely expensive compared to arithmetics as soon as networks are involved. From an algorithmic point of view this is the amount of communication required to let the algorithm run. This is often directly resulting from data locality and data dependency properties. If there is a good data locality and not much data dependency the communication overhead will stay within efficient limits.

Speedup is a measure for the efficiency of parallelization. Speedup shows the increase in processing speed for a certain algorithm compared to the addition of physical clock speed. For example if comparing a single core system to a dual core system (with same clock speeds) an optimal speedup would be 2. This means that the algorithm on the dual core system runs twice as fast as on the corresponding single core system. In this optimal case no overhead for synchronization or communication is required.

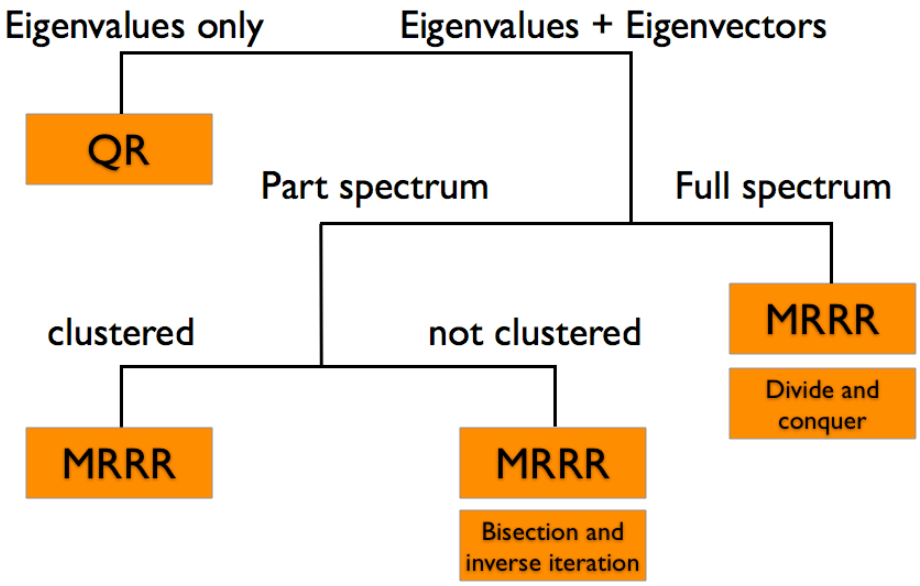

Figure 2: Algorithms Overview (adapted from [12])

3 Algorithms for computing eigenvalues

A variety of different algorithms is available to calculate the eigensolutions. Depending on the requirements a certain algorithm can be chosen. The first decision is if only the eigenvalues or also the eigenvectors should be calculated. Other important points for choosing an algorithm are if all eigenpairs are required or only parts of the spectrum and if the eigenvalues are clustered (small relative distances).

Fig. 2 shows some of the availabe algorithms that are still in use today and suggests when to use them. The QR Iteration, Divide and Conquer as well as MRRR will be described in the following. Bisection and inverse iteration will not be part of this report.

3.1 Naive approach

As the eigenvalues are also the solutions of the characterisc polynomial $det\,(A - \lambda I) = 0$ the most naive approach one could use is to form the characterisc polynomial and use a root finding method on it to obtain the eigenvalues. However the formation and evaluation of the charaterisc polynomial is extremely expensive for all non-trivial cases and can therefore not be applied for bigger matrices. Because of that this approach is not used when calculating eigenvalues with a computer and will not be investigated further.

3.2 QR Iteration

The QR iteration is one of the most common algorithms for calculating eigenvalues. The following lists the basic algorithm (adapted from [6, ch. 6, p. 18]):

```
n=size(A)
```

```
while(n > 1) {
        Factorize A = QR using QR decomposition
        A = RQ
        if (a[n,n-1] < tolerance) {
                output lamba[n] = a[n,n]
                remove row and column n
                n--
        }
}
output lambda[n] = a[n,n]
```

The main work of each step is done in the QR decomposition. The QR decomposition can be found by using Givens rotations on the subdiagonal elements[1]. The Givens rotations are applied column-wise from left to right, top to bottom to not introduce additional non-zero elements. For the algorithm to work, the QR decomposition does not need to be explicitedly calculated. Instead the Givens rotations can be applied in the following way (from left and right) to directly find the required A:

$$A = ... G_2 * \left(G_1 * A * G_1^T\right) * G_2^T ...$$

Also the Givens rotation matrix that only differs in 4 elements from the identity matrix is not calculated explicitedly. Instead its effect is directly calculated on A. As a Givens rotation is required for each subdiagonal, non-zero element each step has $\mathcal{O}\left(n^2\right)$ operations. Therefore the performance of the algorithm greatly depends on the number of non-zero elements in the subdiagonal.

Looking at parallelization the first thing one can notice is that for calculating a Givens rotation the previous rotations need to be applied first (see Figure 3) which makes the algorithm hard to parallelize. However it has been suggested to parallelize the algorithm on an array of processors [11]. Basically the matrix is split up into parts columnwise. Each processor calculates the rotations for its group of columns and passes the results on to the other processors. They apply the received rotations to their elements. The parallelization is designed in such a way that the order will stay consistent. This parallelization requires a lot of communication - each rotation has to be send to all other processors - and therefore is not as efficient as one would like.

3.3 QR Iteration with transformation to compact form

As already mentioned the performance of the QR iteration mainly depends on the number of non-zero entries below the diagonal. This is true also for most other methods. Therefore methods have been found to reduce a matrix to a compact form with most elements below the diagonal equal to zero. These methods all use the fact that eigenvalues stay the same for similar matrices. A transformation that keeps similarity is called a similarity transform and has the general form

$$A = S * A * S^{-1}$$

[1] Alternatively this can be done using Householder rotations. However they are more inefficient when applied to a compact matrix which is why this method is not detailed here.

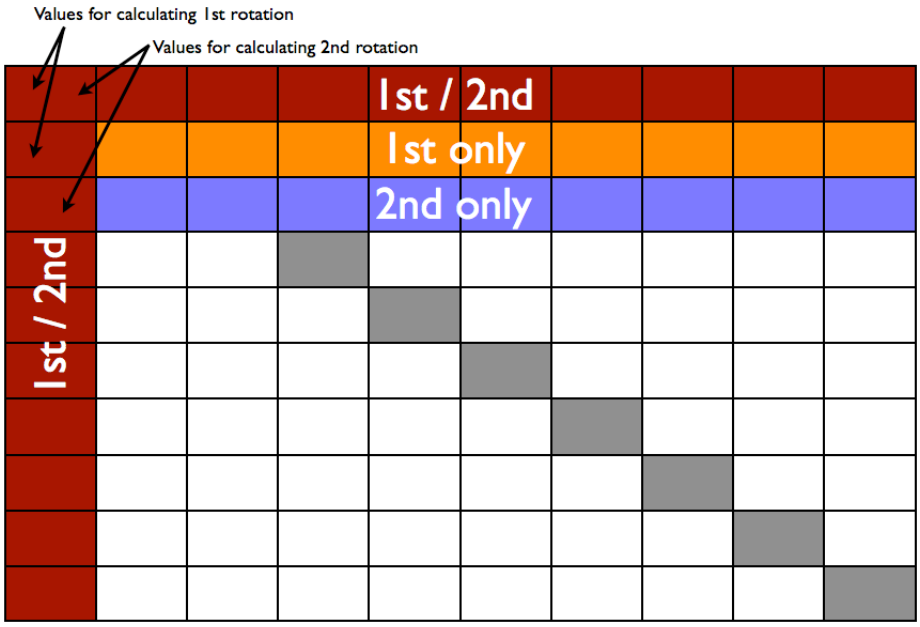

Figure 3: Changed values on 1st and 2nd Givens rotation

If the transformation matrix S is orthogonal (as for example Householder and Givens rotations) the transformation can be simplified to

$$A = S * A * S^T$$

The method that is described here is the Householder reduction. This transformation uses Householder rotations (one per column) to reduce a full matrix to a tridiagonal matrix (if the matrix is symmetric) or an upper Hessenberg type matrix (if the matrix is not symmetric). This results from the fact that similarity transforms keep the symmetricy properties of the input matrix, therefore when reducing a symmetric matrix in such a way that all elements below the lower subdiagonal become zero, also all elements above the upper subdiagonal will become zero.

This leads to the general way of calculating eigenpairs efficiently:

1. Reduce the matrix to compact form
2. Calculate eigenvalues of compact matrix
3. Backtransform the eigenvectors, if required

The last step is required due to the fact that eigenvectors (other than eigenvalues) do not stay the same for similar matrices. Usually the matrix required for the backtransformation can be accumulated during the reduction step or using back accumulation, which is more efficient in some cases. [9, p. 25]

Concerning parallelization the simplest method - applying all Householder rotations one after another - can already be parallelized quite easily as the application consists of matrix-vector operations. Matrix-vector products are

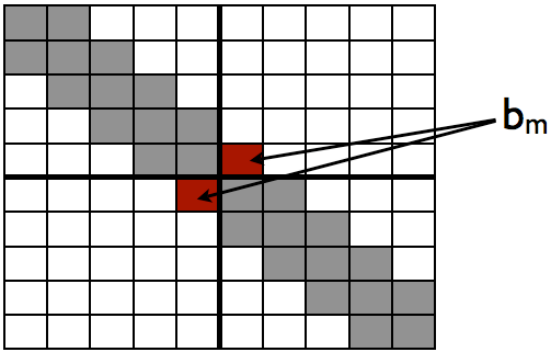

Figure 4: Block structure of tridiagonal matrices (adapted from [10, p. 8])

parallelized by splitting the matrix into several parts (either row or columnwise depending on the order) and letting each processor calculate the resulting values for his part of the matrix and the full input vector. In the end the resulting vector is aggregated.

However in most cases it is better to use blocked Householder transformations as there are more matrix-matrix operations involved which are more efficient in terms of parallelization. In comparison to matrix-vector products with matrix-matrix products each processor can calculate more values of the resulting matrix without the need to interact with other processors. Unfortunately even with blocked Householder transformations a lot of matrix-vector products are involved.

Even more efficient can be two step reduction processes where the matrix is first transformed into a band matrix and then transformed into a tridiagonal matrix (of course only applicable on symmetric matrices). The efficiency gain results from the fact that more matrix-matrix products are involved.

3.4 Divide and Conquer

The divide and conquer method is one of the typical approaches to split work into smaller units in computer sciences. Fortunately this method can also be applied on symmetric, tridiagonal matrices that often appear for eigenvalue problems (after reduction).

A tridiagonal matrix is also almost block tridiagonal with only two elements that need to be treated specially (see fig. 4). The divide and conquer algorithm does this by splitting out this value (both values are the same because of the symmetry) and substracting its absolute value from the top left element (for the lower part) and the bottom right element (for the upper part). The split can therefore be expressed as follows

$$T = \begin{bmatrix} T_1 & \\ & T_2 \end{bmatrix} + |b_m| \, vv^T$$

where T_1 and T_2 are the subparts of matrix T (with modified elements as described before), b_m is the specially treated element and v is a vector that consists of zeros except two elements that are 1 or -1 (depending on b_m being negative or positive).

During the divide operation the algorithm calls itself on both subparts of the matrix. For the bottom of these recursive calls where only a 1x1 matrix is left the eigenvalue is the element in this matrix and the corresponding eigenvector equals to 1.

For the recombinment of the eigenpairs of the submatrices to them of the recombined matrix the seculiar equation is built[2]. Its roots are the eigenvalues and can easily be solved by using zero finders. Having the eigenvalues also the eigenvectors can be calculated with ease. The whole algorithm is as follows (adapted from [10])

```
% T = input matrix, Q = eigenvectors, A = eigenvalues
function [ Q, A ] = dc_eig(T)
        if  T is 1x1
                return Q=1, A=T
        else
                split T to T1, T2
                [Q1, A1] = dc_eig(T1)
                [Q2, A2] = dc_eig(T2)
                based on A1,A2,Q1,Q2 find combined
                        eigenvalues A and eigenvectors Q
                return Q, A
        endif
end
```

The zero-finding runs particularly fast because of an effect called deflation. Deflation means that the eigenvalues of the recombined matrix are often close or even equal to the eigenvalues of the submatrices. Therefore they can be used as an initial guess for the zero finder leading to fast convergence.

For parallelization it is easy to see that the two recursive calls in each step are independent of each other and therefore easy parallelisable. Such an easy parallelisation is sometimes referred to as "embarassingly parallel". The split-up is very flexible in regards to the number of processors used (as long as the number of processors is significantly lower than the matrix dimensions). A drawback of this algorithm is that extra memory is required for the recursive calls.

In practice this algorithm is often combined with QR iteration. For example divide and conquer would be used for splitting up matrices of size with $n > 25$ and as soon as smaller matrix sizes are reached QR and inverse iteration would be used to find the eigenpairs of the smaller matrices. [10, 8]

3.5 MRRR

The last, relatively new algorithm introduced in this report is based on Multiple Relatively Robust Representations and therefore called MRRR. The algorithm uses several LDL^T representations, where L is unit lower triangular and D is diagonal, and can be applied on symmetric, tridiagonal matrices. The factorization of a matrix into L and D is not part of this report but can be found in the respective books. For each cluster of eigenvalues a new representation is found. The new representation is chosen to shift into the cluster of eigenvalues using a suitable τ:

[2]The mathematical details shall not be part of this report. They can be found in [10].

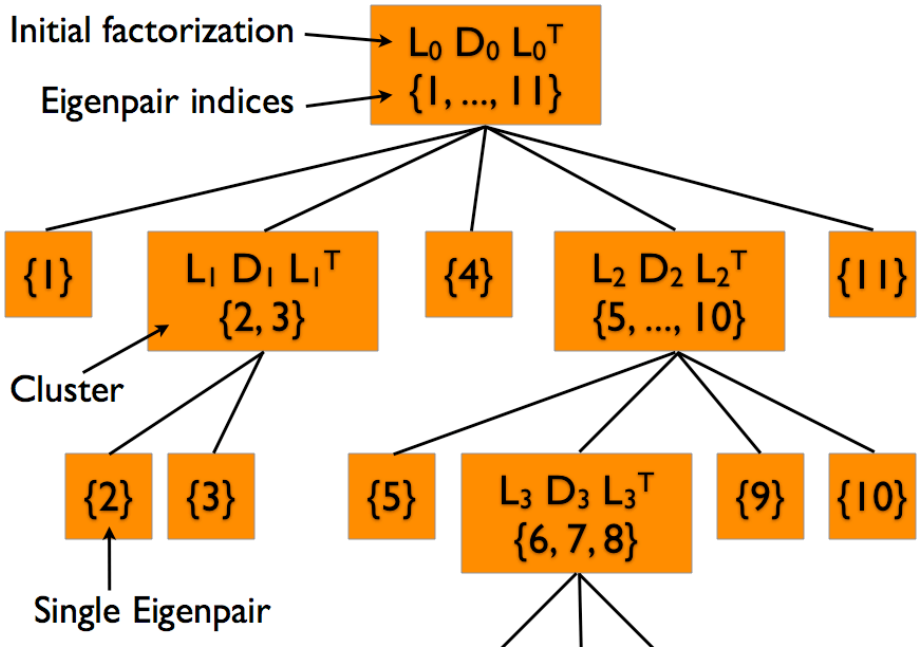

Figure 5: Representation Tree for MRRR (adapted from [7, p. 8])

$$L_C D_C L_C^T = LDL^T - \tau I$$

This means that the eigenvalues need to be "estimated" upfront - one can use any other algorithm for it - a low accuracy is sufficient. The new representation can be seen as a child of its parent representation. In case there are again several clusters of eigenvalues in this representation, new child representations need to be found. Therefore a representation tree can be built up to visualize this (see fig. 5).

The MRRR algorithm has many unique features. First its complexity is only $\mathcal{O}(nk)$ with k being the number of requested eigenpairs. It can be used to calculate only parts of the eigenspectrum, which saves additional time, if not all eigenpairs are required. Another nice property is that the calculated eigenvectors will be orthogonal without reorthogonalization and the results are very accurate.

In terms of parallisation MRRR also shows good behaviour. Each cluster of eigenpairs can be calculated on a different processor indepedently of each other. The serial version of the algorithm uses a queue to put each cluster in, which are then calculated one after another. Parallelism can be reached easily by letting all idle processors get elements from the queue and do the calculations. As no reorthogonalization is required the results can immediately be used and there's no need to wait until all requested eigenpairs are calculated. [7, 5, 3]

4 Conclusion

A variety of algorithms is available to calculate eigenvalues and eigenpairs. In this report QR iteration, Divide and Conquer as well as MRRR have been introduced.

When looking at QR iteration it's easy to see that the number of non-zero elements below the diagonal is the main factor for performance, which is why the matrix is reduced to a compact form first using Householder reduction to let QR iteration run efficiently. The same reduction process is also used for all other algorithms.

In the Divide and Conquer algorithm a typical approach to split the problem into smaller units is used. These smaller problems can be parallelised very easy.

Last but not least the modern MRRR algorithm was introduced which provides some unique features. It can be used on parts of the spectrum, provides very high accuracy and the eigenvectors are orthogonal to each other without reorthogonalization.

Finally I'd like to thank everyone who made JASS09 in St. Petersburg possible. It was a great experience - from a scientific point of view but also personally.

References

[1] ARM, editor. Arm cortex-a9 mpcore [online]. 2009. Available from: `http://www.arm.com/products/CPUs/ARMCortex-A9_MPCore.html` [cited 2009-04-10].

[2] Multi-core smartphones! (not the iphone, yet) [online]. 02 2009. Available from: `http://phone.click2creation.com/index.php/2009/02/multi-core-smartphones-not-the-iphone-yet/` [cited 2009-04-10].

[3] Dominic Antonelli and Christof Vömel. Lapack working note 168: Pdsyevr. scalapack's parallel mrrr algorithm for the symmetric eigenvalue problem. Technical report, UC Berkeley, 2005.

[4] Blaise Barney. Introduction to parallel computing [online]. 01 2009. Available from: `https://computing.llnl.gov/tutorials/parallel_comp/` [cited 2009-04-09].

[5] Paolo Bientinesi, Inderjit S. Dhillon, and Robert A. van de Geijn. A parallel eigensolver for dense symmetric matrices based on multiple relatively robust representations. *SIAM Journal for Scientific Computing*, 27(1):43–66, 08 2005.

[6] Prof. Dr. Hans-Joachim Bungartz. Skript numerical programming i. Technical report, TU München, 2008.

[7] Inderjit S. Dhillon and Beresford N. Parlett. Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices. *Linear Algebra and its Applications*, (387):1–28, 2004.

[8] Kevin Gates and Peter Arbenz. Parallel divide and conquer algorithms for the symmetric tridiagonal eigenproblem. Technical report, Eidgenössische Technische Hochschule Zürich, 10 1994.

[9] Bruno Lang. *Effiziente Orthogonaltransformationen bei der Eigen- und Singulärwertzerlegung*. Bergische Universität GH Wuppertal, 1997.

[10] Jeffery Rutter. A serial implementation of cuppen's divide and conquer algorithm for the symmetric eigenvalue problem. Technical report, UC Berkeley, 02 1994.

[11] G. W. Stewart. A parallel implementation of the qr algorithm. *Parallel Computing*, (5):187–196, 1987.

[12] Adrian Tate. Xt3 optimization - scientific libraries, 08 2006.