Multithreading ART: Comparison

Fábio Bento and Paul Crocker
`m5135@ubi.pt, crocker@di.ubi.pt`

Department of Computer Science, University of Beira Interior
6200-001 Covilhã, Portugal

DCS UBI Technical Report
Number DCS UBI-2012-01
December 2012

Department of Computer Science
University of Beira Interior
6200-001Covilhã
PORTUGAL
PHN: (351) 275-319891
FAX: (206) 275-319899
URL: http://www.di.ubi.pt

Multithreading ART: Comparison

Fábio Bento and Paul Crocker
m5135@ubi.pt, crocker@di.ubi.pt

Department of Computer Science, University of Beira Interior
6200-001 Covilhã, Portugal

Technical Report DCS-2012-01
December 2012

Abstract

Algebraic Reconstruction Techniques (ART) may be be used to reconstruct water vapour images in real-time using data obtained from the Global Navigation Satellite System (GNSS). To achieve this requirement is necessary to parallelize the ART algorithms. This report offers a brief introduction to these algorithms and their parallelization. An algebra and algebraic reconstruction libraries have been written for this purpose and used with several multithreading libraries. The results presented in this work shows that the more efficient multithread SIRT methods tested are about 7 times faster than the sequential implementations using a machine with 12 cores for processing.

1 Introduction

Algebraic Reconstruction Techniques (ART) are iterative algorithms that been used with success in medical imaging applications for many years. These techniques allow image reconstruction using data obtained from a set of projections. The data retrieved for the image reconstruction can be obtained from diverse sources such as X-ray, magnetic resonance imaging and seismic travel time. This technical report is motivated by the Algebraic Reconstruction Techniques used in water vapour image reconstruction. The water vapour image can be reconstructed using the diverse slant delays of a GNNS station in relation to its satellites. The image reconstructed may then be used then for weather forecasting or other meteorological applications.

The need for the parallelization arises from the requisite that in some applications the images need to be reconstructed quickly even in real-time. The parallelization should reduce reconstruction times of the ART techniques which should allow the periodic reconstruction.

The objective of this report is to detail and summarise the various ART method and to show and discuss the results of the work with the parallelization of ART methods. This will serve as a fundamental basis when determining the best parallelization technique in future work under the planned scenarios.

The rest of this document is organized as follows. In the next section Algebraic Reconstruction and respective techniques are introduced, in the third section it will be presented the Algebra and Algebraic Reconstruction libraries created, in the fourth section the Multi-threading libraries namely OpenMP (OMP) and Intel Threading Building Blocks (TBB) are discussed. The fifth section presents the implementation of the diverse libraries used to parallelize the techniques including the OMP, TBB and Eigen3 for matrix operations. In the sixth section the results obtained using the diverse parallelization libraries and optimizations are given. The seven and final section concludes this report.

2 Algebraic Reconstruction

In this section the concept of Algebraic Reconstruction will be introduced and the main algorithms described.

Algebraic Reconstruction is an approach for imaging reconstruction using data obtained from a series of projections such as those obtained from electron microscopy, x-ray photography and in medical imaging like in computed axial tomography (CAT scans), that consists of obtaining data from cross sections of an object from measurements taken from different angular positions around the object and then solving for an array of unknowns that represent the interior of the object being analysed. In Figure 1 we can see 6 line projections from three angular positions around an unknown object. For medical applications Algebraic Reconstruction techniques lack the accuracy and speed of implementation when compared to other methods. However there are situations where is not possible to measure a sufficiently large enough number of projections or when the projections are not uniformly distributed over 180 or 360 degrees, which prevents the use of other techniques such as transform based techniques that can obtain high accuracy.

Algebraic techniques are also useful when energy propagation paths between the source and receiver positions are subject to ray bending on account of retraction or when energy propagation undergoes attenuation along ray paths [1].

In the algebraic techniques presented in this report it is essential to determine the ray paths that connect the corresponding transmitter and receiver positions. When refraction and diffraction effects are substantial (medium inhomogeneities exceed 10% of the average background value and the correlation length of these inhomogeneities is comparable to a wavelength) it becomes impossible to predict ray paths which ends in obtaining meaningless results [1].

2.1 Image and projection representation

Consider a two dimensional image $I = f(x, y)$. Figure 1 shows a square grid superimposed onto this image, we want to obtain this image from the projection data, shown as straight lines which traverse the (x, y) plane. We assume that for each cell $f(x, y)$ is a constant and let $f_j : j = 1..N$ be the constant values for each cell of the image. N is the total number of cells in the grid. A line integral will be defined as array-sum. This ray sum is referred as p_i, where i is the i-ray.

The relationship between f_j and p_i's is expressed as:

$$\sum_{j=1}^{N} w_{ij} f_j = p_i, \qquad i = 1, 2, \cdots, M \tag{1}$$

M is the total number of rays (all projections), w_{ij} is the weight of the contribution of the jth cell to the ith ray integral (this is proportional to fraction of the jth cell intercepted by the ith ray as shown in Figure 1).

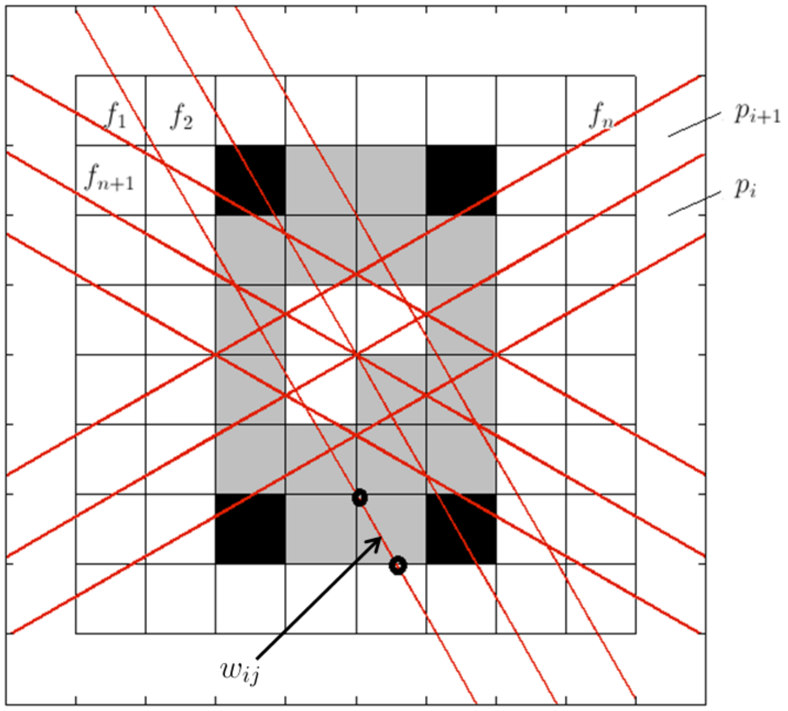


Figure 1: Unknown image on square grid. Each cell value is a unknown variable to be determined using the various projections.

Depending on the context many of the w_{ij} may be zero because only a small number of the w_{ij} contribute to each ray-sum.

If M and N were small one could use traditional matrix theory methods to invert the equations system in (1). In practice N may be large, for example for 256 x 256 image N would be 65.000. If M had more or less the same value then for these values the size of the matrix $[w_{ij}]$ in (1) would be 65.000 x 65.000 which basically rules out any chance of direct matrix inversion [1].

Also when noise is present in the measurement data and $M < N$, even for relatively small N it is not possible to use direct matrix inversion. In this case one can use use a least square method to obtain an approximate solution, however when M and N are large these methods are computationally impracticable [1].

There are however some very attractive methods for solving these equations. These methods are based on the ”method of projections” which Kaczmarz first proposed [2]. First the equation (1) will be expanded to explain the computational procedure of these methods:

$$
\begin{aligned}
w_{11}f_1 + w_{12}f_2 + w_{13}f_3 + \cdots + w_{1N}f_N &= p_1 \\
w_{21}f_1 + w_{22}f_2 + w_{23}f_3 + \cdots + w_{2N}f_N &= p_2 \\
&\vdots \\
w_{M1}f_1 + w_{M2}f_2 + w_{M3}f_3 + \cdots + w_{MN}f_N &= p_M
\end{aligned}
\tag{2}
$$

A grid with N cells gives an image with N degrees of freedom. The image represented by $(f_1, f_2, \cdots, f_N)$ can be seen as a single solution in an N-dimensional space.

Each of the equations in (2) represents a hyperplane. When there is only one solution to the equations it's represented as a single point, namely the intersection of all the hyperplanes. This is the main concept that's illustrated in figure 2. In this figure we have only considered two variables f_1 andf_2 which satisfy the follow equations:

$$
\begin{aligned}
w_{11}f_1 + w_{12}f_2 &= p_1 \\
w_{21}f_1 + w_{22}f_2 &= p_2
\end{aligned}
\tag{3}
$$

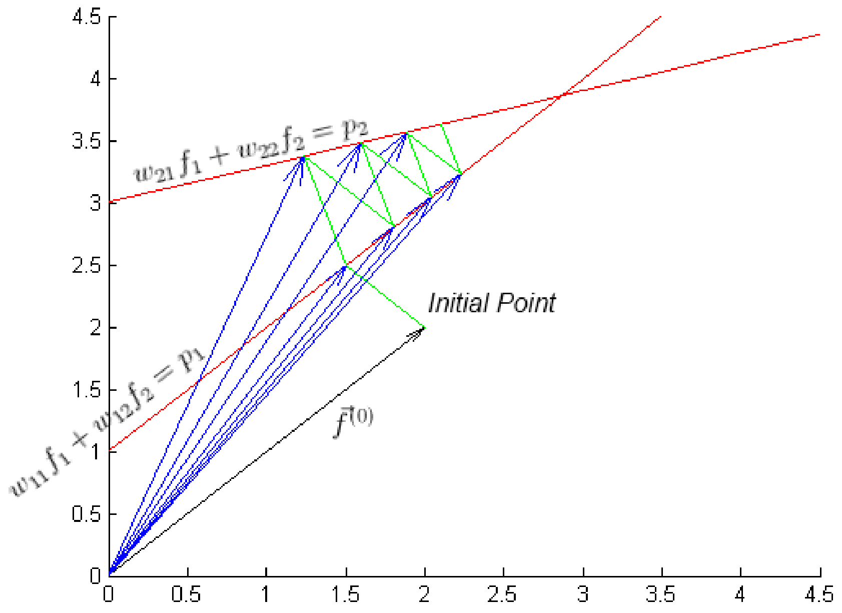


Figure 2: Kaczmarz method illustrated for two unknowns.

The computational procedure to calculate the solution is the following:

1. Start with a initial guess

2. Project the initial guess onto the first line

3. Reproject the resulting point from 2 to the second line (2)

4. Reproject the result point from 3 to the first line and so on

5. If there's a unique solution the algorithm will converge.

The initial guess is written as$(f_1^{(0)}, f_2^{(0)}, \cdots, f_N^{(0)})$ or simply by the vector $\vec{f}^{(0)}$. Often the initial vector is simply a zero vector. This vector is then projected onto the hyperplane using the first equation on (2) resulting in the $\vec{f}^{(1)}$ vector. This can be seen in figure 2 for a two dimensional space. After that $\vec{f}^{(1)}$ is projected by the second equation on (2) on the hyperplane resulting on $\vec{f}^{(2)}$ and so forth. In 2 the projections are the green lines and the blue vectors represent the next estimates of the solution.

When $\vec{f}^{(i-1)}$ is projected on the hyperplane represented by the ith equation it can be mathematically be described as:

$$\vec{f}^{(i)} = \vec{f}^{(i-1)} - \frac{\vec{f}^{(i-1)} \cdot \vec{w}_i - p_i}{\vec{w}_i \cdot \vec{w}_i} \vec{w}_i \tag{4}$$

Where $\vec{w}_i = (w_{i1}, w_{i2}, \cdots, w_{iN})$ and $\vec{w}_i \cdot \vec{w}_i$ is the dot product of the vector $\vec{w}_i$ by itself.

The (4) equation is known as the Kaczmarz method.

Regarding the algorithms convergence, it's easily seen that in the case of two perpendicular hyperplanes then for any initial guess in the (f_1, f_2) plane it is possible to find the solution in only two steps using (4). However if the two hyperplanes have a reduced angle between them there will be a greater number of iterations (depending also on the initial guess).

In fact if the M hyperplanes in (2) could all be made orthogonal (perpendicular) with respect to one another then the solution could be found in only one pass through all the (2) equations (assuming that only one solution exists) [1].

This is theoretically possible using for example a method for orthonormalising a set of vectors such as the Gram-Schmidt procedure. However in practice it isn't computationally viable as the orthnomalizing process itself takes too much time. Another problem with orthogonalization is that it amplifies the noise problem from the measurements into the final solution [1].

If we have $M > N$ in (2) and the projections have been corrupted by noise, no unique solution exists, although a solution in a zone may still be determined.

Figure 3 shows a two variable system with three noisy hyperplanes. In this case we see the result after projecting the initial point onto the first line (in green) and then iterating 100 times, the figure show the projections onto the hyperplanes. As can now be seen the procedure in (4) doesn't converge to a unique solution but instead it oscillates in the neighbourhood of the intersections of the hyperplanes.

In the case of $M < N$ a unique solution also doesn't exist, instead there are multiple solutions. If we have only one equation of the two in (3) to calculate the two variables, then the solution can be any point in the line that corresponds to this equation.

Another advantage of this method is the possibility of adding a priori information already known about the image being reconstructed in order to guide the solution. If we know for example that the image contains no negative values and if during the iterative process we obtain we some negative value one can simply change that value to zero.

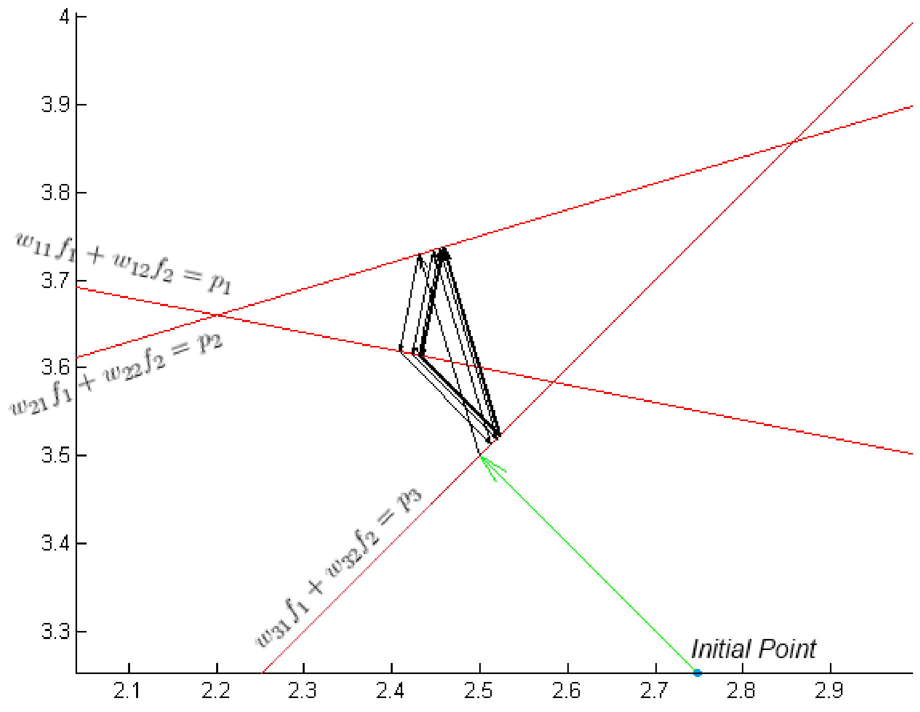


Figure 3: Case where the number of equations is greater than the number of unknowns and projections have been corrupted by noise.

2.2 Techniques

There are two different techniques for algebraic reconstruction, namely: Algebraic Reconstruction Techniques (ART) and Simultaneous Iterative Reconstruction Techniques (SIRT) which will be described in the follow subsections.

Different methods of these techniques namely Kaczmarz, Symmetric Kaczmarz, Landweber and SART which were used for making the tests on this technical report will be now presented.

2.2.1 ART methods

The Algebraic Reconstruction Techniques (ART) are row-action methods that treat the equations one at time. This means that in each iteration each equation is solved individually [3]. The updates in each iteration are made using the equation on (2) plus a relaxation parameter λ_k resulting on:

$$\vec{f}^{(i)} = \vec{f}^{(i-1)} + \lambda_k \frac{p_i - \vec{f}^{(i-1)} \cdot \vec{w}_i}{\vec{w}_i \cdot \vec{w}_i} \vec{w}_i \tag{5}$$

The ART reconstructions usually suffer from so called ”salt and pepper” noise, which is caused by the approximation of the various w_{ij} measured [1].

Kaczmarz

This is the most well know ART method in the literature [4] [5]. It uses a fixed $\lambda_k = \lambda \in (0, 2)$ in the original paper the value 1 was used. In the literature this method is also referred as ART which can give rise to some confusion since it is also used for Algebraic Reconstruction Techniques. It consists of one ”sweep” in each row of A from the top to the bottom: $i = 1, 2, ..., m$.

Symmetric Kaczmarz

This method is a variant of the previous Kaczmarz method. It adds a new ”sweep” using the rows in the reverse order. As result one iteration consists of 2 steps: $i = 1, 2, ..., m-1, m, m-1, ..., 3, 2$.

This method supports both a fixed ($\lambda \in (0, 2)$) and iteration-dependent λ_k.

2.2.2 SIRT methods

The Simultaneous Iterative Reconstruction Techniques (SIRT) are ”simultaneous” because all equations are solved at the same time in one iteration (based on matrix multiplications) [3]. Usually these methods converge slower to the solution than the ART methods, however they result in better looking images [1].

The SIRT methods work by first solving all the equations and only at the end of each iteration are the cell values updated. The change of each cell is the average of each of the changes for that cell [1].

The general form of these methods is as follows:

$$\vec{f}^{(i)} = \vec{f}^{(i-1)} + \lambda_k T A^T M (p_i - A\vec{f}^{(i-1)}), \qquad k = 0, 1, 2, \cdots \tag{6}$$

Where A is the matrix with the various projections, λ_k is a relation parameter and the matrices M and T are symmetric positive definite. The various SIRT methods depend on these matrices. λ_k is defined as $2/\rho(TA^T MA) - \epsilon$. The ρ is the spectral radius, ϵ the machine's epsilon and A^T is the matrix transpose.

Landweber

The Landweber method is described by the following form:

$$\vec{f}^{(i)} = \vec{f}^{(i-1)} + \lambda_k A^T (p_i - A\vec{f}^{(i-1)}), \qquad k = 0, 1, 2, \cdots$$

which corresponds to replacing $M = T = 1$ in (6).

SART

These methods were originally implemented as ART methods [6], but can also be implemented as SIRT method. They are described by the following :

$$\vec{f}^{(i)} = \vec{f}^{(i-1)} + \lambda_k D_r^{-1} A^T D_c^{-1}(p_i - A\vec{f}^{(i-1)}), \qquad k = 0, 1, 2, \cdots$$

D_r^{-1} and D_c^{-1} are diagonal matrices corresponding respectively to the row and column sums:

$$D_r = diag(||\vec{w}_i||_1) \qquad D_c = diag(||\vec{w}_j||_1).$$

where $\vec{w}_j = (w_{1j}, w_{2j}, \cdots, w_{Nj})$.

There's no need to include weights in this method as the convergence for this method was established in [7] [8] and it was shown that $\rho(D_r^{-1} A^T D_c^{-1} A) = 1$.

3 Algebra and Algebraic Reconstruction Libraries

For this work a computational linear algebra library was written with some basic functions like matrix-matrix multiplication, vector-matrix multiplication, scalar-vector division, matrix creation, vector creation. An ART and SIRT library was also written with the diverse algorithms used in Algebraic Reconstruction Techniques including: kaczmarz, symmetric kaczmarz, randomized kaczmarz, landweber, cimmino, component averaging, diagonally relaxed orthogonal projections and simultaneous algebraic reconstruction technique. Not all these algorithms have been used for the results of this report but they were all implemented.

The language used to write these libraries was C++. C++ provides some advantages over other languages such as being an object-oriented programming which is well suited for complex systems [9]. It's also a compiled language which means it's usually faster than the interpreted and JIT-compiled languages. It's an open ISO-standardized language. Is portable, existing diverse compilers for the various operating systems. It's compatible with C code, which means that we can write C code with few or no modifications plus user external C libraries. C++ have also a great library support (including multithreading) with over 3000 results in the project management website SourceForge [10].

Because of all these advantages this was the language which was chosen to write the libraries and the tests discussed in this technical report.

In order to illustarte show what kind of libraries have been produced the headers of the Linear Algebra and Algebraic Reconstruction algorithms are shown below:

```
struct mVector{...}
struct mMatrix{...}
class Math {
public:
        Math();
        virtual ~Math();

        static vector<int> retreiveNonZeroRows(mMatrix *myMatrix);
        static void retreiveSjVector(mMatrix *A, mVector *result);
        static mVector* scalarVectorSum(mVector *V, double scalar);
        static mVector* scalarVectorMultiplication(mVector *V, double scalar);
        static mVector* scalarVectorDivision(double scalar, mVector *V);
        static mMatrix* scalarMatrixMultiplication(mMatrix *M, double scalar);
        static mMatrix* scalarMatrixDivision(mMatrix *M, double scalar);
        static mMatrix* matrixMatrixMultiplication(mMatrix *M1, mMatrix *M2);
        static mVector* matrixVectorMultiplication(mMatrix *M, mVector *V);
        static void vectorSum(mVector *V1, mVector *V2, mVector *result);
        static mVector* vectorSubtraction(mVector *V1, mVector *V2);
        static void vectorMultiplication(mVector *V1, mVector *V2, mVector *result);
        static void cumulativeVector(mVector* myVector, mVector* result);
        static void transposeMatrix(mMatrix *M,mMatrix *result);
        static void normalizeVector(mVector *V, mVector *result);
        static mVector* powerMethod(mMatrix *A);
        static void diagMatrix(mMatrix *M, mMatrix *result);
        static void diagVector(mVector *V, mMatrix *result);
        static void showMatrix(mMatrix *A);
        static void showVector(mVector *V);
        static double level1Norm(mVector *myVector);
        static double euclidianNorm(mVector *myVector);
        static double euclidianNorm(mMatrix *myMatrix);
        static double dotProduct(mVector *firstVector, mVector *secondVector);
        static double getMaxOfVector(mVector *A);
        static double getMaxAbsoluteOfVector(mVector *V);
        static double spectralRadius(mMatrix *myMatrix);

};
```

```
class Algorithms {
public:
        Algorithms();
        //ART
        static mVector* kaczmarz(mMatrix *A, mVector *b, int numInterations);
        static mVector* kaczmarzSymmetric(mMatrix *A, mVector *b, int numInterations);
        static mVector* kaczmarzRandomized(mMatrix *A, mVector *b, int numInterations);
        //SIRT
        static mVector* landweber(mMatrix *A, mVector *b, int numInterations);
        static mVector* cimmino(mMatrix *A, mVector *b, int numInterations);
        static mVector* cav(mMatrix *A, mVector *b, int numInterations);
        static mVector* drop(mMatrix *A, mVector *b, int numInterations);
        static mVector* sart(mMatrix *A, mVector *b, int numInterations);
        virtual ~Algorithms();
private:
        static void retreiveSjVector(mMatrix *A, mVector *result);
        static void createEucNormVector(mVector *V, mVector *result);

};
```

A simple example of creating two matrices and multiplying them with this library is the following:

```
mMatrix *A = new mMatrix(3,3);
A->setMatrixToRand();
mMatrix *B = new mMatrix(3,3);
B->setMatrixToRand();

mMatrix *result;

result=Math::matrixMatrixMultiplication(A,B);
```

These libraries can be later extended and optimized as is needed.

4 Multi-threading Libraries

There are several multithreading libraries for C++. The most elemental approach is to use a low threading library such as Pthreads (Posix Threads) or Boost, common higher level libraries are OpenMP and Intel Threading Building Blocks (TBB) [11]. This work aims to use and then compare the popular OpenMP and TBB libraries, since they provides a high level of abstraction of the implementation and platform plus they are known to be stable [12].

4.1 OpenMP

OpenMP (OMP) is a collection of compiler directives, library routines and environment variables for shared-memory parallelism in C, C++ and Fortran programs. It is portable across different shared memory from different architectures. OMP is managed by the non-profit organization OpenMP Architecture Review Board whose members include the following organizations: AMD, Fujitsu, HP, IBM, Intel, Microsoft, NEC, NVIDIA, Oracle Corporation, Texas Instruments and others [13]. The OMP API requires that the programmer explicitly write the actions to be taken to execute the program in parallel while aiding the coding when compared to lower level API's, by introducing a high level of abstraction which doesn't make it necessary to specify the action of each individual thread, it is not an automatically parallel programming model. OMP doesn't check for data dependencies therefore conflicts such as race conditions, deadlocks and other multithreading problems may still occur [14]. It is up to the programmer to ensure correctness and deal with these specific problems. OMP supplies primitives for locks and other multithreading primitives. OMP is included in many compilers by default including (but not limited to) GCC, XL C/C++ / Fortran, Visual Studio 2008-2010 C++ and nagfor [15].

An example of the OMP library is the following extracted code from the multiplication algorithm:

```
#pragma omp parallel for
for (int i=0; i<M1->rows-1;i+=2){
        int k;
        register double s00,s10;
        s00=s10=0.0;
        for (k=0;k<M2->rows;k++){
                s00 += M1->theMatrix[i][k] * v1[k];
                s10 += M1->theMatrix[i+1][k] * v1[k];
        }
        result->theMatrix[i][j]=s00;
        result->theMatrix[i+1][j]=s10;
}
```

Notice the *#pragma omp parallel for* directive which automatically parallelize the for loop using the library.

4.2 Intel Threading Building Blocks

Intel Threading Building Blocks (TBB) is another library that helps the programmer write parallel programs that take advantage of multicore processors that is also portable and scalable. Using TBB the programmer may build general purpose parallel programs and doesn't require any special languages or compilers. It is based on the use of C++ templates aiming in this way to be flexible yet efficient. TBB contains parallel algorithms and data structures it also provides scalable memory allocation and task scheduling [16].

It fully supports nested parallelism which allow the programmer create larger parallel components from smaller components. The library uses tasks instead of threads and it internally maps the tasks onto threads in a efficient way [17].

TBB according to [17], was built with performance in mind, allowing computational intensive work parallelization. TBB doesn't address all threading problems however it is compatible with other thread packages (for instance its possible to write mixed OMP and TBB code).

An example of implemented code is given below which implements the same vector-matrix multiplication algorithm as was used for the OMP example (the details of dealing with the cases when the number of rows is odd are not shown):

```
mMatrix *M;
mVector *V,*result;

void operator()( const blocked_range<int>& range ) const {

        int i,j;
        register double s00,s01;

        for (i=range.begin(); i < range.end(); i+=2){
                s00=s01=0.0;
                for (j=0; j<M->columns;j++){
                        s00 += V->theVector[j] * M->theMatrix[i][j];
                        s01 += V->theVector[j] * M->theMatrix[i+1][j];
                }
                result->theVector[i] =s00;
                result->theVector[i+1] =s01;
        }

        //..case row number is not an even number
}

mVector* Math::matrixVectorMultiplication(mMatrix *M, mVector *V){

        mVector *result=new mVector(M->rows);
        int i,j;
        ITBvectorMatrixMultiplication myVMmultiplication;

        myVMmultiplication.result=result;
        myVMmultiplication.M=M;
        myVMmultiplication.V=V;

        parallel_for( blocked_range<int>(0, M->rows-1), myVMmultiplication );

        return result;
}
```

Notice the need of creating a new struct which is used for parallelizing the same loop as used in the OMP directive. This struct is then initialized in the main function and all variables it needs are then assigned. A call to the *parallel_for* function is made in the end so the for is executed using threads.

5 Parallelizing using OMP, TBB and Eigen3

In this section we start by describing the parallelization of the basic building blocks of the algebraic reconstruction algorithms, namely the operations: matrix-matrix multiplication and vector-matrix multiplication. A diverse set of multithreading libraries was used for performance analysis. The libraries tested were OMP, TBB and Eigen3 which is a free open source library that automatically includes OMP parallelizing techniques. In this section the algorithms which were parallelized plus the methods used from the respective libraries will be described. The results obtained will be analysed in the following section.

The matrix-matrix algorithm pseudo-code is presented as follows :

```
Result <- Zero
for (j = 0; j<=M2->columnsNumber-1; j+=2)
        for (i = 0; i<=M2->rows; i++)
                Store M2->column(i ) in Vector1
                Store M2->column(i+1) in Vector2
        for (i = 0; i<=M1->rowsNumber-1; i+=2)
                for (k = 0; k<=M2->rowsNumber; k++)
                        Result(i,j) = Result(i,j) + M1(i,k) * Vector1(k)
                        Result(i,j+1) = Result(i,j+1) + M1(i,k) * Vector2(k)
                        Result(i+1,j) = Result(i+1,j) + M1(i+1,k) * Vector1(k)
                        Result(i+1,j+1) = Result(i+1,j+1) + M1(i+1,k) * Vector2(k)

        //..case row number is not an even number

//..case columns number is not an even number

return Result
```

Where M1 and M2 are the first and second matrices respectively and Result the result matrix of the multiplication. The details of dealing with the cases when the number of rows or columns is odd are not shown.

The procedure can be basically described by the follow steps:

1. For each two columns in M2
    - 1.1. Store the two columns in two vectors
    - 1.2. For each row in M1
        - 1.2.1. Multiply each element of the row by each element of the vector 1 and vector 2
        - 1.2.2. Store the result of the multiplications in the result matrix

The matrix-vector algorithm pseudo-code is presented as follow:

```
for (i = 0; i<=M−>rows−1; i+=2)
        for (j = 0; j<=M−>columns; j++)
                Result(i) = Result(i) + V(j) * M(i,j)
                Result(i+1) = Result(i+1) + V(j) * M(i+1,j)

if (M−>rows%2 is 1) then
        set i to M−>rows−1
        for (j = 0; j<=M−>columns; j++)
                Result(i) = Result(i) + V(j) * M(i,j)

return Result
```

Where M is the matrix, V the vector and Result the matrix of the multiplication.
The procedure can be basically described by the follow steps:

1. For each two rows in M

   1.1. Store the two rows in two vectors

   1.2. Multiply each element of the vector V by each element of the vector 1 and vector 2

   1.3. Store the result of the multiplications in the result vector

Some optimizations were made to these algorithms. This include the explicitly use of the register via the register keyword for the sums of the multiplications, since this avoids accessing main memory every time the sum the result of the multiplications. An example of this register optimization is the following:

```
        register double s00;
        s00=0.0;
        for (k=0;k<M2−>rows;k++){
                s00 += M1−>theMatrix[i] [k] * v1[k];
        }
        result−>theMatrix[i][j] =s00;
```

Notice the use of the *register* keyword in the multiplication accumulator.

Another optimization made to the algorithm was to use more than one instruction per loop (known as loop unwinding). For example instead of making one iteration per loop make two. An example is show below:

```
        for (j=0; j<M−>columns;j++){
                s00 += V−>theVector[j] * M−>theMatrix[i][j];
                s01 += V−>theVector[j] * M−>theMatrix[i+1][j];
        }
```

Notice that in only one iteration two operations are executed (in this case the multiplication of two columns by a vector). This technique increases the program speed by eliminating the instructions that control the loop, it also reduces the delay from reading data from the memory [18] [19].

The last optimization was to divide the matrices into vectors (known as block structure). Since vectors can be much smaller than a matrix they can fit or partially fit into cache memory which may decrease the time to retrieve their elements. This also makes the multiplications faster [20].

In the matrix multiplication one of the matrices were divided into columns vectors while in the vector multiplication the matrix were divided into rows vectors. An example this optimization is below:

```
for (i=0; i<M2->rows;i++){
        v1[i]=M2->theMatrix[i][j];
}

for (i=0; i<M1->rows-1;i+=2){
        register double s00,s10;
        s00=s10=0.0;
        for (k=0;k<M2->rows;k++){
                s00 += M1->theMatrix[i] [k] * v1[k];
                s10 += M1->theMatrix[i+1][k] * v1[k];
        }
        result->theMatrix[i] [j] =s00;
        result->theMatrix[i+1][j] =s10;
}
```

In this case it caches one of the columns of the matrices.

This optimizations were implemented in all algorithms including parallel algorithms and the sequential one.

In next subsections the specific implementations of the various parallelization libraries namely OMP and TBB will be explained. The Eigen3 library which includes OMP parallelization shall also be discussed.

5.1 Parallelization OMP

The OMP parallelization was by far the simplest to implement. For both cases (matrix-matrix and matrix-vector multiplication) the OMP directive *"#pragma omp parallel"* and *"#pragma omp for"* were used. The *"#pragma omp parallel"* directive initializes a parallel section which means all threads will execute the code in it. In this case it was used to create the vectors. The *"#pragma omp for"* directive distributes the number of threads available in a for loop. In the matrix-matrix multiplication algorithm was also used the *"#pragma omp parallel for"* directive which do the same of the other two directives discussed combined. This means that it automatically parallelizes a for section and distributes the available threads on it [14]. By default the number of threads used in the Visual C++ implementation of OMP is the same as the number of processor cores detected by the Operating System (including virtual processors, including hyperthreading CPUs) [21].

In the matrix-matrix multiplication algorithm the directive *"#pragma omp parallel"* was used for each thread have independent vectors. The *"#pragma omp for"* directive was used in the main matrix-matrix multiplication for cycle. This can been checked in the follow extracted code:

```
#pragma omp parallel
{
        double *v1=new double[M2->rows];
        double *v2=new double[M2->rows];

#pragma omp for
        for (j=0; j < M2->columns-1; j+=2){
                int i,k;
                for (i=0; i<M2->rows;i++) {
                        v1[i]=M2->theMatrix[i][j];
                        v2[i]=M2->theMatrix[i][j+1];
                }

                for (i=0; i<M1->rows-1;i+=2){
                        register double s00,s01,s10,s11;
                        s00=s01=s10=s11=0.0;
                        for (k=0;k<M2->rows;k++){
                                s00 += M1->theMatrix[i] [k] * v1[k];
                                s01 += M1->theMatrix[i] [k] * v2[k];
                                s10 += M1->theMatrix[i+1][k] * v1[k];
                                s11 += M1->theMatrix[i+1][k] * v2[k];
                        }
                        result->theMatrix[i] [j] =s00;
                        result->theMatrix[i] [j+1] =s01;
                        result->theMatrix[i+1][j] =s10;
                        result->theMatrix[i+1][j+1] =s11;
                }
        ...
        }
...
}
```

Besides this directives the *"#pragma omp parallel for"* directive was also used in the matrix-matrix multiplication. It was used for the last column processing (in the case of the matrix isn't divisible by 2), since in it isn't used any vector. These vectors are pointers which seem to cause problems if used inside of a for with the *"#pragma omp parallel for"* directive. The use of *"#pragma omp parallel for"* can be checked in the follow extracted code:

```
#pragma omp parallel for
for (int i=0; i<M1->rows-1;i+=2){
        int k;
        register double s00,s10;
        s00=s10=0.0;
        for (k=0;k<M2->rows;k++){
                s00 += M1->theMatrix[i] [k] * v1[k];
                s10 += M1->theMatrix[i+1][k] * v1[k];
        }
        result->theMatrix[i] [j] =s00;
        result->theMatrix[i+1][j] =s10;
}
```

In the vector-matrix multiplication the same process with *"#pragma omp parallel"* and *"#pragma omp for"* directives was used in the main loop how can be checked in the follow code:

```
#pragma omp parallel
{
        int i,j;
        register double s00,s01;

#pragma omp for
for (i=0; i < (M−>rows−1); i+=2){
        s00=s01=0.0;
        for (j=0; j<M−>columns; j++){
                s00 += V−>theVector[j] * M−>theMatrix[i][j];
                s01 += V−>theVector[j] * M−>theMatrix[i+1][j];
        }
        result−>theVector[i] =s00;
        result−>theVector[i+1] =s01;
}
```

Note that the original code was not modified, the only code added was the OMP pragma directives so OMP know what to parallelize and how.

5.2 Parallelization TBB

The TBB parallelization required much more modifications than the OMP case. The parallelization was applied exactly in the same loops of the OMP, however there was the need of creating a new struct for every new parallelized block of code. TBB allows these functions to be declared both in structs and in classes leaving the choice to the programmer [22], in this case the struct option was chosen.

For the matrix-matrix multiplication two structs were created. One for the multiplication of two columns simultaneously by two rows and another for the multiplication of only one column by two rows.

In the following code the implementation of the struct of only one column is shown:

```
        struct ITBmatrixMultiplication1C {
                mMatrix *M1, *M2, *result;
                double *v1;
                int j;

                void operator()( const blocked_range<int>& range ) const {

                        for( int i=range.begin(); i<range.end(); i+=2 ){
                                register double s00,s10;
                                s00=s10=0.0;
                                for (int k=0;k<M2−>rows;k++){
                                        s00 += M1−>theMatrix[i] [k] * v1[k];
                                        s10 += M1−>theMatrix[i+1][k] * v1[k];
                                }
                                result−>theMatrix[i] [j] =s00;
                                result−>theMatrix[i+1][j] =s10;
                        }
                }
        };
```

The *void operator()* is the function that gets called by the TBB library to be parallelized. The variables declared in the struct are variables that the function *void operator()* needs in order to work properly. For example in this code we need the matrices we are going to multiply plus a result matrix to store the calculations. The majority of the variables are pointers because it's much more efficient to pass big objects via reference than via copy. The object function calls *range.begin()* and *range.end()* contain the begin of initializer and conditional expressions, these parameters are calculated internally by the TBB library plus the parameters passed to the struct.

Besides these modifications it's also necessary to initialize the structs created. This is done with the following code:

```
ITBmatrixMultiplication1C my1CMultiplication;

my1CMultiplication.result=result;
my1CMultiplication.j=j;
my1CMultiplication.M1=M1;
my1CMultiplication.M2=M2;
my1CMultiplication.v1=v1;

parallel_for( blocked_range<int>(0, M1->rows-1), my1CMultiplication );
```

The first line declares and initialize the struct. The variables necessary to the struct are then assigned. In the end the parallel code is executed using the function call *parallel_for*. This function receives an object of the type *blocked_range* which receives the initializer for the loop condition, the for loop conditional expression and the struct that contains the code to be executed.

The call to the *parallel_for* must be made in the same place where the code of the struct is in the sequential program. As stated before the *range.begin()* and *range.end()* are calculated by the library for each thread using the *blocked_range* parameters especially defined for the loop.

Vector-matrix multiplication was parallelized using this method. It uses a struct for the main for loop which receives the matrix, vector and result vector. Then the structure is initialized and parallelized using a *parallel_for* call.

5.3 Parallelization Eigen3

The Eigen3 library in contrast with the OMP and TBB libraries isn't a library specialized in parallelization. It is described as a "C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms" [23].

The ART methods were implemented again using this library for this reason instead of using our math/algebra library.

This library implements both BLAS/LAPACK. BLAS has a complete implementation while LAPACK has a partial implementation [24].

Eigen is versatile: supports all matrix sizes, standard numeric types, various matrix decompositions and geometry features, includes user modules that provides many specialized features like matrix functions and polynomial solver.

Eigen is fast: includes explicit vectorization support performed for SSE 2/3/4, ARM NEON and AltiVec instructions. Fixed-size matrix are optimized

Eigen is reliable: algorithms are selected for reliability, it's tested with its own test suit, with the BLAS test suite and also with parts of the LAPACK test suite.

Eigen is elegant: the API is clean and expressive looking more natural to C++ programmers. It's very easy to implement an algorithm on top of Eigen.

Eigen has a great compiler support: many tests were run to guarantee compiler support plus work around any compiler bugs. It's in conformity with the C++98 standard [23].

This library supports two different type of storages for matrices: row-major and column-major. Row-major means that the matrix is stored row by row, the first row is stored first, followed by second, the third and so on. Column-major in other side stores the matrix column by column, first stores the first column, then the second, the third and so on. By default the matrices store the data with column-major [25].

A illustrative example of the column-major and row-major is the follow.

Being A the matrix:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Storing A in the row-major order in memory it will be organized as follow:

1 2 3 4 5 6 7 8 9

However storing A with the column-major order in memory will organize it as:

1 4 7 2 5 8 3 6 9

An important feature also present in the Eigen3 library is the support for parallelization. For this it uses the OMP library internally and automatically having only the need to activate the OMP library in the project. Unfortunately this support is limited. Eigen3 only currently supports the parallelization of general matrix-matrix products [26].

Using Eigen the code for implementing the matrix-matrix multiplication is very simply. An example is shown below:

```
MatrixXd a(2700,2500);
a.setRandom();

MatrixXd b(2500,2700);
b.setRandom();

MatrixXd result = a*b;
```

In this example we create two matrices one with 2700 x 2500 dimensions and another with 2500 x 2700, then we set their values to random. To execute the multiplication of both matrices we only need to use the (overloaded) operator *. The result is then saved in the result matrix. All the matrices used in this example consist of the type *MatrixXd* which means that their element type is *double*.

Please note that all these three libraries are free to use even on commercial projects [23] [27].

6 Results

In this section the results obtained from the parallelization discussed before will be presented. Note that what is being tested is only the performance between each implementation, the ART/SIRT algorithms convergence are not tested. There will be tested four implementations namely a sequential implementation, an OpenMP implementation, an Intel Threading Building Blocks implementation and an Eigen3 implementation. For the ART methods only three implementations: sequential, Eigen3 Column-major and Eigen3 Row-major will be tested. When not specified in the Eigen3 tests, is is assumed that the column-major order was used. All these tests were made on two different computers whose specifications are given in the table 1.

Table 1: Computers Specifications.

	Computer 1	Computer 2
CPU Name	Intel Core i7-3610QM	Six-Core AMD Opteron(tm) Processor 2435
CPU Speed	2,30 GHz	2,60 GHz
CPU Number Cores	4	6
CPU Number Logical Processors	8	6
Number CPUs	1	2
Memory Ram	6 GB	16 GB

The specifications for each test were the follow:
Matrix-matrix multiplication:

- Matrix A with the size of 2700 x 2500, Matrix B with the size of 2500 x 2700.
- The results were obtained after calculating an average of 50 multiplications.

Matrix-vector multiplication:

- Matrix with the size of 7700 x 7000.
- Vector with the size of 7000.
- The results were obtained after calculating an average of 50 multiplications.

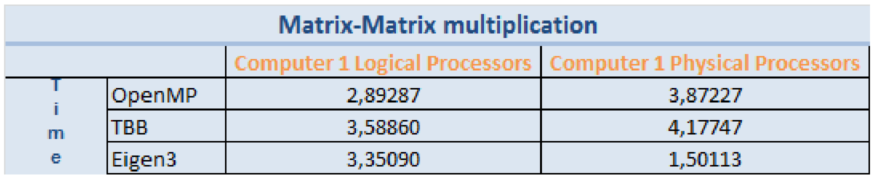
SIRT Landweber, SIRT SART, ART Kaczmarz and ART Symmetric Kaczmarz:

- Matrix with the size of 2700 x 2500.
- Vector with the size of 2700.
- The number of iterations for each algorithm was 50.
- The results were obtained after calculating the average of 50 algorithm calls.

Note that all SIRT implementations (including Eigen3) calculate the spectral radius using the power method since it's very efficient in terms of performance. All implementations have the SSE2 instructions activated in the compiler. The number of threads used was the same as the number of the physical processors detected. This decision was made because it was detected in various tests that the Eigen3 performance was significantly altered if the number of logical processors were used instead of the number of physical processors. An example of this effect can be seen in the table 2 where it is possible to see the matrix-matrix multiplication using the logical processors and physical processors for the parallel implementations. Note that the number of logical processors in computer 2 is the same as the number of physical processors while in the computer 1 the number of logical processors is the double of the physical processors (hyper-threading technology).

In the table 2 it is possible to see that the Eigen3 implementation on logical processors in the computer 1 isn't the best one in terms of performance due to the use of logical processors (the same was also detected in other tests not included here). However using only the physical processors the Eigen3 version is the fastest with more than 2 seconds of difference between the other implementations.

Table 2: Matrix-matrix multiplication in the same computer with logical and physical processors.



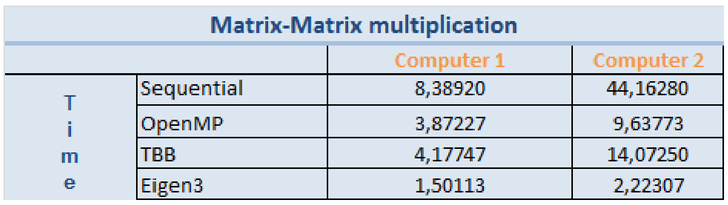
Matrix-Matrix multiplication			
		Computer 1 Logical Processors	Computer 1 Physical Processors
Time	OpenMP	2,89287	3,87227
	TBB	3,58860	4,17747
	Eigen3	3,35090	1,50113

In the next subsections the various tests including matrix-matrix multiplication, matrix-vector multiplication, SIRT Landweber, SIRT SART, ART Kaczmarz and ART Symmetric Kaczmarz will be presented.

6.1 Matrix-matrix multiplication

The results of the matrix-matrix multiplication are displayed below in the table 3 and in the figure 4. The Eigen3 library automatically parallelizes the multiplication of the matrices being only necessary to activate OpenMP in the compiler.

Table 3: Matrix-matrix multiplication.



Matrix-Matrix multiplication			
		Computer 1	Computer 2
Time	Sequential	8,38920	44,16280
	OpenMP	3,87227	9,63773
	TBB	4,17747	14,07250
	Eigen3	1,50113	2,22307

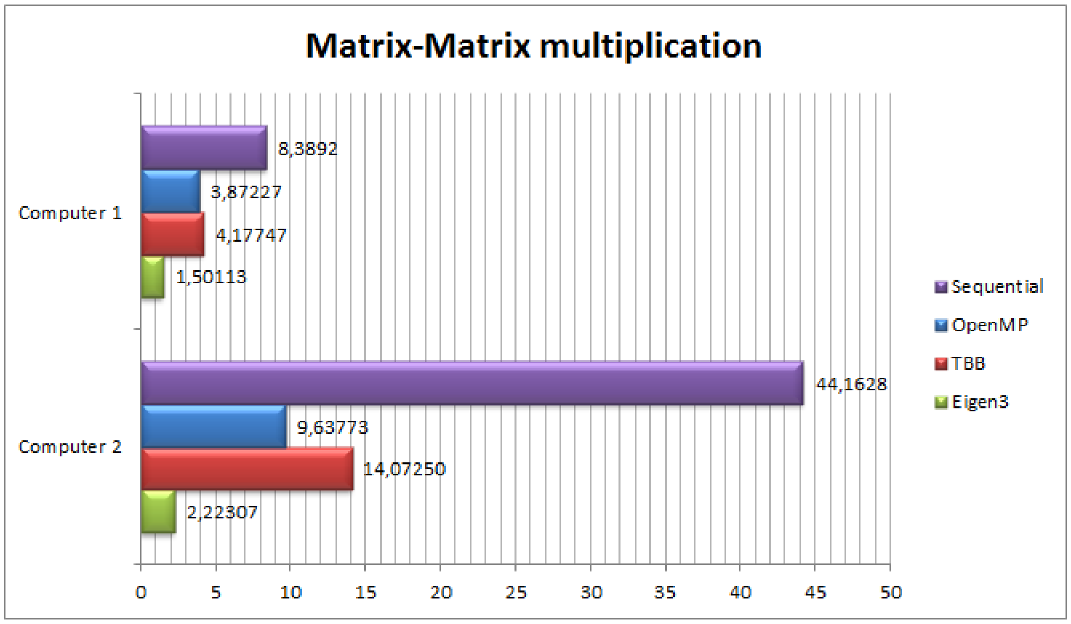


Figure 4: Matrix-matrix multiplication.

In this test is already visible that the computer 1 is faster in all implementations than the computer 2. It's also visible a big difference between the sequential implementation and the parallel versions. This difference allow us to conclude that the parallelization of matrices is essential for a fast execution of the Algebraic Reconstruction algorithms, more specifically the SIRT algorithms. The OMP version is a bit faster than the TBB implementation. The faster implementation is the Eigen3, reaching up to twelve seconds of difference to the TBB implementation in computer 2.

6.2 Matrix-vector multiplication

The results of the matrix-vector multiplication are displayed below in the table 4 and in the figure 5. Note that the Eigen3 library don't make any parallelization of this multiplication although being a sequential implementation it contains some optimizations.

Table 4: Matrix-vector multiplication.

Matrix-Vector multiplication			
		Computer 1	Computer 2
Time	Sequential	0,03406	0,21326
	OpenMP	0,02633	0,06573
	TBB	0,02580	0,06360
	Eigen3	0,03313	0,11387

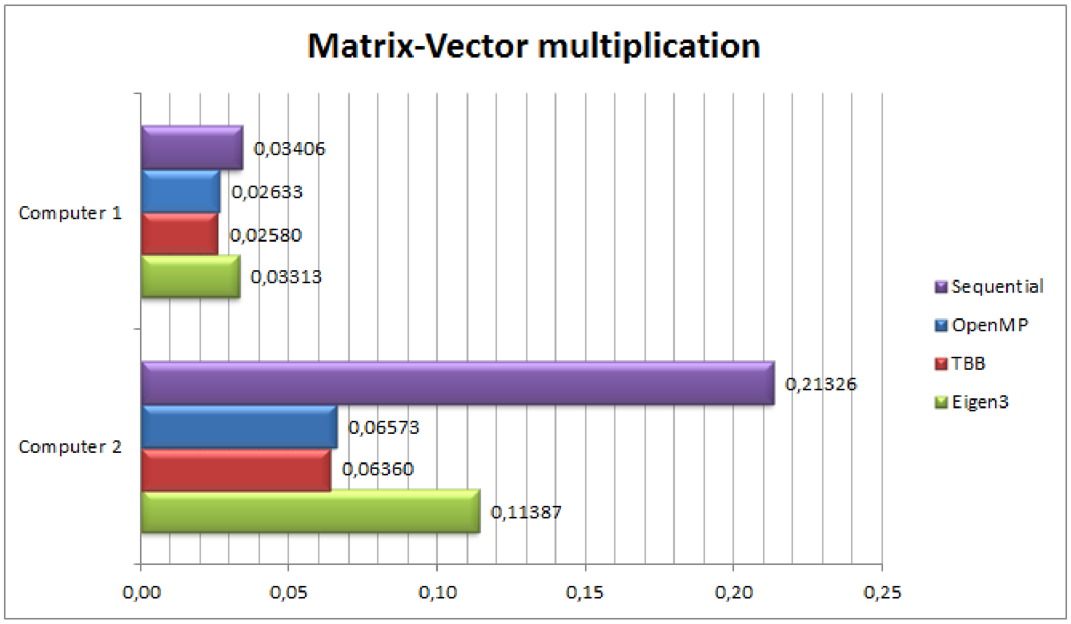


Figure 5: Matrix-vector multiplication.

Analysing the results it is possible to verify that the parallel versions are more efficient than the sequential and Eigen3 versions. The difference between the parallel versions (OMP and TBB) are minimal. The Eigen3 implementation is slower in this test (than the parallel implementations) as it is not parallelized, which indicates that perhaps it would be appropriate to parallelize this library matrix-vector multiplication. Also it's important to note that even though the difference between the sequential version and the parallel versions may be significant (case of computer 2) this multiplications doesn't have so much importance on the overall performance as the matrix-matrix multiplication. For example in the matrix-vector multiplication there aren't any implementation which reaches one second (even with a matrix and vector much bigger than in matrix-matrix multiplication test), on the other hand in the matrix-matrix multiplication even with smaller matrices sizes the execution time reaches up to 44 seconds in the sequential version of computer 2 and the faster parallel version of the computer 2 takes more than two seconds.

6.3 SIRT Landweber

The matrix-matrix and matrix-vector implementations were used in the implementations of the algorithms ART and SIRT. In this case the method SIRT Landweber will be presented. The results of the Landweber method are displayed below in the table 5 and in the figure 6.

Table 5: Landweber method.

SIRT Landweber			
		Computer 1	Computer 2
Time	Sequential	8,40140	45,52840
	OpenMP	3,93527	10,74650
	TBB	4,38747	16,28730
	Eigen3	2,14413	6,41347

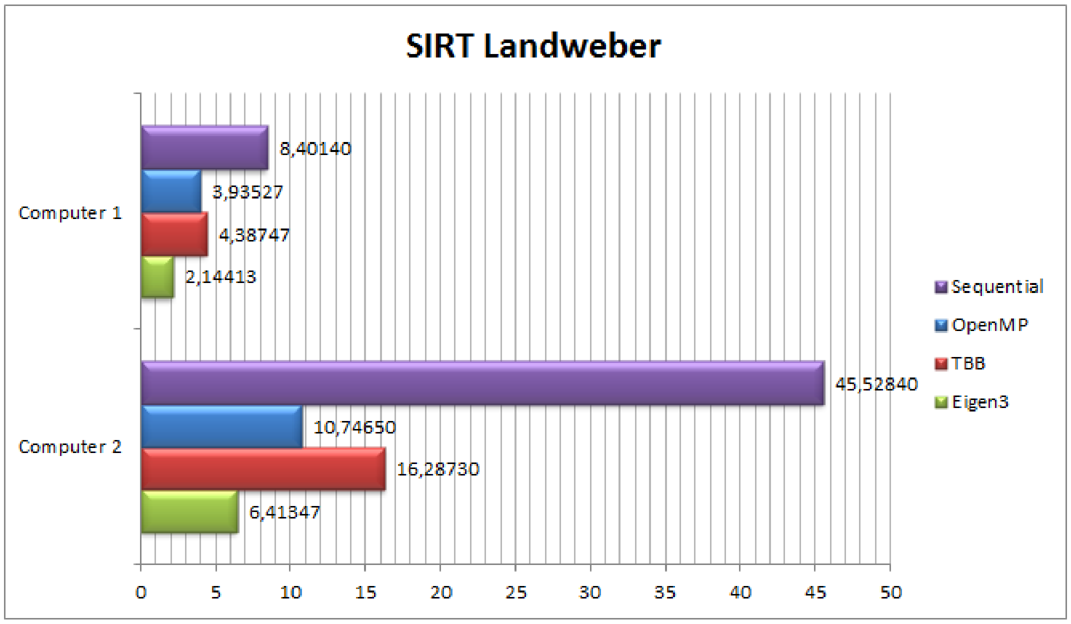


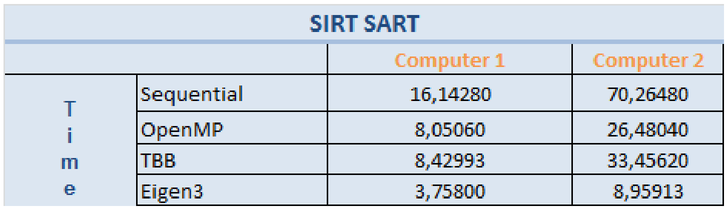
Figure 6: Landweber method.

At the first check it is possible to notice a big difference between the sequential version and the others. This can be explained because the matrix-matrix multiplication contained in this algorithm are extremely slow, the matrix-vector multiplications are also included but have obviously less impact. Also it can be seen that the difference between the parallel versions and the Eigen3 version which occurred in the matrix multiplications tests (especially in computer 2) are not so obvious here. This fact is due to SIRT methods that contain diverse multiplications of matrix-vector. In this specific case the algorithm is executed with 50 iterations where each iteration contains two matrix-vector multiplications summing a total of 100 multiplications. The matrix-matrix multiplication case only occurs one time for the calculation of the spectral radius.

6.4 SIRT SART

The results of the SART method are displayed below in the table 6 and in the figure 7.

Table 6: SART method.



SIRT SART

		Computer 1	Computer 2
Time	Sequential	16,14280	70,26480
	OpenMP	8,05060	26,48040
	TBB	8,42993	33,45620
	Eigen3	3,75800	8,95913

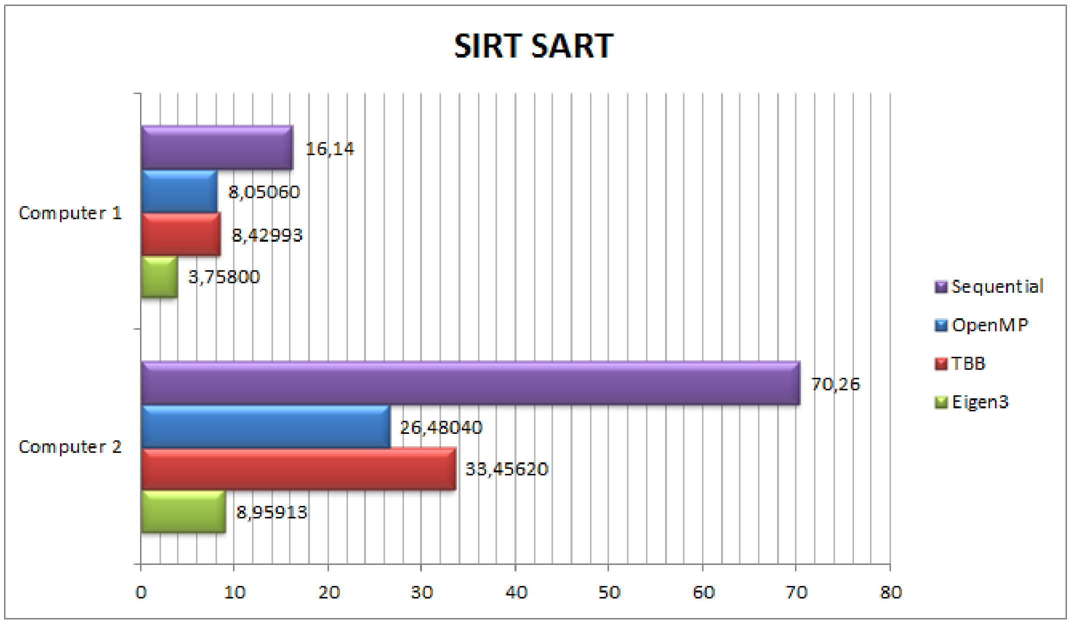


Figure 7: SART method.

The SART algorithm presents similar results when compared to the Landweber algorithm. SART takes however some more time to execute than the Landweber. The difference between the sequential version and the others are still high (approximately the double comparing to the slower parallel implementation). This difference can be explained because there exists one additional multiplication of matrices in this algorithm, plus it's necessary to calculate two more matrices for the multiplications. These two matrices are D_r^{-1} and D_c^{-1} described in the second section. Like Landweber it also contains two multiplications of matrix-vector per iteration.

6.5 ART Kaczmarz

Regarding the ART algorithms only the sequential and Eigen3 implementations are tested. This's because these algorithms don't include matrix-matrix and matrix-vector multiplications which were what was parallelized. There were tested two different Eigen3 methods: column-major and row-major which were discussed in the fifth section of this report. This two different versions were tested because each one have very different performance results in this ART methods. The results of the Kaczmarz method are displayed below in the table 7 and in the figure 8.

Table 7: Kaczmarz method.

ART Kaczmarz			
		Computer 1	Computer 2
Time	Sequential	1,35727	3,22750
	Eigen3 - ColMajor	3,84413	8,45700
	Eigen3 - RowMajor	0,91287	2,69847

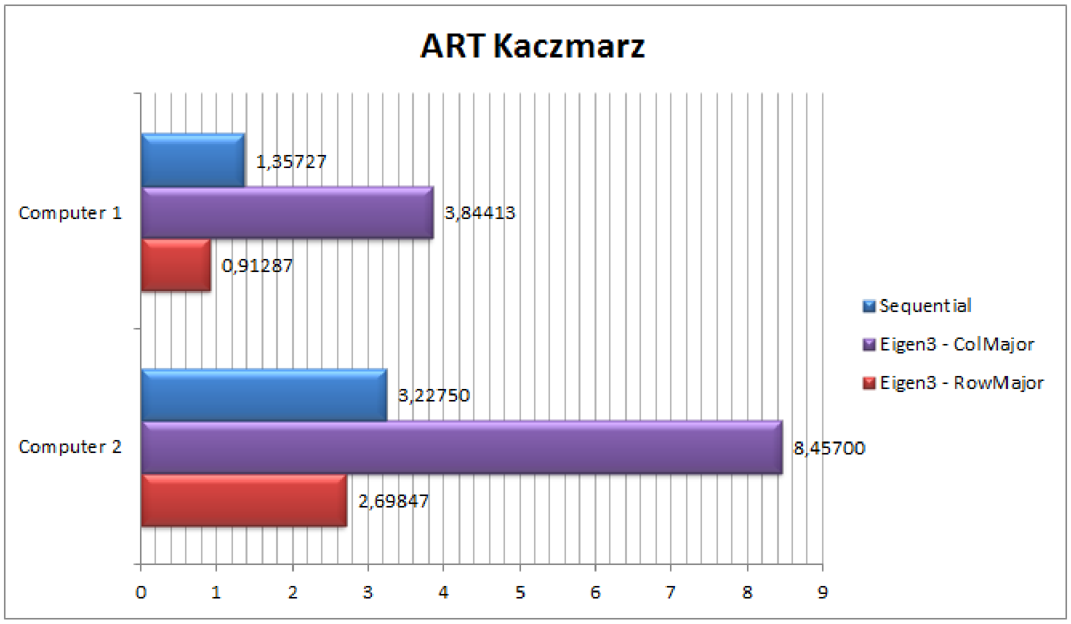


Figure 8: Kaczmarz method.

The results presented confirm the big difference between each of the Eigen3 implementations. All the ART methods make use of the diverse rows of the input matrix in the various iterations. How can be checked the row-major is the more efficient because the matrix is organized sequentially in the memory by rows, being the access to the rows sequential. The sequential implementation performance is similar to the row-major this because in its implementation the matrix is also organized in the memory by rows. The version column-major is much more slower than the row-major. Since the matrix in this implementation is organized in memory by columns the access to each row is not sequential.

6.6 ART Symmetric Kaczmarz

The results of the Kaczmarz Symmetric method are displayed below in the table 8 and in the figure 9.

Table 8: Symmetric Kaczmarz method.

ART Symmetric Kaczmarz			
		Computer 1	Computer 2
Time	Sequential	2,74693	6,77100
	Eigen3 - ColMajor	7,57620	16,92770
	Eigen3 - RowMajor	1,75220	5,66247

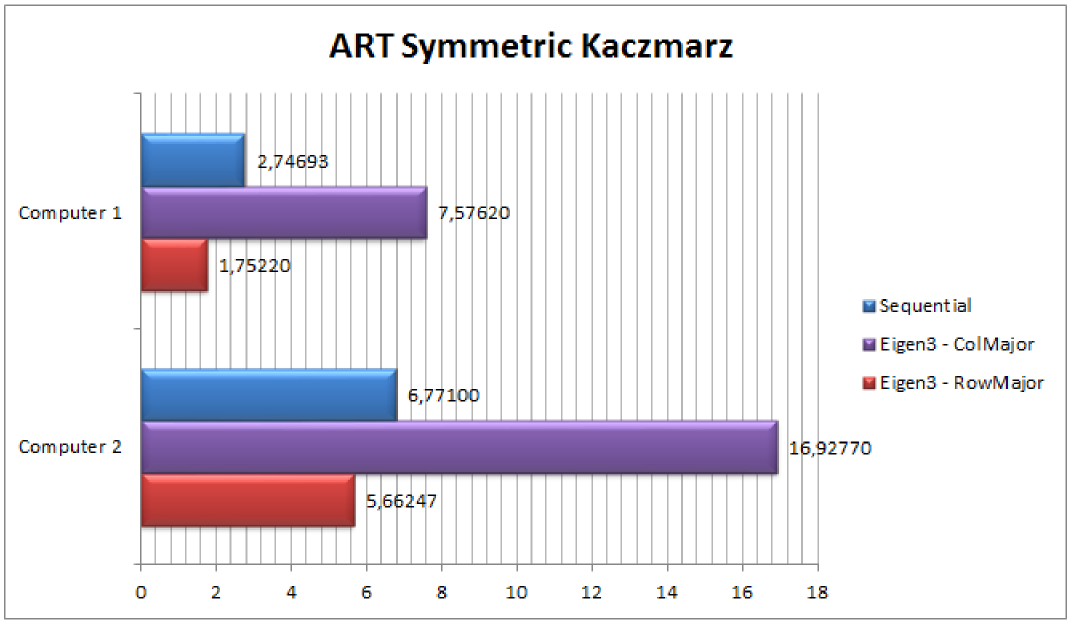


Figure 9: Symmetric Kaczmarz method.

The symmetric version of kaczmarz method is identical to the previous one. However it takes approximately the double of the time in all implementations because of the ”double sweep” which this algorithm implements.

7 Conclusion

The main objective of this report which was to introduce the Algebraic Reconstruction techniques and to compare the performance of different multithreading libraries has been attained. In this report three libraries have been studied and used in the implementations of the ART methods namely: OpenMP, Intel Threading Building Blocks and Eigen3.

The OpenMP library was very simple to use (using its pragma directives) and is included by default in a high number of compilers. Plus it's portable across different hardware and software architectures. However it may be a little hard to debug, plus it doesn't check for data dependencies. The Intel Threading Building Blocks was also relatively simple to implement however it requires more steps than the OMP. Because it uses structures and classes it may be harder to maintain than the pragma oriented OMP.

Eigen3 revealed itself as an excellent algebra library. Although it isn't a multithreading library *per se* it supports internally the matrix-matrix multiplication parallelization using OMP. It also wraps the BLAS and LAPACK libraries which are well known for their good performance [28] [29]. In the tests executed it was proven that Eigen3 is the more efficient library losing only the test of matrix-vector multiplication. This library allows one to write very simple pieces of code for algebra operations thanks to its heavy use of C++ templates, in contrast to BLAS/LAPACK.

In the performed tests it was possible to observe that parallelization was essential for a fast execution of the SIRT methods, this is due to their use of matrix-matrix and matrix-vector multiplications. In the Landweber and SART methods it was possible to observe that the faster multithread implementation is about seven times faster than the sequential implementation. The matrix-matrix multiplications however was observed to be the main bottleneck of SIRT methods with matrix-vector multiplications having much less impact in execution time.

With the tests it was also observed that the Eigen3 is the most efficient library losing only the matrix-vector multiplication to the parallel libraries. This is a strong reason to think in implementing matrix-vector parallelization with this library in the future. Applying the same matrix-matrix multiplication algorithm in Eigen3 with logical cores vs physical cores was proven to be very different. The performance of Eigen3 parallel multiplication seems to degrade very much if the number of threads is greater than the number of physical cores.

The TBB implementation on the various tests was proven to be always slightly slower than the OMP implementation, the only exception was the matrix-vector multiplication but the difference was minimal.

In the ART tests it was possible to verify what was already expected from the Symmetric Kaczmarz method: it took the double of the time than Kaczmarz method. The cause is the "double sweep" implemented by the Symmetric Kaczmarz method. It was also possible to observe that the ART methods are very dependent on the order of the matrix in memory. The Eigen3 row-major and Eigen3 column-major are completely different in terms of results. The Eigen3 row-major is much faster than the column-major. This difference reached about eleven seconds in the kaczmarz symmetric test on the computer 2.

Future work aims to examine the convergence of the various ART and SIRT methods and to implement a parallel matrix-vector Eigen3 multiplication (as discussed before). Finally it would also be interesting to implement and test the algorithms using a graphics processing unit (GPU) using some GPU library like NVIDIA's CUDA.

References

[1] A. C. Kak and M. Slaney, Eds., *Principles of Computerized Tomographic Imaging*. New York: IEEE Press, 1988.

[2] S. Kaczmarz, “Angenäherte Auflösung von Systemen linearer Gleichungen,” *Bulletin Internat. Acad. Polon. Sciences et Lettres*, pp. 355–357, 1937.

[3] P. C. Hansen and M. Saxild-Hansen, “AIR tools - A MATLAB package of algebraic iterative reconstruction methods,” *J. Computational Applied Mathematics*, vol. 236, no. 8, pp. 2167–2178, 2012. [Online]. Available: http://dx.doi.org/10.1016/j.cam.2011.09.039

[4] G. T. Herman, *Fundamentals of Computerized Tomography: Image Reconstruction from Projections*. Springer-Verlag, 2009. [Online]. Available: http://www.springer.com/computer/computer+imaging/book/978-1-85233-617-2

[5] C. D. C. D. Meyer, *Matrix analysis and applied linear algebra*. pub-SIAM:adr: Society for Industrial and Applied Mathematics, 2000. [Online]. Available: http://www.loc.gov/catdir/enhancements/fy0668/00029725-d.html;http://www.loc.gov/catdir/enhancements/fy0668/00029725-t.html

[6] A. H. Andersen and A. C. Kak, “Simultaneous algebraic reconstruction technique (SART): A superior implementation of the art algorithm,” *Ultrasonic Imaging*, vol. 6, no. 1, pp. 81–94, Jan. 1984. [Online]. Available: http://cobweb.ecn.purdue.edu/RVL/Publications/SART_84.pdf

[7] Y. Censor and T. Elfving, “Block-iterative algorithms with diagonally scaled oblique projections for the linear feasibility problem,” *SIAM Journal on Matrix Analysis and Applications*, vol. 24, pp. 40–58, 2002. [Online]. Available: http://www.optimization-online.org/DB_HTML/2001/12/418.html

[8] M. Jiang and G. Wang, “Convergence of the simultaneous algebraic reconstruction technique (SART),” *IEEE Trans. Image Processing*, vol. 12, no. 8, pp. 957–961, Aug. 2003. [Online]. Available: http://dx.doi.org/10.1109/TIP.2003.815295

[9] B. Overland, *C++ Without Fear: A Beginner’s Guide That Makes You Feel Smart*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2011.

[10] S. Amaya. Information on the c++ language: A brief description. Accessed: 15/12/2012. [Online]. Available: http://www.cplusplus.com/info/description/

[11] J. Reinders, *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O’Reilly, 2007. [Online]. Available: http://www.oreilly.com/catalog/9780596514808/index.html

[12] P. Wang. (2008) Compare windows* threads, openmp*, intel threading building blocks for parallel programming. Accessed: 15/12/2012. [Online]. Available: http://software.intel.com/en-us/blogs/2008/12/16/compare-windows-threads-openmp-intel-threading-building-blocks-for-parallel-programming/

[13] R. Friedman. (2012, Oct.) About the openmp arb and openmp.org. Accessed: 15/12/2012. [Online]. Available: http://openmp.org/wp/about-openmp/

[14] OpenMP Architecture Review Board, “Openmp application program interface,” Specification, 2011. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP3.1.pdf

[15] O. ARB. (2012) Openmp compilers. Accessed: 18/12/2012. [Online]. Available: http://openmp.org/wp/openmp-compilers/

[16] Intel. Why use intel tbb? Accessed: 15/12/2012. [Online]. Available: http://threadingbuildingblocks.org/

[17] ——. Intel threading building blocks benefits. Accessed: 15/12/2012. [Online]. Available: http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/index.htm

[18] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*. Reading, MA: Addison-Wesley, 1979.

[19] W. Petersen, *Introduction to parallel computing : [a practical guide with examples in C*. Oxford New York: Oxford University Press, 2004.

[20] V. C. V. Rao, "Tutorial notes on optimizing performance of parallel programs," Presentation on 5th International Conference and Exhibition on High-Performance Computing in the Asia-Pacific Region, June 2002.

[21] MSDN. Visual studio omp library reference. Accessed: 15/12/2012. [Online]. Available: http://msdn.microsoft.com/en-us/library/yw6c0z19(v=vs.80).aspx

[22] Intel, "Intel threading building blocks tutorial," Specification, 2012. [Online]. Available: http://software.intel.com/sites/default/files/m/f/0/3/Tutorial.pdf

[23] G. Guennebaud, B. Jacob *et al.* (2011) Eigen v3. Accessed: 15/12/2012. [Online]. Available: http://eigen.tuxfamily.org

[24] ——. (2011) Some important changes between eigen 2 and eigen 3. Accessed: 15/12/2012. [Online]. Available: http://eigen.tuxfamily.org/index.php?title=3.0

[25] ——. (2011) Column-major and row-major storage. Accessed: 15/12/2012. [Online]. Available: http://eigen.tuxfamily.org/dox/TopicStorageOrders.html

[26] ——. (2011) How does eigen compare to blas/lapack? Accessed: 15/12/2012. [Online]. Available: http://eigen.tuxfamily.org/index.php?title=FAQ

[27] Intel. (2012) Intel(r) threading building blocks - release notes. Accessed: 15/12/2012. [Online]. Available: http://threadingbuildingblocks.org/sites/default/files/resources/tbb-release-notes-4-1.txt

[28] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley, "An updated set of Basic Linear Algebra Subprograms (BLAS)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, Jun. 2002.

[29] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Blackford, and D. Sorenson, *LAPACK Users' Guide, Third Edition*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1999.