Parallelization Of The GMRES Method

Morgan Görtz

LUND UNIVERSITY

Centre for Mathematical Sciences
Numerical Analysis

CENTRUM SCIENTIARUM MATHEMATICARUM

Abstract

In this thesis a GMRES method is constructed and tested. The method is implemented in a C++ program with the MPI protocol to make it parallel. The parallel parts of this program are discussed in detail, especially the communication and the partitioning of the data between the processes. In the tests that were conducted we come to three important conclusions: the method works as expected, a substantial improvement is observed when more processes are introduced, and the method seems to be stable for very large systems.

Contents

1 List of terms

Parallel method	A method that utilizes multiple simultaneous processes.
Serial method	A method that only utilizes one process.
Local index	The index of a cell of a partition.
Global index	The index of a partitioned vector.
Local communication	Communication within one process.
Global communication	Communication between active processes.

2 Introduction

Solving linear systems is necessary for many numeric topics. It is for example, essential for solving stiff differential equations, solving several approximation problems, and running most advanced simulations. Construction of fast linear solvers is necessary because linear solvers are often an integral part of numerical methods. For example, the implicit Runge-Kutta methods, which construct and solve multiple different linear systems.

The linear solver implemented in this paper is the **G**eneralized **M**inimal **RES**idual (GMRES) method. It is an iterative solver that finds approximations within the Krylov subspace, a subspace of $\mathbb{R}^n$, of the system. The GMRES method works with approximations to a problem and allows the user to provide an initial approximation that the solver can start from. This is useful if you have an idea of what the solution to the system is, for example with linear multistep methods. The tests here were inspired by the implicit Euler method which fits well with the GMRES method.

Efforts were concentrated on making the GMRES method faster and hence more efficient. One way of doing this is to parallelize the method. Most modern computers have multiple processing cores that allow multiple programs to run simultaneously or allow one program to use multiple cores. For the latter you need a protocol that handles the communication between the cores. It was decided to use the most common protocol, MPI, within the C++ programming language. Although the protocol allows communication, it should not be used in abundance because communication is costly due to finite signal speed and hardware protocols. Due to this fact, a serial method can be very different from its parallel counterpart in terms of data handling. In this paper, the way the data and communication are handled is discussed in detail, as well as how the method was constructed.

3 Preliminaries

In this thesis, a set of theorems and mathematical concepts are introduced. These are necessary for the theoretical part of the GMRES method. Part of the implementation will also be discussed. First a brief introduction to the MPI protocol will be presented along with a couple of examples to clarify how the protocol is implemented. Lastly, a fully functioning MPI-parallelized vector class, in C++, is constructed, created, and discussed.

3.1 Cayleigh-Hamilton Theorem

The Cayleigh-Hamilton Theorem [1]:

Given an arbitrary matrix $A \in \mathbb{R}^{n\times n}$, the characteristic polynomial defined as:

$$p(x) = \det(A - Ix)$$

has the property:

$$p(A) = 0.$$

3.2 The Krylov Space

The Krylov space [2] $\mathcal{K}_m(A, y)$ is defined as the space spanned by the vectors:

$$(y, Ay, \ldots, A^{m-1}y),\ A \in \mathbb{R}^{n\times n},\ y \in \mathbb{R}^n.$$

3.3 GMRES

The GMRES method [2] is an iterative solver that solves:

$$Ax = b \Leftrightarrow b - Ax = 0,$$

where a non-singular matrix $A \in \mathbb{R}^{n\times n}$ and a vector $y \in \mathbb{R}^n$ is given. An initial guess, say $x_0 \in \mathbb{R}^n$, to the system is also given.

The main idea of the GMRES method is using a set of linearly independent vectors, $(v_1, \ldots, v_n) \in \mathbb{R}^n$, spanning a specific subspace of $\mathbb{R}^n$, to reduce the normed residual, $\|r_k\|_2$ over k. This is done by solving the following recursive least-squares problem:

$$\begin{cases} b - Ax_0 = r_0,\ k = 0 \\ \min\limits_{\dot{x}\in \text{ span}(v_1,\ldots,v_k)} \|b - A(x_{k-1} + \dot{x})\|_2 = \|b - Ax_k\|_2 = \|r_k\|_2,\ k = 1, \ldots, n. \end{cases}$$

We also note that if the basis $(v_1, \ldots, v_n)$ is orthogonal, the recursive step may be simplified to:

$$\min_{\alpha\in\mathbb{R}} \|b - A(x_{k-1} + \alpha v_k)\|_2 = \|b - Ax_k\|_2 = \|r_k\|_2.$$

The subspace that is used in the GMRES method is not necessarily defined as orthogonal, but in most implementations an orthogonal basis is found to make calculations easier.

The subspace that is used in GMRES method is always the Krylov space:

$$\mathcal{K}_m(A, r_0) = \text{ span}(r_0, Ar_0, \ldots, A^{k-1}r_0).$$

This specific subspace is chosen so that the residual r_k converges to $\mathbf{0}$ very quickly for diagonalizable matrices. In most cases, for such matrices, very few iterations have to be performed before the residual is negligible. Such fast convergence is greatly welcomed, especially in massive linear systems where a satisfying solution might be found within the first couple of iterations and thereby saving a great amount of computation time.

Convergence

We will show that if A is non-singular then x can be found within n iterations. Assume we are on the n:th iteration. Then x_n can be written as:

$$x_n = x_0 - \alpha_1 r_0 - \alpha_2 A r_0 - \ldots - \alpha_n A^{n-1} r_0.$$

If we put this into the least-squares problem we get:

$$\min_{\alpha_i \in \mathbb{C},\ i \in \{1,\ldots,n\}} \|b - Ax_0 + A\sum_{i=1}^{n} \alpha_i A^{i-1} r_0\|_2 = \min_{\alpha_i \in \mathbb{C},\ i \in \{1,\ldots,n\}} \|r_0 + \sum_{i=1}^{n} \alpha_i A^i r_0\|_2 =$$

$$\min_{\alpha_i \in \mathbb{C},\ i \in \{1,\ldots,n\}} \|\sum_{i=1}^{n} \alpha_i A^i\|_2 \|r_0\|_2 = \min_{p \in P_n, p(0)=1} \|p(A)\|_2 \|r_0\|_2.$$

Now to show that this is equal to zero and hence $x = x_n$, all we need is to find a polynomial of degree n that has the property:

$$p(A) = 0 \ \wedge \ p(0) = 1.$$

Because A is non-singular, the characteristic polynomial $p(x) = \det(A - Ix) \in P_n$ is non vanishing in 0 and hence we can create the polynomial:

$$\bar{p}(x) = \frac{p(x)}{p(0)} \in P_n$$

with the property:

$$\bar{p}(0) = 1.$$

Together with the Cayleigh Hamilton theorem we have:

$$\bar{p}(A) = \frac{p(A)}{p(0)} = 0$$

Hence there exists a polynomial with the required properties.

Rate of convergence

Say $\|r_k\|_2$ is the normed residual of the k:th iteration and that A is non-singular. Then we can write:

$$\|r_k\|_2 = \min_{p \in P_k, p(0)=1} \|p(A)\|_2 \|r_0\|_2 \Leftrightarrow$$

$$\frac{\|r_k\|_2}{\|r_0\|_2} = \|\bar{p}_k(A)\|_2$$

if:

$$\bar{p}_k = \min_{p \in P_k, p(0)=1} \|p(A)\|_2.$$

Assuming A is also diagonalizable and that $A = V\Lambda V^{-1}$ then:

$$\frac{\|r_k\|_2}{\|r_0\|_2} = \|V\bar{p}_k(\Lambda)V^{-1}\|_2 \Rightarrow$$

$$\frac{\|r_k\|_2}{\|r_0\|_2} \leq \kappa_2(V)\|\bar{p}_k(\Lambda)\|_2.$$

If the matrix A is diagonalizable, we note that the rate of convergence is directly linked to the largest diagonal entry of $\bar{p}_k(\Lambda)$, say σ_k, and at what rate $|\sigma_k|$ tends to zero as $k \to n$.

3.4 Householder transformations

Householder Transformations [3]:

To find the reflection of a point x with respect to the hyperplane orthogonal to a vector v, we have the following equation:

$$\dot{x} = x - 2(x, v)v = x - 2v(v^H, x) = (I - 2vv^H)x.$$

The matrix $I - 2vv^H$ is the householder transformation. By construction the matrix is symmetric and if v is normal, it is also orthogonal.

Triangulation with Householder transformations

One use of Householder transformations is to transform a non singular linear system to a triangular one. To do this a set of orthogonal Householder transformations $P_1, \ldots, P_{n-1} \in \mathbb{R}^{n\times n}$ are constructed with respect to a system, $A \in \mathbb{R}^{n\times n}, b \in \mathbb{R}$. With theses transformations we can transform the system

$$Ax = b$$

to:

$$P_{n-1} \ldots P_1 Ax = P_{n-1} \ldots P_1 b \Leftrightarrow$$

$$Rx = P_{n-1} \ldots P_1 b, \ R \text{ upper triangular.}$$

The construction of Householder transformations for triangularization

Say you have a non-singular matrix $A \in \mathbb{R}^{n\times n}$. Then by taking the first column, say A_1, we construct a vector v_1 by:

$$\alpha = -sgn(A_{11})\sqrt{\sum_{i=1}^{n}(A_{1i})^2},$$

$$r = \sqrt{\frac{1}{2}(\alpha^2 - A_{11}\alpha)},$$

$$v_{11} = \frac{A_{11} - \alpha}{2r}, \text{ and } v_{1i} = \frac{A_{1i}}{2r}$$

where A_{1i} and v_{1i} are the i:th index of the corresponding vector. The vector v_i by construction is a a unit vector. Using the vector v_1, we produce the first orthogonal Householder

transformation P_1. Applying the transformation P_1 to A produces a matrix, say A^1, with the property $A^1_1 = (\alpha_1, 0 \ldots, 0),\ \alpha_1 \in \mathbb{R}$. Because the first column of A^1 is of the form we want, the next Householder transformation P_2 will be on the form:

$$P_2 := \begin{bmatrix} 1 & 0 & \ldots & 0 \\ 0 & & & \\ \vdots & & \hat{P}_2 & \\ 0 & & & \end{bmatrix}.$$

To get $\hat{P}_2 \in \mathbb{R}^{(n-1)\times(n-1)}$, we generate a vector $\hat{v}_2 \in \mathbb{R}^{n-1}$ with the equation from before, but now with respect to A^1 and offset by one:

$$\alpha = -sgn(A^1_{22})\sqrt{\sum_{i=1}^{n-1}(A^1_{2i})^2},$$

$$r = \sqrt{\frac{1}{2}(\alpha^2 - A^1_{22}\alpha)},$$

$$\hat{v}_{21} = \frac{A^1_{22} - \alpha}{2r}, \text{ and } \hat{v}_{2i} = \frac{A^1_{1i}}{2r}.$$

Now constructing the orthogonal transformation $\hat{P}_2$ from the unit vector $\hat{v}_2$ we get the orthogonal transformation P_2 as previously defined. Applying P_2 to A^1 gives a matrix, say A^2, with the same property as A^1, but also has the second column of the form:

$$A^2_2 = (\alpha_{21}, \alpha_{22}, 0, \ldots, 0),\ \alpha_{21}, \alpha_{22} \in \mathbb{R}.$$

Continuing this pattern by offsetting the equations and using the equations to construct Householder transformations that only effect the last i columns and rows, by defining them as:

$$P_i = \begin{bmatrix} I_{i-1} & 0 \\ 0 & \hat{P}_i \end{bmatrix},\ I_i \text{ is the identity matrix of dimension } i,$$

we end up with the orthogonal transformations $P_1, \ldots, P_{n-1}$. These transformations when applied to the matrix A gives us the desired result:

$$P_{n-1} \ldots P_1 A = R,\ R, \text{ upper triangular.}$$

3.5 Givens Rotations

A givens rotation [4] J_i is an orthogonal linear transformation that only acts on positions i and $i+1$. The rotation is given by replacing the i:th and $i+1$:th row of the identity matrix to:

$$J = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & 0 & \\ & & c & s & & \\ & & -s & c & & \\ & 0 & & & \ddots & \\ & & & & & 1 \end{bmatrix}$$

where $s = \sin(\theta)$ and $c = \cos(\theta)$ for some $\theta \in \mathbb{R}$.

3.6 MPI

MPI [5] is a protocol that handles communication between processes. A MPI code works by having all processors run the code at the same time in different instances with different memory. In the beginning of each program the separate processes are given separate ID's and groups. These ID's can be used to identify and distribute work on the separate processes. A very simple setup would be:

Begin MPI
Get ID
Print ID
End MPI.

```
#include <mpi.h>
#include <iostream>

using namespace MPI;
using namespace std;

int main(){
    Init();
    int id = COMM_WORLD.Get_rank();
    cout<<"Hello from process: "<<id<<endl;
    Finalize();
    }
```

Figure 1: A C++ implementation of the simple program

In this code, the processes would each be assigned an ID and then simply print it. Communication is the next step because we wish to distribute work and MPI implements a broad set of communication types, for example:

Broadcast, send something from one process to all others
Send and receive, send something from one process to anther
Reduce, do a reduction operation on all processes entries and return it to a processes (for example summation)
Reduce All, do a reduction operation on all processes' entries and return it to all processes

In the method we intend to parallelize, we use vectors. The easiest way to combine MPI with vectors and vector operations is to split the vectors into parts:

$$v = (v_1, \ldots, v_n)$$

$$v = (\hat{v}_1, \ldots, \hat{v}_k).$$

Assuming we have k processes, we handle $\hat{v}_i$ on the separate processes.

Say we have a vector v that is partitioned on the different processes as vectors $\hat{v}_i$. A way to get the norm of the vector v on all processes would be:

Begin MPI
entry=Sum of the squared entries of $\hat{v}_i$
Reduce All(summation,entry,result)
norm=square root(result)
End MPI.

```
#include <mpi.h>
#include <math.h>

using namespace MPI;

int main(){
    Init();
    //Generate an example vector
    double vector[5]={1,2,3,4,5};
    //Calculate the local summs
    double sum=0;
    for(int i=0;i<5;i++){
        sum=vector[i]*vector[i];
        }
    //Calculate the global norm
    double norm=0;
    MPI_Allreduce(&sum,&norm,1,DOUBLE,SUM,COMM_WORLD);
    norm=sqrt(norm);
    Finalize();
    }
```

Figure 2: A C++ implementation of the norm program

Note that we do not need the ID of the processes in this method because we only use global communication.

3.7 A parallel vector class

For the parallelized vectors in this thesis, a special vector class, in C++, was written. This vector class utilizes a partition protocol that distributes the vector into cells among the active processes. It also handles basic arithmetic operations between vectors.

3.7.1 Partitioning

Normally a partition scheme is given by the user for specific problems and computer set-ups. Because no scheme was given, one had to be created. To partition a vector you need two parameters: the size of the vector, say n, and the number of processes, say k. Because this program is assumed to be run on identical processors within one computer, we can assume an equal distribution is optimal. When talking about the partitioned vector and the partitions themselves, the vectors' indices are referred to as global and the cells are referred to as local. The partition scheme that was used was that of Figure 3.

The first process: $L_{first} = n - (k-1)\text{floor}(\frac{n}{k})$

The other processes: $L_{rest} = \text{floor}(\frac{n}{k})$

Figure 3: The partition scheme

To verify that this is a valid partition, we sum all the cells lengths

$$n - (k-1)\text{floor}(\frac{n}{k}) + \sum_{i=2}^{k} floor(\frac{n}{k}) = n,$$

to validate that all elements of the vector are accounted for. In my method, I call this function *segment_length(n)*, where the number of processes is a global variable in MPI.

Mapping the partitions

With the length of all the separate partitions defined, we need to map theses partitions to the global vector. This is done by having the first process handle the first set of indices, the second process the second set and so on. To keep track of these positions, we define the variable l_i as the first global index of the partitioned vector on the current process i and u_i as the one after the last. These variables are defined as follows:

$$i \in \{1, \ldots, k\}$$

$$l_i = \begin{cases} 0, \ i = 1 \\ L_{first} + (i-2) * L_{rest}, \ \text{else} \end{cases}$$

$$u_i = L_{first} + (i-1) * L_{rest}.$$

Theses variables are defined as functions *segment_lower(n)* and *segment_upper(n)* in my program, where n is the dimension of the vector and the active process is determined by the process that calls it. These variables are used to easily identify what interval of the global vector a process is active in.

Utility functions

To work more easily with the defined protocol, two extra utility functions were created. The first *get_process(n,i)* returns an integer that represents the process that handles the global index i of a partitioned vector of dimension n. As well as a function *get_index(n,i)* that returns the local index to the local cell for the the global index i if n is the dimension.

Example

Given a vector $v \in \mathbb{R}^n$ and k processors, the vector v is partitioned as follows:

$$v = (v_1, \ldots, v_n)$$

$$v = (\hat{v}_1, \ldots, \hat{v}_k)$$

where $\#\hat{v}_1 = L_{first}$ and $\#\hat{v}_i = L_{rest},\ i = 2, \ldots, k$.

Say we wish to work with v_j, then we can use the utility function *get_process(j,n)* to get the $\hat{v}_i$ that this entry is partitioned to and then use the *get_index(j,n)* to get the index of the cell $\hat{v}_i$ that corresponds to v_j.

3.7.2 The main structure of the class

The following functions are defined in my vector class:

1. len(): Get the length of the vector
2. $li()$,$ui()$: where $[li(), ui())$ is the acting range of the global indices for the current process.
3. ([ ]): Get a single entry
4. (*=): Multiplication with scalar
5. (+,-,=,+=,-=): Vector operations
6. (*): Vector dot product.
7. (==,!=): Boolean operators.

Constructor

When an instance of this class is created, the only variable that is needed is the dimension of the global vector. All processors allocate a vector segment according to the partition scheme above (*segment_length(n)*) and store the length of the global vector in a variable *length*.

```
//Constructor
mpi_doubleVector(int n){
    length=n;
    segment=segment_lenght(n);

    //Get the lower and upper intervall of the active process
    LI=segment_lower(n);
    UI=segment_upper(n);

    //Get a vector segment accoringly
    vector=(double*)calloc(segment,sizeof(double));
    //Just a placeholder
    tmp=(double*)calloc(1,sizeof(double));

    };
```

Figure 4: The constructor of my *mpi_doubleVector* class

Single indicies

To make this class versatile, global indicies are used when modifying the vector. This is handled by using a dump variable and it works as follows:

Say I have a vector v and I want to set:

$$v[i] = 10.$$

If the process that has the global index tries this operation, it will succeed in changing the variable that corresponds to that index. But if another process tries to call this operation, it will only change a dump variable and hence do nothing. The way it is implemented is show in figure 5

```
//Get a single entry to the process that contains that entry
double& operator[](int i){
    if(i>=length){
        cout<<endl<<"Indentation error"<<endl;
        }
    const int id =COMM_WORLD.Get_rank();
    //Is index allocated in the current process
    if(get_process(length,i)==id){
        return vector[get_index(length,i)];
        }
        //If not return a dump varible
        return tmp[0];
    }
```

Figure 5: Overloading the [ ] operation using a dump variable

Creating parallel operations

With this setup, we could construct parallel independent operations, such as addition and subtraction, as is usually done with a loop for each index. This however would be counterproductive as it would involve a ton of meaningless operations being performed (so many that we might as well work in serial). Instead, the local variables l_i and u_i are used to isolate the relevant intervals for the local cells to bypass that problem. This is done and is implemented for example in overloaded += operation in figure 6.

```
//Create the += operation
void operator+=(mpi_doubleVector in){
    //If they don't have the same length then throw an exception
    if(length!=in.len()){
        cout<<"Vectors has different lenght"<<endl;
        throw;
    }else{
        for(int i =LI;i<UI;i++){
            vector_at_index(i)+=in[i];
            }
        }
    }
```

Figure 6: Overloading the += operation using local restriction variables $l_i = LI, u_i = UI$

Creating the parallel dot product

The only method in this class (not counting the Boolean operators) that requires communication between processes is the dot product. This is simple and very similar to the previous example in Figure 6. The only difference is that we perform an Allreduce at the end. The code for the dot product is presented in Figure 7.

```
//Create a dot product
double operator*(mpi_doubleVector in){
    //If they don't have the same length then throw an exception
    if(length!=in.len()){
        cout<<"Vectors has different lenght"<<endl;
        throw;
    }else{
        double tmp1=0,out=0;
        for(int i =LI;i<UI;i++){
            tmp1+=in[i]*vector_at_index(i);
            }
        MPI_Allreduce(&tmp1,&out,1,MPI_DOUBLE,MPI_SUM,COMM_WORLD);
        return out;
        }
    }
```

Figure 7: Overloading the * operation to get the dot product

3.8 CSV documents

I will use different programming languages when handling the data from all the tests. For the programs to work with the data generated and/or modified, some form of file to export the data needs to be generated. One of the easiest formats for this job is a *csv* (comma separated values) document [6]. This document has a set of values separated with commas and line breaks. The structure of the document is that of a table so, for example, Table 1 is written to a file as shown in Figure 8.

Table 1: The table we wish to express as an *csv* document.

Number	data1	data2
1	0.1	0.2
2	0.2	0.3
3	0.3	0.4

```
Number,data1,data2
1,0.1,0.2
2,0.2,0.3
3,0.3,0.4
```

Figure 8: Table 1 expressed as an *csv* document.

4 The GMRES method we implement

The implementation of the GMRES method that we are using was constructed and presented by Homer F. Walker in the paper *Implementation of the GMRES Method Using Householder Transformations* [7].

4.1 General GMRES

The system we wish to solve is:

$$Ax = b \Leftrightarrow b - Ax = 0$$

where a non-singular matrix $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$, and an initial guess $x_0 \in \mathbb{R}^n$ are given. We also define the initial residual as:

$$r_0 = b - Ax_0.$$

4.2 Elements of this implementation

In this implementation we are satisfied with a residual r_j such that:

$$\|b - Ax_j\|_2 = \|r_j\|_2 < tol$$

where *tol* is a tolerance parameter given by the user. We also restart the GMRES method when k iterations have been performed. Then we set the new calculated approximation x_k as x_0 and restart. The amount of times that the method is allowed to restart is set by a parameter *max_starts*. The GMRES method needs to be continuously restarted due to memory constraints. In this specific implementation of GMRES, we will not solve the least-squares problem until k iterations have been performed or a sufficient approximation to the system has been found. Because each iteration will require us to save an amount of data relative to the dimension of the system, system memory will fill up fairly quickly. The solution to the memory problem is to restart the method before that problem occurs.

4.3 The method

We will use Householder transformations P_i as defined in section 3.4 and Givens rotations J_i as defined in section 3.5 to orthogonalize the Krylov space to make it easier to solve the least-squares problem.

4.3.1 The iterative part of the GMRES implementation

The first part of the GMRES implementation will be described as an accumulative recursion over an index i in the following section. For clarification, v_i is a vector and v_{ij} is the j:th index of the vector v_i.

Initial step ($i = 1$)

First we generate P_1 and define v_1:

$$\begin{cases} P_1 : \ P_1 r_0 = \|r_0\|_2 e_1 \\ v_1 := P_1 A P_1 e_1. \end{cases}$$

Then we find P_2 such that:

$$P_2 : P_2 v_1 = (v_{11}, \psi, 0 \ldots, 0), \ \psi \in \mathbb{R}.$$

With this new transformation, P_2, we get J_1 by solving:

$$J_1 P_2 v_1 = (\alpha, 0 \ldots, 0), \ \alpha \in \mathbb{R}.$$

Finally we define R_1 by:

$$R_1 = J_1 P_2 v_1.$$

At this point we have v_1, P_1, P_2, J_1, and R_1 which is enough to start the recursion.

Recursive step ($i = 2, \ldots, k$)

First v_i is defined as:

$$v_i := P_i \ldots P_1 A P_1 \ldots P_i e_i.$$

With this definition, we find P_{i+1} by:

$$P_{i+1} : \ P_{i+1} v_i = (v_{i1}, ..., v_{ii}, \psi, 0, \ldots, 0), \ \psi \in \mathbb{R}.$$

Then we define a new vector $\bar{v}_i$ as:

$$\bar{v}_i := J_{i-1} \ldots J_1 P_{i+1} v_i.$$

To get J_i we solve:

$$J_i : \ J_i \bar{v}_i = (\bar{v}_{i1}, ..., \bar{v}_{i,i-1}, \theta, 0, \ldots, 0), \ \theta \in \mathbb{R}.$$

R_i is constructed by adding $J_i \bar{v}_i$ as a column to R_{i-1}:

$$R_i = [R_{i-1}, J_i \bar{v}_i].$$

Finally we check if we need to do another iteration. If ($i = k$) or ($|[J_1 \ldots J_i P_1 r_0]_{i+1}| < tol$), continue to the Solver in section 4.3.2. Otherwise, do the next recursive step $i + 1$.

4.3.2 The least-squares part of the GMRES implementation

In this section, we will use the variables generated in the accumulative recursion in section 4.3.1 to find the new approximation x_m assuming the recursion stopped at $i = m$. x_m will be the solution to the GMRES implementation and the GMRES method for m iterations. The statements made in the following section will be proven in section 4.4

Solver

Say that the iteration stopped at $i = m$. Then the first m rows and columns of R are an upper triangular matrix, say $\hat{R}_m$, by construction. This matrix and the first m coordinates of $w = J_1 \dots J_i P_1 r_0 \in \mathbb{R}^n$, say $\hat{w} \in \mathbb{R}^m$, form the system:

$$\hat{R}_m y = \hat{w}.$$

Solving this $y \in \mathbb{R}^m$ will be the last step of finding the solution to the GMRES method for m iterations, because the solution x_m can be calculated by:

$$x_m = x_0 + y_1 P_1 e_1 + \dots + y_1 P_1 \dots P_m e_m.$$

This x_m will solve the least-square problem:

$$\min_{\dot{x} \in\ x_0 + \mathcal{K}_m(A, r_0)} \|b - A\dot{x}\|_2 = \|b - Ax_m\|_2 = \|w_{m+1}\| = \|r_m\|_2$$

which completes the GMRES method for m iterations.

Restart

Once x_m is found, we need to check if our new approximation satisfies the desired tolerance. This is simple because $\|r_m\|_2 = \|w_{m+1}\|_2$. If our new approximation satisfies the desired tolerance or if the maximum number of restarts (max_starts) is met, we return the new approximation with a corresponding flag. If not, restart the method by doing another recursion in section 4.3.1 with x_m as x_0

4.4 A comment about how this works

In this section, we will prove two major statements made in section 4.3.2 in the solver subsection. We will also use the definitions stated in section 4.3.1. The first proof will be to show that $(P_1 e_1, \dots, P_1 \dots P_m e_m)$ is a basis for $\mathcal{K}_m(A, r_0)$. The second proof will show that x_m is indeed the solution to the least-square approximation and hence the solution to the GMRES method after m iterations.

A basis of the Krylov space

In this section we will show that:

$$\text{span}(P_1 e_1, \dots, P_1 \dots P_m e_m) = \mathcal{K}_m(A, r_0).$$

The proof will be inductive. We will assume that $m > 1$ without loss of generality.

Proof:
The base case comes directly from the definition of P_1. We know that:

$$P_1 r_0 = \|r_0\|_2 e_1.$$

Because P_1 is orthogonal and symmetric, we know it is self-inverse. This implies that:

$$(P_1)^2 r_0 = \|r_0\| e_1 \Leftrightarrow r_0 = \|r_0\| P_1 e_1.$$

That in turn means that $P_1 e_1$ spans the same space as r_0 which is the space of $\mathcal{K}_1(A, r_0)$.

With the base proved, we wish to show that:

$$\text{span}(P_1 e_1, \ldots, P_1 \ldots P_i e_i) = \mathcal{K}_i(A, r_0) \Rightarrow \text{span}(P_1 e_1, \ldots, P_1 \ldots P_{i+1} e_{i+1}) = \mathcal{K}_{i+1}(A, r_0).$$

To start, we note that if:

$$\mathcal{K}_i(A, r_0) = \text{span}(r_0, Ar_0, \ldots, A^{i-1} r_0) = \text{span}(P_1 e_1, \ldots, P_1 \ldots P_i e_i) \Rightarrow$$

$$\mathcal{K}_{i+1}(A, r_0) = \text{span}(r_0, Ar_0, \ldots, A^{i-1} r_0, A^i r_0) = \text{span}(P_1 e_1, AP_1 e_1, \ldots, AP_1 \ldots P_i e_i).$$

This is because the r_0 component in $\mathcal{K}_{i+1}(A, r_0)$ is spanned by $P_1 e_1$ and the rest of the components are accounted for due to the induction hypothesis (it is just multiplied by A). This means that we can instead prove the statement:

$$\text{span}(P_1 e_1, \ldots, P_1 \ldots P_{i+1} e_{i+1}) = \text{span}(P_1 e_1, AP_1 e_1, \ldots, AP_1 \ldots P_i e_i).$$

This can be proven if we apply the orthogonal transformation $P_i \ldots P_2 P_1$ to each subspace:

$$\text{span}(P_1 e_1, \ldots, P_1 \ldots P_{i+1} e_{i+1}) = \text{span}(P_1 e_1, AP_1 e_1, \ldots, AP_1 \ldots P_i e_i) \Leftrightarrow$$

$$\text{span}(P_{i+1} \ldots P_2 P_1 (P_1 e_1, \ldots, P_1 \ldots P_{i+1} e_{i+1})) = \text{span}(P_{i+1} \ldots P_2 P_1 (P_1 e_1, AP_1 e_1, \ldots, AP_1 \ldots P_i e_i)).$$

On the left hand side, we realize that P_i is self-inverse and $P_j e_i = e_i,\ j > i$. On the right hand side, we know that $P_i \ldots P_1 A P_1 \ldots P_i = v_i$ by definition. Applying this information to the equation gives us:

$$\text{span}(e_1, \ldots, e_{i+1}) = \text{span}(P_{i+1} \ldots P_2 v_1, \ldots, P_{i+1} v_i).$$

Now we use the definition of P_{j+1}:

$$P_{j+1} :\ P_{j+1} v_j = (v_{j1}, ..., v_{jj}, \psi, 0, \ldots, 0),\ \psi \in \mathbb{R}.$$

Here we note that $P_j P_{i+1} v_j = P_{i+1} v_j,\ j > i+1$ which means that:

$$\text{span}(e_1, \ldots, e_{i+1}) = \text{span}(P_2 v_1, \ldots, P_{i+1} v_i).$$

Then by taking the only possible non-zeros rows of $[P_2 v_1, \ldots, P_{i+1} v_i]$, say $T \in \mathbb{R}^{(i+1)\times(i+1)}$, we get an upper triangular square matrix. Due to the vectors $(P_2 v_1, \ldots, P_{i+1} v_i)$ only being able to effect the first $i+1$ indicies as a basis, we conclude that:

$$u \in \text{span}(P_{i+1} \ldots P_2 v_1, \ldots, P_{i+1} v_i) \Leftrightarrow$$

$$u_j = 0,\ j > i+1 \Leftrightarrow$$

$$u \in \text{span}(e_1, \ldots, e_{i+1}).\ \square$$

The proof is now finished by the principle of mathematical induction.

The solution to the least-square problem

To show that the solution of our method is indeed the solution to the GMRES method we will show that:

$$\min_{\dot{x}\in\ x_0+\mathcal{K}_m(A,r_0)} \|b - A\dot{x}\|_2 = \|b - Ax_m\|_2 = \|r_m\|_2 = (\mathrm{X}).$$

Proof:
In the previous proof, we showed that $\dot{x}$ can be written as:

$$\dot{x} = x_0 + \alpha_1 P_1 e_1 + \ldots + \alpha_m P_1 \ldots P_m e_m,\ \alpha_i \in \mathbb{R}$$

where $i = 1, \ldots, m$, due to $(P_1 e_1, \ldots, P_m \ldots P_1 e_m)$ being a basis for $\mathcal{K}_m(A, r_0)$. So let us put this form in the least-square problem that we wish to prove:

$$(\mathrm{X}) = \min_{\alpha_i\in\mathbb{R}:\ i=1,\ldots,m} \|b - A(x_0 + \alpha_1 P_1 e_1 + \ldots + \alpha_m P_1 \ldots P_m e_m)\|_2 =$$

$$\min_{\alpha_i\in\mathbb{R}:\ i=1,\ldots,m} \|r_0 - A(\alpha_1 P_1 e_1 + \ldots + \alpha_m P_1 \ldots P_m e_m)\|_2.$$

Next we notice that P_i is orthogonal. This means that we can multiply any sequence of them in the least-squares problem and not change the outcome. We do this with the multiplication of $(P_{m+1} \ldots P_1)$ to get:

$$(\mathrm{X}) = \min_{\alpha_i\in\mathbb{R}:\ i=1,\ldots,m} \|P_{m+1} \ldots P_1 r_0 - P_{m+1} \ldots P_1 A(\alpha_1 P_1 e_1 + \ldots + \alpha_m P_1 \ldots P_m e_m)\|_2.$$

We now observe that $v_i = P_i \ldots P_1 A P_1 \ldots, P_i e_i$ and insert that in the problem:

$$(\mathrm{X}) = \min_{\alpha_i\in\mathbb{R}:\ i=1,\ldots,m} \|P_{m+1} \ldots P_1 r_0 - (\alpha_1 P_{m+1} \ldots P_2 v_1 + \ldots + \alpha_m P_{m+1} v_m)\|_2.$$

At this point many terms will disappear. Let us start with $P_1 r_0$. We know that:

$$P_1 r_0 = \|r_0\|_2 e_1.$$

We also know that $P_i e_1 = e_1,\ i > 1$, because of construction. Put these two facts together and we get:

$$P_{m+1} \ldots P_1 r_0 = P_1 r_0.$$

But that is not all. We also know that $P_{i+1} v_i$ is composed of only zeros after the index $i + 1$. This means multiplying with $P_j,\ j > i + 1$ is redundant. Removing all these unnecessary factors leaves us with the problem:

$$(\mathrm{X}) = \min_{\alpha_i\in\mathbb{R}:\ i=1,\ldots,m} \|P_1 r_0 - (\alpha_1 P_2 v_1 + \ldots + \alpha_m P_{m+1} v_m)\|_2.$$

It is now time to multiply with the orthogonal transformation $(J_m, \ldots, J_1)$:

$$(\mathrm{X}) = \min_{\alpha_i\in\mathbb{R}:\ i=1,\ldots,m} \|J_m \ldots J_1 P_1 r_0 - J_m \ldots J_1(\alpha_1 P_2 v_1 + \ldots + \alpha_m P_{m+1} v_m)\|_2.$$

Now note that $J_m \ldots J_1 P_1 r_0 = w$. But we will also realize that:

$$J_1 P_2 v_1 = (\beta_1, 0, \ldots, 0) = J_2 J_1 P_2 v_1 = J_3 J_2 J_1 P_2 v_1 = \ldots$$

$$J_2 J_1 P_3 v_2 = (\beta_{21}, \beta_{22}, 0, \ldots, 0) = J_3 J_2 J_1 P_3 v_2 = J_4 J_3 J_2 J_1 P_3 v_2 = \ldots$$

$$\vdots$$

Inserting this knowledge into the problem gives us:

$$(\mathrm{X}) = \min_{\alpha_i \in \mathbb{R}:\ i=1,\ldots,m} \|w - (\alpha_1 J_1 P_2 v_1 + \ldots + \alpha_m J_m \ldots J_1 P_{m+1} v_m)\|_2.$$

To finish this proof, we only need to realize that α_i is a scalar to the vector $J_i \ldots J_1 P_{i+1} v_i$, which happens to be the i:th column of R_m. Because R_m is an upper triangular matrix, we can only effect the first m rows. Therefore, α_i must be uniquely defined as the coefficients of the vector $y \in \mathbb{R}^n$, given that $\hat{R}_m$ is the first m column and rows of R_m and that

$$\hat{R}_m y = \hat{w},$$

if $\hat{w}$ is the first m indicies of w. Finally, we look at the least-square equation and realize that w only has the first $m+1$ indicies as possible non-zero entries. This, together with the fact that the first m coefficients of w will be subtracted due to the choice of coefficients α_i, leads us to the statement:

$$(\mathrm{X}) = \min_{\alpha_i \in \mathbb{R}:\ i=1,\ldots,m} \|w - (\alpha_1 J_1 P_2 v_1 + \ldots + \alpha_m J_m \ldots J_1 P_{m+1} v_m)\|_2 =$$

$$\|w - (y_1 J_1 P_2 v_1 + \ldots + y_m J_m \ldots J_1 P_{m+1} v_m)\|_2 = \|(0, \ldots, 0, w_{m+1}, 0 \ldots, 0)\|_2 = \|r_m\|_2 = |w_{m+1}|. \ \square$$

5 The components of the GMRES implementation

5.1 Data structure

Variables

1. Householder transformations P_i
2. Givens rotations J_i
3. Residual vector $\bar{w}$
4. The triangular matrix R

Storage

The Householder transformation P_i can be defined by a vector v as described in section 3.4. We will use this definition to store the Householder transformations as columns of a matrix H. In this chapter we will see p_i as the vector corresponding to the Householder transformation P_i. The Givens rotations can be defined as an angle. Instead of using a vector of angles, we choose to use two vectors: one with the sine value and one of the cosine value of the given angles as the vectors $\bar{s}$ and $\bar{c}$. The triangular matrix and the residual vector are stored in a matrix R and vector $\bar{w}$.

Parallelization

The restart size is very small with respect to the dimension, say n, of the system to be solved. We therefore assume that the restart size is less than the size of the cell defined by the partition of a vector with dimension n on the first process as described in section 3.7.2. This allows us to make most of the calculations local to the first process without

any communication. With this in mind, the variables are divided into two groups: one consisting of variables not requiring communication between processes, serial variables, and ones that do. Let us call the latter parallel. This partition of variables is shown in Table 2.

Parallel	Serial
P_i	J_i
H	R
	$\bar{w}, \bar{s}, \bar{c}$

Table 2: Parallel and serial variables

5.2 The serial variables

The serial process on the first core in this method will handle the least-squares problem. The process will require the construction and solving of a triangular system. For this, a serial vector, matrix, and triangular solver class were created.

5.2.1 Parallelized Householder functions

The Householder functions are divided into two methods. The first generates a vector h that represents the transformation given by a vector x. The other takes the vector h and applies the represented transformation to a vector y.

Generating the Householder transformations

The Householder transformations that we wish to generate are of the form discussed in Section 3.4. Because we are working with multiple sizes of householder transformations $\hat{P}_k \in \mathbb{R}^{k\times k}$, we give a parameter k that symbolizes the offset. With this k and a vector $x = (x_1, \ldots, x_n)$, we can construct the vector $v \in \mathbb{R}^n$, which defines the Householder transformation by the following equations:

$$\alpha = -sgn(x_k)\sqrt{\sum_{i=k}^{n} x_i^2},$$

$$r = \sqrt{\frac{1}{2}(\alpha^2 - x_k\alpha)},$$

$$v_j = 0 : \ j < k, \ v_k = \frac{x_k - \alpha}{2r}, \text{ and } v_i = \frac{x_i}{2r} : \ i > k.$$

The transformation $\hat{P}_i$ is then used to construct P_k by:

$$P_k = \begin{bmatrix} I_{n-k} & 0 \\ 0 & \hat{P}_k \end{bmatrix}.$$

This P_k is constructed with the following property:

$$P_k x = (x_1, \ldots, x_{k-1}, \alpha, 0, \ldots, 0).$$

Because the GMRES method requires us to perform this operation, the α is introduced as an output to the function. This is done so we can perform the transformation $P_k x$ by

the definition instead of calculating it. The code uses similar syntax as the dot product in the parallelized vector class in section 3.7.2, with the exceptions that the active indices are restricted by setting conditions in the loop and that the calculations are made on the first process (ID 0) before being broadcast. The calculation is performed on the first process because the resulting parallelized vector $P_k x$ will not include any non-zero entries on any other process in this implementation of the GMRES method.

```
void generateHousholder(mpi_doubleVector x, mpi_doubleVector out,int k,double *alpha){
    /**
    * Vector x that the transformation is to be defined by
    *  and an offset integer k (from start)
    *
    * It also has an output varible alpha that gives the first
    * vector possitions value after a transformation
    */

    //Get the id
    const int id =COMM_WORLD.Get_rank();
    //Get a varible to sum the segments
    double segsum=0;

    for(int i =x.li(); i<x.ui();i++){
        if(i>=k){
            //transfer the input vector to the output
            out[i]=x[i];
            //Calculate the sum part of alpha
            segsum+=x[i]*x[i];
            }
        }

    double r=0;
    //Get the result on id 0
    MPI_Reduce(&segsum,&r,1,MPI_DOUBLE,MPI_SUM,0,COMM_WORLD);
    //do the final calculations
    if(id==0){
        //Generate alpha and r
        *alpha = -sign(x[k]) * sqrt(r);
        r = sqrt( 0.5*((*alpha)*(*alpha)-x[k]*(*alpha)));
        //get the first value of the transformation vector
        out[k]=x[k]-*alpha;
    }else{
        *alpha=0;
        }
    //Send r to all process:
    MPI_Bcast(&r, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    //and apply 1/2r:
    for(int i =x.li(); i<x.ui();i++){
        if(i>=k){
            out[i]*=(1./(2.*r));
            }
        }
    }
```

Figure 9: The parallelized function that generates the Householder transformations

Applying the Householder transformations

Given a vector p_k that characterizes an Householder transformation P_k generated by the

above method, we create an algorithm that transforms a given vector x by:

$$P_k x = y = (I - 2p_k p_k^T)x \Leftrightarrow$$

$$y = x - 2p_k(x, p_k).$$

The method in my code calculates (x, p_k). Given this, the rest of the calculations can be performed without communication.

```
void applyHouseholder(mpi_doubleVector t,mpi_doubleVector x,mpi_doubleVector y,int k){
    /**
    * Takes a transformation vector t, a vector that the transformation is to be made on x
    *  and an offset integer k (from start)
    *
    */

    double segsum=0;

    for(int i =x.li(); i<x.ui();i++){
        if(i>=k){
            segsum+=t[i]*x[i];
            }
        }

    double tc=0;
    //Broadcast the sum
    MPI_Allreduce(&segsum,&tc,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
    tc*=2.0;

    //The changed part:
    for(int i =x.li(); i<x.ui();i++){
        if(i>=k){
            y[i]=x[i]-tc*t[i];
            }
        }
    }
```

Figure 10: The parallelized function that applies the Householder transformations ($p_k = t$ in this code).

The GMRES method

The code for the method can be found in Appendix 1. The method is constructed as a class without a constructor and only has one type parameter that determines what matrix class is used. GMRES is a function to this class and it takes the parameters required to solve a system together with a tolerance and restart size. The method is written similarly to the other parts of this chapter with the exception of the serial part on the first process. However, these methods and parts are not discussed in this paper.

6 Test: Does the method work

With this test, we validate that the implemented method works. We will construct a test case where the basic properties of the method will be analyzed and compared to the theoretical one in order to see if they share the same properties.

6.1 Method

A test case was created to determine if the method works. A random dense matrix $A \in \mathbb{R}^{n \times n}$ and vector $b \in \mathbb{R}^n$ are generated, where $n = 500$. This system is then solved with initial approximation $x_0 = \mathbf{0}$ for 1, 2, 3 and 4 active processes.

To determine if the method acts appropriately, we log the normalized residual $\|r_i\|_2$ in each iteration and save the result in a *csv* document. With these four documents of data (execution time) four graphs are produced and superimposed to one figure with a Python script.

6.2 Presentation

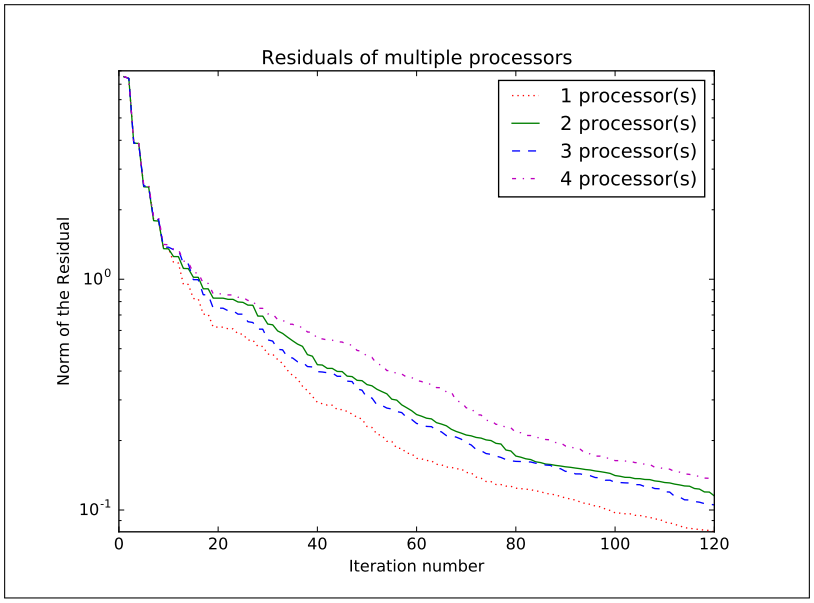

Figure 11: The residual of the approximation after each iteration

6.3 Conclusion

The residuals in the graph are decreasing. This means that the solver is able to find better approximations to the problem after each iteration. The residuals are almost exactly the same for each iteration. This is what we want because, mathematically speaking, the method should do the same thing in each iteration, independent of how many processors are active. The small deviations that are observed are most likely due to different rounding errors caused by the reduction steps. The deviations occur because the order in which the elements are multiplied makes a difference. A final observation is that the curve seems to be strictly monotonically decreasing. The form of the curve fits our method because we find better approximations in each iteration. Incidents where the new approximation is the same as the old are very uncommon. This is enough to conclude that the method works as we anticipate and that we can proceed to analyze the improvements it brings.

7 Test: Improvements

In this test, the execution time of the parallelized linear solver is analyzed. The analysis covers solving both sparse and dense matrices with multiple active processes. Fairly large systems are used because parallelization only makes sense for larger cases and the tests are performed with differential equation solvers in mind (IRK 1). In this report, it was concluded that the method does in fact improve execution time by introducing more processors. We also observe that the more complex and the larger the system is, the larger the improvement becomes.

7.1 Main questions we wish to answer

1. What is the reduction in computation time when more processors are used?
2. How does computation time differ between a sparse and a dense matrix?
3. What can be said about the improvement in execution time when larger more complex systems are solved?
4. At what rate does the improvement grow if we increase the number of iterations?

7.2 Method

First, the GMRES method previously constructed is modified to take number of iterations instead of a tolerance. With this solver, two testing programs that call this method need to be constructed: one solving a sparse matrix and the other a dense. The two programs take four parameters:

1. The name of an the output file (<name>.csv).
2. Dimension of the matrix ($\dim(A)$).
3. Number of iterations (itr).
4. How many times the system should be solved and execution time logged. (sample size)

With the dimension as input, say n, we generate the following systems:

Dense Matrix

The program that solves a dense matrix generates a random matrix $R \in \mathbb{R}^{n\times n}$ and $b \in \mathbb{R}^n$ with entries spanning from $0 - \frac{100}{n}$. The entries are bounded by $\frac{100}{n}$ in order to avoid overflow errors. The matrix A is then constructed by:

$$A = I - 10^{-2}R$$

in the form of the implicit Euler method, where $R = f_x(t, x)$ and $h = 10^{-2}$. Given an initial condition $x(t_0) = b$, we can perform a step, where $x(t_1) = x,\ t_1 = t_0 + h$, with the implicit Euler method, by solving the following system:

$$Ax = b$$

Sparse Matrix

The sparse matrix we solve for is the finite element discretization of the heat equation:

$$\frac{\partial^2 u(t,x)}{\partial x^2} = \frac{\partial^2 u(t,x)}{\partial t},\ u(t,0) = u(t,l) = 0$$

$$\frac{1}{\Delta x^2}\begin{bmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & 0 & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & 1 & -2 & 1 \\ & & & & 1 & -2 \end{bmatrix} \bar{u}(t) = \frac{1}{\Delta x^2} H\bar{u}(t) = \frac{d\bar{u}(t)}{dt},$$

where the system we solve is the first step of the implicit Euler method on the discretization:

$$(I - \frac{h}{\Delta x^2} H)\bar{u}_1 = \bar{u}_0$$

For the initial step $\bar{u}_0 \in \mathbb{R}^n$, we discretize the function $f(x) = \sin(2\pi x)$ on the interval $(0, 1)$. Thus:

$$u_0^i = f(\frac{i+1}{n+1}), i = 0, \ldots, n$$

With this particular discretization, we note that $\Delta x = (n+1)^{-1}$. To complete the system, we set $\Delta h = 10^{-2}$ to produce the following problem to be solved:

$$(I - 10^{-2}(n+1)^2 H)\bar{u}_1 = \bar{u}_0$$

Main structure of the testing programs

The programs does the following operations depending on the desired sample size:

1. Generate dense matrix and the b vector with random entries between $0 - \frac{100}{n}$ or generate the sparse matrix and the accompanying initial condition.
2. Generate x_{itr} by letting the method do itr iterations.
3. Log the execution time it took to find the itr's approximation and log it in the output document.

This program produces a document with multiple samples of the execution time solving systems with the same dimension and iteration parameters.

Tests

The programs were run through an R script that solved multiple different problems with one, two, or three processors. The documents that were produced were analyzed and the mean value of its content was logged as the answer to the particular system. The script generated another *csv* document that had the mean execution time for various different systems. This document was used in a python script that produced a graph of superimposed data (execution time) for one, two, or three processors solving the different systems. The following tests where conducted:

Table 3: The parameters for the dense tests

Test	Dimension	Iterations	Samples
Test 1	$2 * 10^2 - 2 * 10^3$	15	5
Test 2	10^3	$21 - 126$	5

Table 4: The parameters for the sparse tests

Test	Dimension	Iterations	Samples
Test 1	$2 * 10^2 - 2 * 10^3$	55	5
Test 2	10^3	$105 - 240$	5

7.3 Presentation

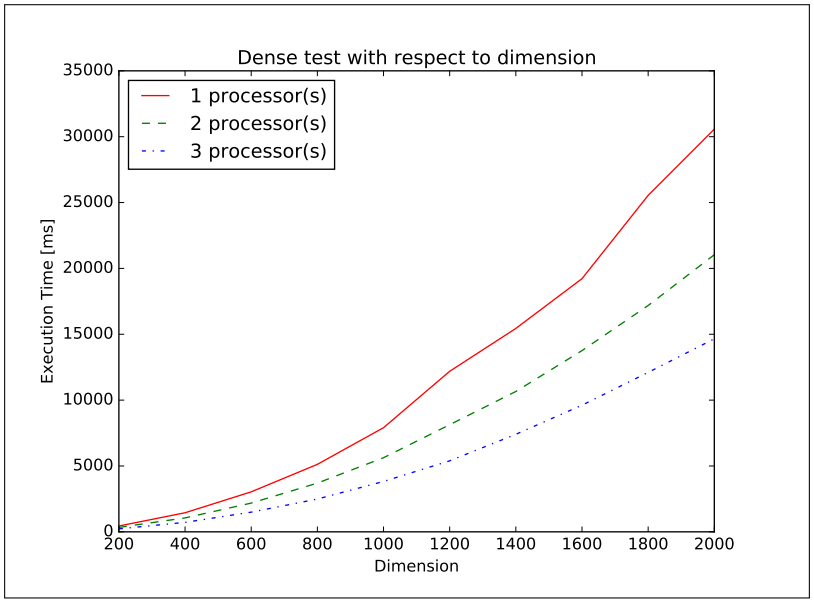

Figure 12: Execution time with respect to dimension for solving dense systems. The test parameters are those of Table 3, Test 1.

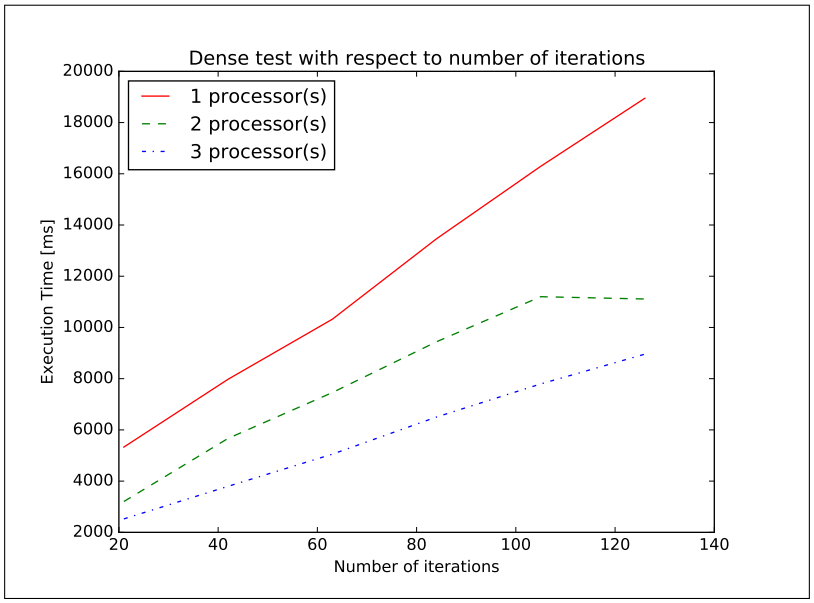

Figure 13: Execution time with respect to number of iterations for solving dense systems. The test parameters are those of Table 3, Test 2.

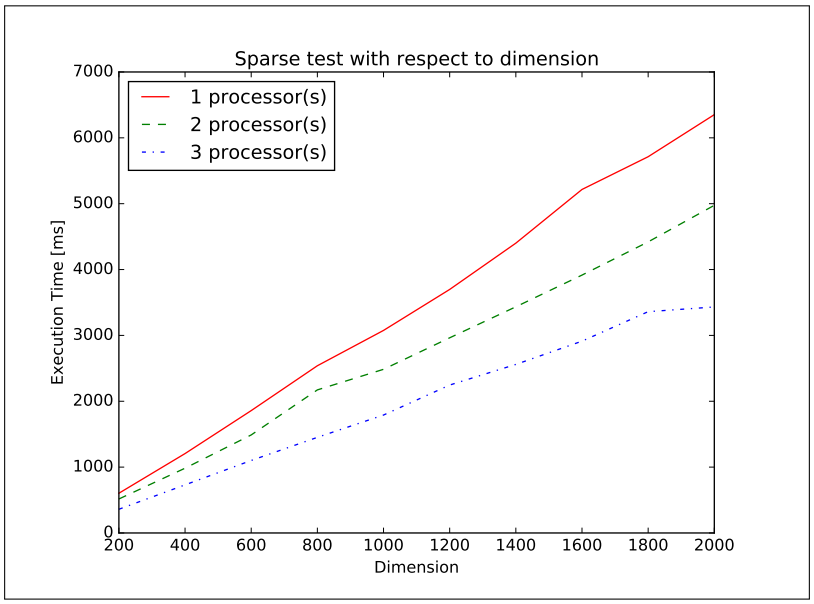

Figure 14: Execution time with respect dimension for solving sparse systems. The test parameters are those of Table 4, Test 1.

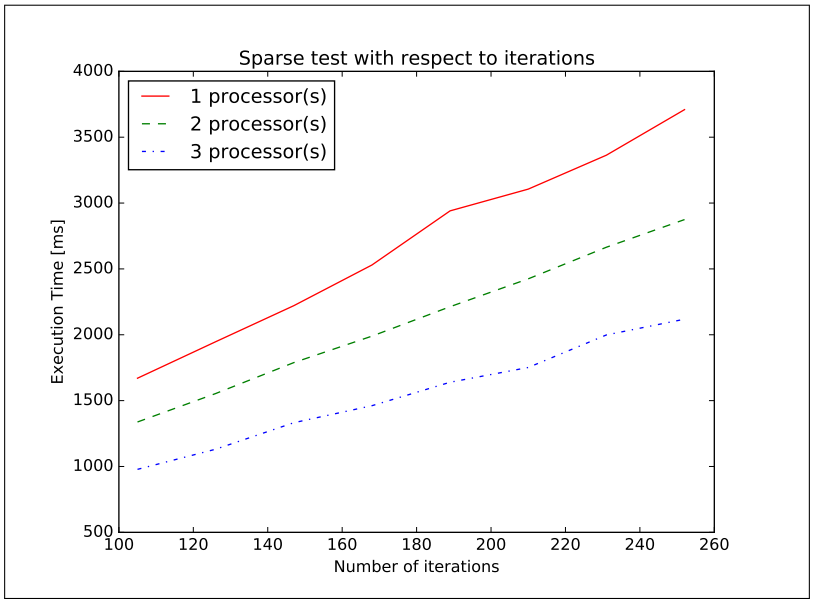

Figure 15: Execution time with respect to number of iterations for solving sparse systems. The test parameters are those of Table 4, Test 2.

7.4 Validity discussion

The program was run on a lightweight computer with no other active programs for optimal test conditions. With multiple samples, any inconsistencies should be minimal. Because we are interested in the impact of involving multiple processors, this setup should give us a general idea of the improvement, the nature of the improvement, and what form the improvement takes.

Mathematically the problem does not change if we use one or one thousand processors because the method relies only on mathematical transformations [7]. This means that any time we look at the execution time with respect to number of processors or number of iterations it is not an artificial improvement. The same cannot be said when we look with respect to dimension. The dense system and sparse system are dependent on dimension. With this in mind, we can continue to analyze the results.

7.5 Discussion

Improvements were seen in all tests. The largest improvements were seen when dense systems were solved. With dense matrices, quadratic curves were seen with respect to dimension (Figure 12) whereas linear curves were observed in the sparse case (Figure 14). This makes sense because a vector-matrix multiplication is of order $\mathcal{O}(n^2)$ for dense matrices and $\mathcal{O}(n)$ for sparse and this operation is done relative to the number of iterations (constant) with respect to dimension (scaling). We can clearly see spreading in Figure 12 and 14 which would suggest that solving larger dimensional systems would result in a larger improvement of execution time.

When introducing more iterations, linear curves should occur because the same operation is performed more times. This is evident in Figures 13 and 15 and even here we can see spreading which also leads to a larger improvement of execution time when

increasing the number of iterations. This fact, together with the previous statement about improvement with respect to dimension, leads to the conclusion that we can see greater improvement in computation time if we work with larger more dense systems that require more iterations. This is what we want to see because this implies that the more the parallelized solver works, the greater the improvement.

7.6 Conclusions

Improvements with respect to degree

We see a quadratic improvement when dense systems are solved and a linear improvement for sparse systems. This means that the larger the system is the greater the improvement becomes. This is verified for systems with dimensions in the interval $[200, 2000]$.

The iterations effect on execution time

Because the number of iterations reflects how much the solver has to work on a specific problem, there should be a consistent improvement as long as an initial improvement for that specific problem exists. In the tests we have conducted, there is an initial improvement for matrices with dimensions between two hundred to two thousand.

Larger more complex systems

Everything in these tests suggests that a greater improvement will be seen when larger more complex systems are to be solved. This means that the more the method has to work, the greater the extent of the improvement.

8 Test: Consistency for very large systems

Lastly, we analyze the consistency of the method compared to dimension. In this part we only analyze sparse systems because dense matrices introduce an array of problems when large, for example: overflows, memory allocation, and very time consuming to solve. It will be shown that the method holds up for large sparse systems and that it tends to act fairly similar relative to the smaller cases.

8.1 Main questions we wish to answer

1. Is the method stable for very large systems?
2. Is the shape of the execution time / dimension curve preserved for larger systems?
3. What can be said about the improvement in execution time when larger more complex systems are solved?

8.2 Method

In this test, we use the results from Table 4 Test 1 to generate a linear curve by finding the least-square approximation for the generated sample points. To do this, a python script was written to solve the system:

$$D^T Dx = D^T Y$$

where:

$$D = \begin{bmatrix} 1 & d_1 \\ 1 & d_2 \\ 1 & d_3 \\ \vdots & \\ 1 & d_n \end{bmatrix}, \; d_i : \text{ the different dimensions sampled,}$$

$$y = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{bmatrix}, \; e_i : \text{ the result in execution time for the corresponding dimension } d_i,$$

and

$$x = \begin{bmatrix} a \\ b \end{bmatrix}.$$

This particular system when solved gives the coefficients a, b to the function $l(x) = ax + b$, which is the least-square approximation we wish to analyze [8]. With this formula, the linear approximations for one, two, and three processors are generated and together with the sample points are superimposed to one graph: Figure 16.

These least-square approximations are then compared to a few samples from very large systems. To do this, we use the same method that generated the set of sample points used in the approximations, but with larger dimensions. The new set of samples were generated with the parameters of Table 5:

Table 5: The parameters for sparse systems with large dimensions

Dimension	Iterations	Samples
$4e3, 8e3, 12e3$	55	5

Lastly, a second graph is produced with the approximations ranging to the dimension of the new set of sample points with the new sample points marked in the graph, Figure 17.

8.3 Presentation

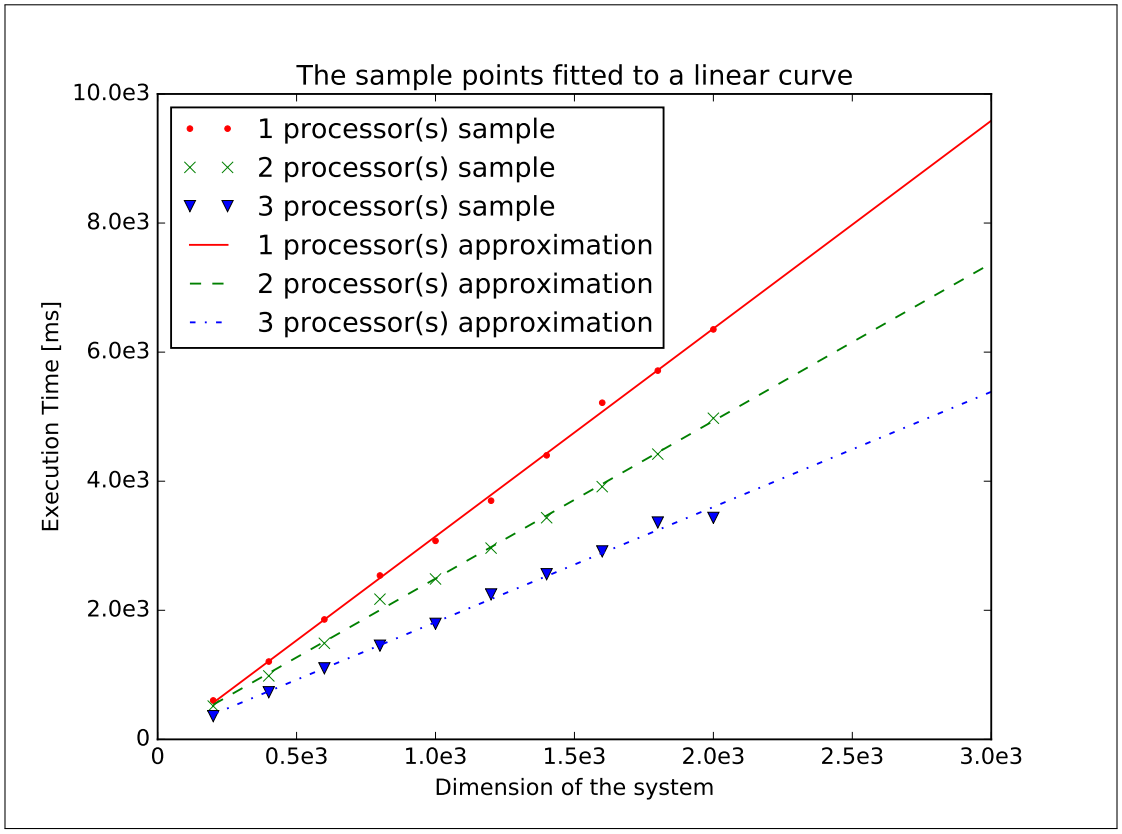

Figure 16: Approximated linear curves generated by the sample points from Table 4 Test 1

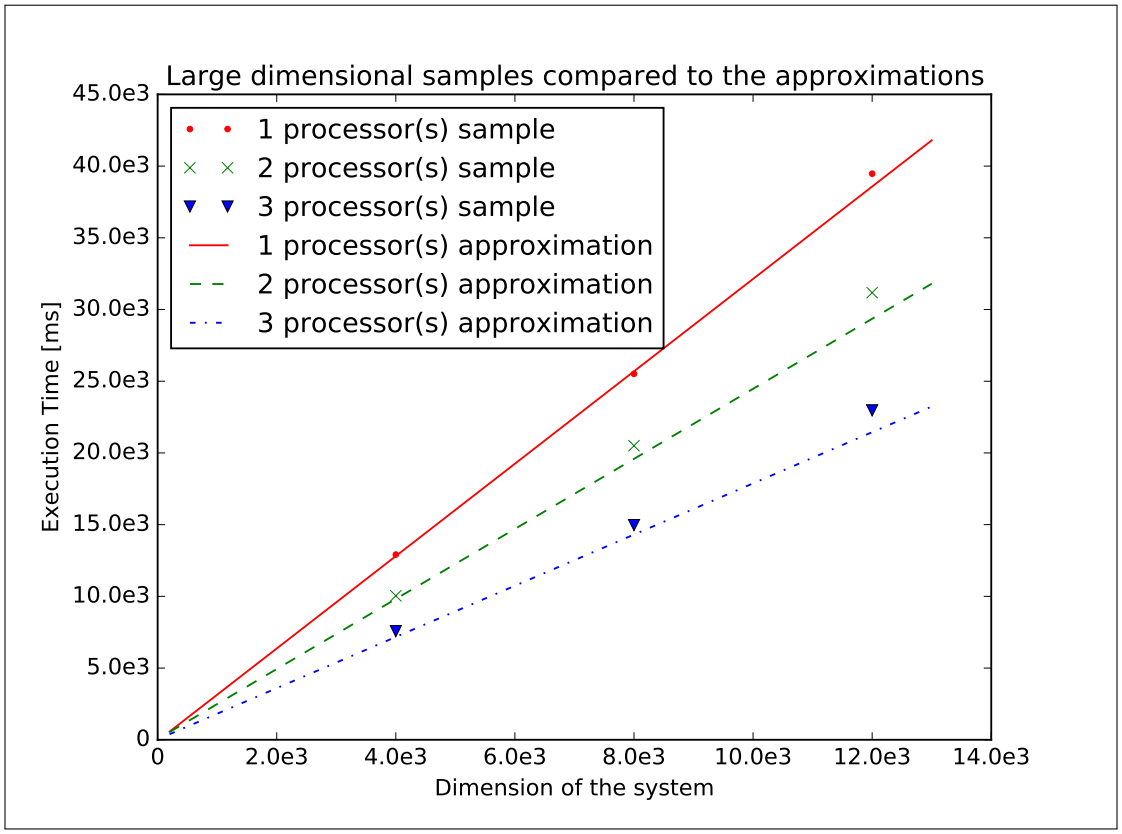

Figure 17: The approximation compared to the large dimensional samples of Table 5.

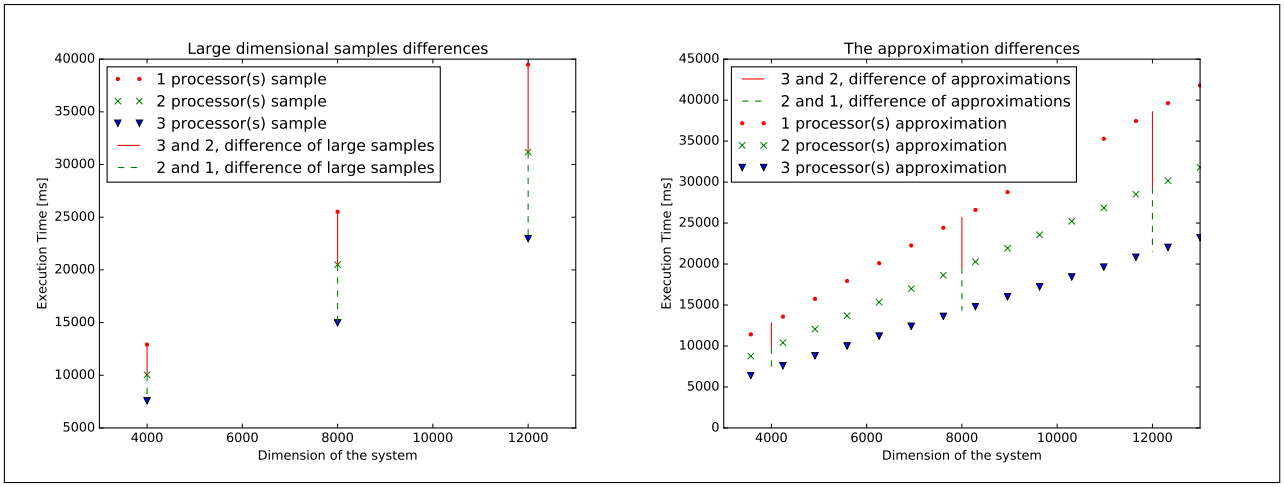

Figure 18: The differences of the large samples of table 5 and the approximations at the corresponding dimensions

8.4 Discussion

In Figure 17, it is apparent that the differences, as visualized in Figure 18, of the improvements between number of processors, stay almost the same whereas execution time for two and three processors increases slightly relative to the approximations. This increase could be because of communication and the increased amount of data that

needs to be transferred. This would explain why for one processor the samples seem to oscillate around the approximation (No communication), whereas for two and three processors the sample points stay above the approximation curves. This phenomenon is not observed in Figure 16, which could be because the dimension is substantially lower and that the amount of data sent is comparatively cheaply, computation wise, compared to the initiation of the transfer. Lastly we note that the increase in Figure 17, for two and three processors, are similar. This implies that the difference is most likely not due to the amount of communication initialization, but rather the amount of data that is being sent. Although several facts point to this theory, one would need to conduct more tests in this larger dimensional interval to confirm it. One way would be to analyze differences of two and three processors and verify that the overall increase in execution time for both of them are monotonically increasing.

If the amount of data transferred is the reason of the increase of execution time, the deviation noted in Figure 17 would be a linear component overlooked in the smaller dimensional range. This would imply that the real approximations would be:

$$\hat{l}_i = l_i + \zeta x, \quad 0 < \zeta << a_i, \ i = 2, 3$$

if l_2, l_3 are the approximation curves for two and three processors and a_2, a_3 are their derivatives. This would imply that the form of curve stays linear and that the difference between approximation and samples of two and three processors are parallel since the difference should not include a linear component. This is clarified in the following graph, Figure 19.

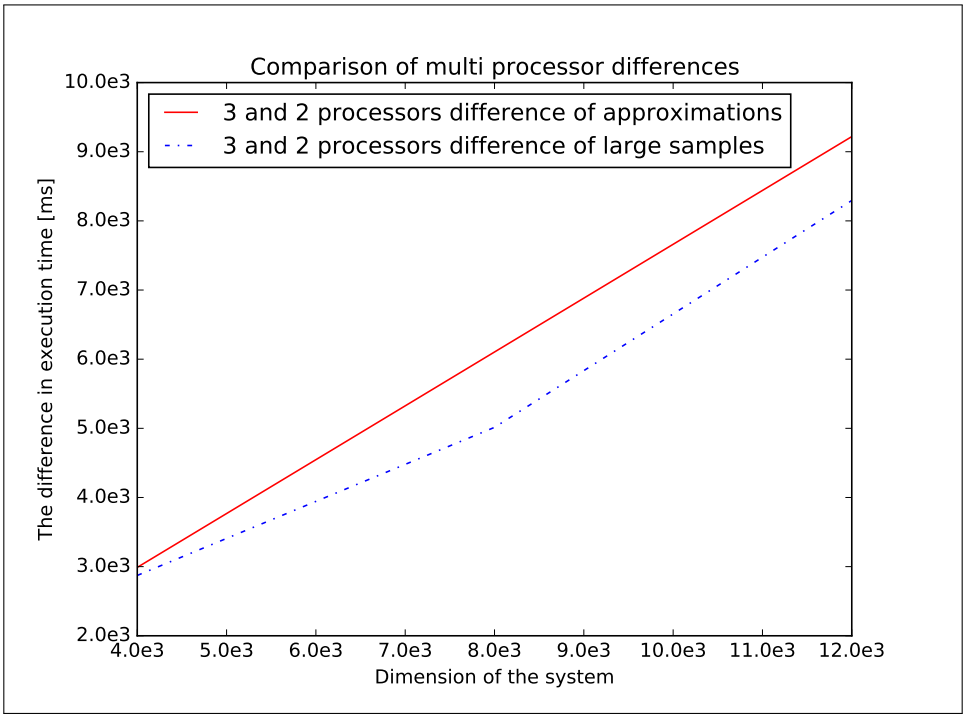

Figure 19: A comparison graph of the normed differences between 2 and 3 active processors for approximations and samples. Here we can see that the lines are fairly parallel which would imply that we have a constant difference between the approximation and real samples.

8.5 Conclusion

Stability for larger systems

Overall the method seems to be fairly stable. The ratio of the improvements seems to stay relatively the same in the interval the tests where conducted in ($[200, 14000]$). The improvement for the smaller systems is similar to that of the larger with slight deviation.

The method's execution time with respect to large dimensions

The larger dimensional problems seem to have a slight perturbation. This perturbation is not large enough to believe it could be quadratic so it is assumed to be linear. This means that the method's execution time should stay linear with respect to dimension.

Efficiency for larger systems

Even though the slight loss of efficiency has been the main topic in the discussion, the method performs substantially better for larger systems when parallelized. Moreover the loss seems to be independent of number of processors introduced, which would imply that introducing more processors would not be affected by this perturbation.

9 Continued work

Because the tests in this report are made on one computer, communication is relatively inexpensive when compared to MPI over multiple computers. It would be interesting to see if this method holds up when using such communication. If it does hold up, one could analyze how large the system has to be in order to have an improvement and maybe even find a ratio of dimension compared to number of computers initialized.

There is going to be a point when the RAM on the computer is exhausted. When this happens, there is going to be a large performance hit and whether or not this method can take the hit and still be efficient compared to not being parallelized is yet to be analyzed. The problem here is that one would need to make sure that the RAM is distributed equally over the different processes, otherwise one would have one or more processes bottlenecking.

Up until now, the systems that have been solved have either been random or rather simple computation wise. These are the systems we assume this method solves due to it working with approximation (we know how the systems act to some extent). What if the systems where constructed to be difficult? We have quite a lot of communication when parallelized and, in each reduction step, our solutions differ due to different rounding errors. We clearly saw this phenomenon in Figure 11 when verifying if the method was working correctly. This slight problem, that was glanced over in the discussion in Chapter 6, could be amplified by the choice of system being solved. It would be interesting to see if there is such a system or family of systems where the method derails and cannot converge to a correct solution when parallelized but does converge when not.

10 Conclusion

It became apparent early in the project that finding a clean way of parallelizing the method was needed. The first revisions were messy because global and local indicies of the parallelized vectors both occurred in the GMRES method and its sub-methods. Due to the confusion, the method had multiple bugs and was nearly unreadable. Because there were major problems, the method had to be re-rewritten several times. The resulting concept of writing a separate class for the parallelized vectors cleaned up the code substantially and therefore solved most of the problems. By having the variables in the method parallelized, most of the code was able to be written as if it was in serial. The implementation of this vector class was a natural step forwards because the partition scheme and the utility functions were written from the start. Writing the vector class was therefore simply collecting all of the discrete functions and putting them into one package. Even though re-writing the method several times was tedious, it was rewarding. The final method's code is easy to read and more importantly, it is easy to check that the separate parts of this method are working.

The parallelized implementation of the GMRES method compared to the serial counterpart reduced execution time substantially for both sparse and dense systems for varied dimensions. The improvement of the parallelized method is greater when more iterations are performed to find better approximations. Both of the previous statements lead to the conclusion that the overall implementation produces an improvement. The cost of this improvement seems to be minor differences in rounding errors, as seen in Figure 11, from reduction steps in cross-communication. This deviation is not a problem because the method restarts after each couple of iterations with a new generated approximation. This new approximation is handled as the initial one and therefore does not allow the error to grow. The overall form of the improvement seems to reflect on the form of running the method serial. Quadratic improvements were seen in dense system and linear in sparse. The improvement seems to stay stable for larger sparse systems that where analyzed and only slight deviations where noted. In summary the implementation is easy to read and validate, brings an overall improvement with respect to the serial case for both sparse and dense matrices, and stays stable for very large sparse systems.

References

[1] The Cayley-Hamilton Theorem 2/6/2016:
`http://mathworld.wolfram.com/Cayley-HamiltonTheorem.html`

[2] Philipp Birken, *Numerical methods for stiff problems*, 2015, p46-48

[3] Householder transformations 2/6/2016:
`https://en.wikipedia.org/w/index.php?title=Householder_transformation&oldid=723064434`

[4] Givens rotations 2/6/2016:
`https://en.wikipedia.org/w/index.php?title=Givens_rotation&oldid=721320896`

[5] An introduction to the MPI protocol. Last checked 22/4/2016:
`https://computing.llnl.gov/tutorials/mpi/`

[6] Information about *csv* documents. Last checked 3/6/2016:
`https://en.wikipedia.org/w/index.php?title=Comma-separated_values&oldid=723065927`

[7] Homer F. Walker, *Implementation of the GMRES Method Using Householder Transformations*, 1988, p152-163

[8] Linear least-square approximations 10/6/2016:
`https://en.wikipedia.org/w/index.php?title=Linear_least_squares_%28mathematics%29&oldid=721711977`

A Appendices

A.1 The code for the GMRES method

```
using namespace std;
using namespace MPI;


template <class matrixclass> class GMRES{
    /**
     *  Takes type of matrix needs to be compatible with regular multiplication
     *  with mpi_doubleVector
     *
     */

    public:
    // stoping residual, Matrix, rhs,dimesion of the system, approximation, restartsize
    int solver(double stopping_res, matrixclass *Matrix, mpi_doubleVector *rhs,int n, mpi_doubleVector *approx, int restartsize){
        const int max_starts=1e2;
        //Get the local segment length
        const int id =COMM_WORLD.Get_rank();
        //The varibles for this method
        int i, j, k=0, start, ready = 0;
        mpi_doubleVector v(n);
        mpi_doubleVector u(n);
        doubleMatrix R(restartsize,restartsize);
        mpi_doubleMatrix H(restartsize,n);
        doubleVector cosinus(restartsize+1);
        doubleVector sinus(restartsize+1);
        doubleVector w(restartsize+1);
        double currenterror;
        //Create a serial linear solver instance
        serialsolver <doubleVector,doubleMatrix> Solver;
        //Do the first multiplication
        v=(*Matrix)*(*approx);
        v = (*rhs)- v;
        double tmp=sqrt(v*v);
        //store the current error
        currenterror=tmp;
        //Generate and apply the first Housholder transformation
        generateHousholder(v,u,0,&w[0]);
        //set all other indicies to 0
        for(int i=0;i<restartsize+1;i++){
            if(i>0){
                w[i]=0;
                }
            }
        //Stop if approximation is good from the start
        if(tmp<stopping_res){
            ready=1;
            }
        //-------------------------------------------------
        //              Number of starts
        //-------------------------------------------------
        for(start = 0; start <= max_starts && !ready; start++){
            //Do the same as above to start the method
            if(start){
                v=(*Matrix)*(*approx);
                v = (*rhs)- v;
                generateHousholder(v,u,0,&w[0]);
                for(int i=0;i<restartsize+1;i++){
                    if(i>0){
                        w[i]=0;
                        }
                    }
                }
        //-----------------------------------
        //          The GMRES Method
        //-----------------------------------
        for(k = 0; k < restartsize; k++){
            //store the current trasformation in H
            H[k]=u;
            //make v an unit vector
            for(j=v.li();j<v.ui();j++){
                v[j]=0;
                }
            v[k]=1;
            //Apply the last k + 1 Householder transformations in reverse order:
            for(i = k; i >= 0; i--){
                applyHouseholder(H[i],v,v,i);
                }
            u=(*Matrix)*v;
            v=u;
            //Apply last k + 1 Householder transformations:
            for(i = 0; i <= k; i++){
                applyHouseholder(H[i],v,v,i);
                }
            //Generate and apply the last transformation
            if(k < n - 1){
                //Let u be the zero vector
                for(i=u.li();i<u.ui();i++){
                    u[i]=0;
                    }
                generateHousholder(v,u,k+1,&tmp);
                /* Apply this transformation: */
                for(int i=v.li();i<v.ui();i++){
                    if(i==k){
                        v[k+1]=tmp;
                        }
                    if(i>k+1){
                        v[i]=0;
                        }
                    }
                }
```

```
            //A double that has the w[k+1]
            double tmp=0;
            //Generate and apply the givens rotations on v and w
            if(id==0){
                givens_rotations(v,w,R,sinus,cosinus,k);
                tmp=w[k+1];
                }
            //Broadcast the current error
            MPI_Bcast(&tmp, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            //store the current error
            currenterror=fabs(tmp);
            //Check if the solution is good enough
            if(fabs(tmp) < stopping_res){
                ready = 1;
                break;
                }
        }
    //-----------------------------------
    //          The Solver
    //-----------------------------------
        if(k==restartsize){
            k--;
            }
        //Solve the triangular system and transfer it to u
        if(id==0){
            doubleVector x=Solver.solveUpperTriangular(R,w,k+1);
            //transfer the solution to u
            for(i=0;i<k+1;i++){
                u[i]=x[i];
                }
            }
        //Calculate the new approximation
        for(i = 0; i <= k; i++){
            //Unit vector
            for(j =v.li();j<v.ui();j++){
                v[j]=0;
                }
            v[i]=1;
            //Apply last i + 1 householder transformations in reverse order:
            for(j = i; j >= 0; j--){
                applyHouseholder(H[j],v,v,j);
                }
            //Get the coeficiant we wish to multiply the vectors with
            double c=0;
            c=u[i];
            //make sure all parts has it
            MPI_Bcast(&c, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            v *= c; //multiply with scalar
            (*approx) += v; //Get the new approximation
            }

        //If the error is small enough stop, else continue
        if(currenterror < stopping_res){
            ready = 1;
            }
        }
        //Check if we have done maximum number of starts
        if(start > max_starts){
            start = max_starts;
            }
        return (start * restartsize + k + 1);
    }
    private:

    void givens_rotations(mpi_doubleVector v,doubleVector w,doubleMatrix R,doubleVector sinus,doubleVector cosinus,int k);

    void generateHousholder(mpi_doubleVector x, mpi_doubleVector out,int k,double *alpha);

    void applyHouseholder(mpi_doubleVector t,mpi_doubleVector x,mpi_doubleVector y,int k);
    //Simple sign function
    int sign(double v);
};
```

LUNFNA-4010-2016

Numerical Analysis
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden
http://www.maths.lth.se/