Abstract

The purpose of this project is to extend parallel Diffpack with an external parallel package called Trilinos. Numerical methods are effective at solving a variety of problems on parallel computers. Some of the functions provided by Trilinos can not be found in Diffpack. This project has made it possible for Diffpack to use these functions in a simple and professional manner. Different parallel performance have also be compared against each other.
Preface

This thesis is written at Simula Research Laboratory. With little knowledge from Scientific Computing, the aim is to get an insight into parallel programming and numerical methods on parallel computers.

Acknowledgements: First of all I would like to thank my supervisor Xing Cai for providing assistance, enthusiasm, and exceptional feedback. Secondly I would like to thank all my fellow master students at SINTEF ICT: André Rigland Brodtkorb, Hanne Moen, Trygve Fladby, Martin Lilleng Sætra and Lars Moastuen for social gatherings and insightful discussions. And thanks to my fellow master students at Simula: Mads Fredrik Hoel, Anders Schau Knatten, Martin Burheim Tingstad and Magnus Vikstrøm for Diffpack experience sharing and keeping me company on late nights at Simula.

Finally, I would like to thank my family, especially my mother and father for continued love and support that helped me throughout the project.

Oslo,
July 2007
Guo Wei Ma
Contents

Abstract iii

Preface v

1. Introduction 1

2. Introduction to Parallel Computing 3
   2.1. Parallel Speedup 4
   2.2. Parallel Efficiency 4
   2.3. Parallel Overhead 4

3. Basics of Trilinos 5
   3.1. Iterative Solvers 5
   3.2. Preconditioner 6
      3.2.1. AztecOO 6
      3.2.2. IFPACK 7
      3.2.3. ML 8
   3.3. Convergence Monitor 9
   3.4. Usage of Trilinos 9

4. Extending Diffpack 13
   4.1. Diffpack simulator 13
   4.2. Design and Implementation 14
      4.2.1. TrilinosItSolver 14
      4.2.2. TrilinosPrecond 19
      4.2.3. TrilinosMonitor 21
      4.2.4. TrilinosPrm 22
   4.3. Integration and Obstacles 22
      4.3.1. Trilinos Files 25
      4.3.2. Behaviour of Trilinos 26
   4.4. A Diffpack example 27
Chapter 1

Introduction

The aim of this thesis is to extend parallel Diffpack [12] with Trilinos [15]. The area of research is in the field of Scientific Computing. Diffpack has been extended with Trilinos before, but only in minor and unofficial manners. It has never been an extension with Trilinos as extensive as the one provided in this thesis.

We want to extend Diffpack with iterative solvers, preconditioners and convergence monitors from Trilinos. Along the process this thesis will answer the following questions:

1. Is it possible to extend Diffpack in an elegant and flexible fashion?
2. Is there any profit integrating Trilinos into Diffpack?

The motivation for this thesis is to investigate two parallel packages which are respectively independent of each other, but yet have resemblance. Question no. 1 investigates the possibility of extending Diffpack with Trilinos functionalities. What are the obstacles during this process? At the end it is desirable to measure their performance, both pure Diffpack, pure Trilinos and combinations. The main goal of parallel computing is to save time. Is there any time saved by extending Diffpack? If there are functions in Trilinos not included in Diffpack, it is also desirable to use these as though they are Diffpack functions.

This thesis is organized as follows. Chapter 2 gives an introduction to parallel computing. Some of the most used concepts are introduced for use in later chapters. Subjects like speedup, overhead and efficiency are introduced.

Chapter 3 gives an overview of Trilinos. The most well-known packages and those used in this thesis are mentioned. Then a simple Trilinos
code example is outlined to give readers an introduction on how Trilinos works.

Chapter 4 explores Diffpack and investigates the possibility for extending it with an external package. Topics discussed are: design, anatomy of Diffpack simulators, integration of Trilinos, and at the end a Diffpack simulator is introduced.

Chapter 5 will contain measurements. Concepts introduced in Chapter 2 will be examined. Measurements give an overview of how well Diffpack and Trilinos work. This chapter answers the question if there is any profit extending Diffpack with Trilinos.

And at last Chapter 6 summarizes the thesis, reviewing the most important findings and future work.
Chapter 2

Introduction to Parallel Computing

Traditionally, software has been written for serial computation. It is meant to be executed by a single computer having a single CPU. Only one instruction can be executed at any moment in time. Parallel computing does not have to follow this pattern. It can work with several instructions at the same time.

The motivations behind parallel computing are saving time and solving larger problems. Parallel programming focuses on partitioning the overall problem into separate tasks, allocating tasks to processors to get meaningful results. Single computers have finite memory resources. For large problems, using multiple computers memories may overcome this obstacle. Other reasons for parallel computing are using available compute resources on a wide area network and using multiple “cheap” computing resources instead of paying for time on a supercomputer.

The Message Passing Interface [4] (MPI) is a standard API (Application Programming Interface) that can be used to create parallel applications. Both Trilinos and Diffpack are based on MPI. However, no solid knowledge of MPI is required for an user of these packages. For Trilinos only the visible MPI routines are `MPI_Init` and `MPI_Finalize`. For Diffpack no MPI routines are visible at all. The purpose is to make parallel programming easier and more efficient.
2.1 Parallel Speedup

The parallel speedup ($S_p$) is defined as the length of time it takes a program to run on a single processor divided by the time it takes to run on multiple processors ($p$):

$$S_p = \frac{\text{time for serial execution}}{\text{time for parallel execution}} = \frac{t_s}{t_p}$$

where $t_s$ is the serial run time and $t_p$ is the parallel run time for $p$ processors. $S_p$ indicates how much performance gain is achieved by parallelizing the application and running it in parallel on multiple processors.

2.2 Parallel Efficiency

Efficiency ($e_p$) is a measure of parallel performance that is closely related to speedup and is defined as the ratio of parallel speedup ($S_p$) to $p$ processors:

$$e_p = \frac{S_p}{p} = \frac{\text{sequential time}}{p \cdot \text{parallel time}} = \frac{t_s}{p \cdot t_p}$$

2.3 Parallel Overhead

Parallel overhead is the amount of time required to coordinate parallel tasks, as opposed to doing computational work. The overhead can be affected by following factors:

- Task start-up time
- Synchronizations
- Data communications
- Task termination time

More detailed description about speedup, efficiency and overhead can be found in [21, Ch.1.5] and [8, Ch.5].
Chapter 3

Basics of Trilinos

This chapter will explore the spectrum of functions available in Trilinos. Trilinos is an iterative linear solver package provided by Sandia National Laboratories [3]. Trilinos runs on a wide variety of platforms, including Linux, SGI64, DEC, and Solaris.

Basically Trilinos is a collection of packages. Each package is focused on algorithms in its problem domain. Trilinos offers: Distributed linear algebra objects, Krylov solvers, ILU-type preconditioners, Smoothed aggregation multilevel preconditioners and many more. For better overview, see [10].

Within Trilinos there are over twenty packages. To learn and understand all of those will be an enormous and time-consuming challenge and out-of-scope of this thesis. However, to get familiar with some of the most common and well known packages and be able to utilise them in practical use should be reasonable. The focus area will be the iterative solvers, preconditioners and convergence monitors. Integration with Diffpack will be outlined in Chapter 4.

3.1 Iterative Solvers

Table 3.1 on the following page gives an overview of parallel iterative solvers provided by Diffpack and Trilinos. In Trilinos the iterative solvers are provided by the package AztecOO [9]. From the table it is obvious Diffpack has all the solvers of Trilinos along with some others. From this point of view there is no advantage in using solvers from Trilinos, unless Trilinos proves to be much faster than Diffpack. This will be discussed in Chapter 5.

AztecOO presents only iterative solvers of the Krylov methods. An introduction to the basic idea of Krylov methods can be found in [16]. Krylov methods are among the most successful methods currently available in numerical linear algebra. Some of the reasons are that they include matrix-
vector and inner product operations which are good suited for parallel computing.

<table>
<thead>
<tr>
<th>Basic sol.</th>
<th>Diffpack</th>
<th>Trilinos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Multilevel</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Successive over relaxation</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Symmetric successive over relaxation</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Krylov solver</th>
<th>Diffpack</th>
<th>Trilinos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bi-Conjugate gradient</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Conjugate gradient</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Conjugate gradient squared</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Generalized minimal residual</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Minimum residual</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Orthomin</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Symmetric minimum residual</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Symmetric LQ</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Transpose-free quasi-minimal residual</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 3.1: Iterative solvers of Diffpack and Trilinos.

3.2 Preconditioner

For readers not familiar with preconditioners, this section outlines the idea of a preconditioner, followed by a short description of some of the preconditioners available in Trilinos. More details and further interest in preconditioners can be found in [5, Ch.3] and [13, Appendix C.3].

The convergence rate of an iterative solver depends on the spectral properties of the coefficient matrix. A preconditioner $M$ of a matrix $A$ results in that $M^{-1}A$ has a smaller condition number than $A$. The original linear system becomes:

$$Ax = b \implies M^{-1}Ax = M^{-1}b$$

The preconditioned linear system will have the same solution as the original $Ax = b$, but is better conditioned. A smaller condition number has a better convergence rate. If the cost of computing time and applying $M^{-1}$ is small, a gain in efficiency is achieved. Some of the preconditioner packages in Trilinos are: AztecOO [9], IFPACK[18] and ML[7].

3.2.1 AztecOO

Preconditioners within AztecOO are Jacobi, polynomial preconditioner and domain decomposistion preconditioner.
3.2. Preconditioner

*Jacobi preconditioner* is the simplest preconditioner, it consists of only diagonal coefficients of the matrix. $M^{-1}$ is straightforward to compute.

$$M = m_{i,j} = \begin{cases} a_{i,j} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

*Polynomial preconditioner* is direct approximations of the inverse of the coefficient matrix. The preconditioner is defined by $M^{-1} = P(A)$ where $P$ is a polynomial, typically of low degree. The original system is replaced by the preconditioned system:

$$Ax = b \implies P(A)Ax = P(A)b$$

The linear system can be solved by for instance the conjugate gradient iteration. Trilinos provides two polynomial preconditioners: Neumann and Least-square. Default polynomial order is 1.

*Domain Decomposition preconditioner* is based on decomposing the domain into a number of either non-overlapping or overlapping subdomains. The main idea is to obtain more parallelism at the subdomain level. The advantage of this approach is the use of different methods within different subdomains. A large problem can be dissected into several small problems and solved on different processors. The use of domain decomposition is a popular choice for many large scale applications. AztecOO provides one level overlapping domain decomposition preconditioner. Subdomain solvers are listed in Table 3.2. Reading material about domain decomposition preconditioner can be found in [11].

3.2.2 IFPACK

IFPACK is used to define point and block relaxation preconditioners, various incomplete factorization and one-level additive Schwarz preconditioners with overlap. IFPACK provides subdomain solvers such as incomplete Cholesky factorization and Incomplete LU factorization. The exact LU factorization of the local submatrix is through the Amesos [17] package. For a complete list of preconditioners in IFPACK, see Table 3.2. The user can adopt a minimal-overlap (that is, zero-row overlap), or ask IFPACK to extend the overlap. The resulting preconditioner reads:

$$P^{-1}_{\text{IFPACK}} = \sum_{i=0}^{\text{NumProcs}-1} P_i A_i^{-1} R_i$$

where $R_i$ is the restriction operator from the global vector to the overlapping subdomain $i$, and $P_i$ is the prolongator operator. IFPACK preconditioners can also be used as smoothers for multilevel solvers in the package ML.
3.2.3 ML

The ML package is based on multilevel methods (more mathematical details about multilevel can be found in [6]). The ML package provides multilevel solvers and preconditioners based on geometric and algebraic coarsening schemes. Some main schemes for ML are: Smoothed aggregation, Edge-element AMG for Maxwells equations and n-level (smoothed) aggregation domain decomposition and more. For more details see [7].

To sum up, the preconditioners in Trilinos are listed in table below. Commands are for internal Trilinos use.

<table>
<thead>
<tr>
<th>Name</th>
<th>Command</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi</td>
<td>AZ_Jacobi</td>
<td>k step Jacobi. Default step: 1</td>
</tr>
<tr>
<td>Neumann</td>
<td>AZ_Neumann</td>
<td>Polynomial, default order: 1</td>
</tr>
<tr>
<td>Least-square</td>
<td>AZ_ls</td>
<td>Polynomial, default order: 1</td>
</tr>
<tr>
<td>Sym.Gauss-Siedel</td>
<td>AZ_sym_GS</td>
<td>Non-overlapping domain decomposition k step symmetric Gauss-Siedel. Default k step: 1</td>
</tr>
<tr>
<td>DD</td>
<td>AZ_dom_decomp</td>
<td>Domain Decomposition preconditioner. Subdomain solvers are: ilut, ilu, rilu, blu and icc. Details can be found in [9].</td>
</tr>
<tr>
<td>Point Relaxation</td>
<td>Ifpack_PR</td>
<td>Point (damped) relaxation preconditioner for Jacobi(default), Gauss-Seidel, or symmetric Gauss-Seidel.</td>
</tr>
<tr>
<td>Block Relaxation</td>
<td>Ifpack_BR</td>
<td>Block relaxation preconditioner. Similar properties as IfpackPR.</td>
</tr>
<tr>
<td>Incomplete Cholesky</td>
<td>Ifpack_IC</td>
<td>Incomplete Cholesky factorization with dropping based on the level-of-fill of the graph.</td>
</tr>
<tr>
<td>Incomplete LU</td>
<td>Ifpack_ILU</td>
<td>Incomplete LU factorization with dropping based on the level-of-fill of the graph.</td>
</tr>
<tr>
<td>Complete LU</td>
<td>Ifpack_Amesos</td>
<td>Complete LU factorization on each subdomain.</td>
</tr>
<tr>
<td>Multi-Level</td>
<td>using Teuchos[2] parameterlist</td>
<td>Smoothed aggregation, Edge-element AMG for Maxwells equations and n-level (smoothed) aggregation domain decomposition. Detailed usage, see [7].</td>
</tr>
</tbody>
</table>

Table 3.2: Preconditioners in Trilinos
3.3 Convergence Monitor

The convergence monitors available in Trilinos are listed in Table 3.3. The iterative solver terminates if the corresponding residual is less than or equal to $\varepsilon$. The convergence monitors are provided by the AztecOO package.

<table>
<thead>
<tr>
<th>Convergence criteria</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>$| r^k | / | r^0 |$</td>
<td>AZ_r0</td>
</tr>
<tr>
<td>$| r^k | / | b |$</td>
<td>AZ_rhs</td>
</tr>
<tr>
<td>$| r^k | / | A |_{\infty}$</td>
<td>AZ_Anorm</td>
</tr>
<tr>
<td>$| r^k |<em>{\infty} / (| A |</em>{\infty} * | x^k |<em>1 + | b |</em>{\infty})$</td>
<td>AZ_noscaled</td>
</tr>
</tbody>
</table>

Table 3.3: Convergence monitors in Trilinos, $r^k = Ax^k - b$.

3.4 Usage of Trilinos

One of the first tasks when approaching Trilinos is to get familiar with the package Epetra [1]. This package defines the basic classes for distributed matrices, vectors and linear problems. Epetra is the common language that all the Trilinos packages work with, either as input or as partial result. This functionality proves to be quite handy, and allows powerful combinations across the Trilinos platform. The sequence order for Epetra object construction is shown in Figure 3.1.

![Figure 3.1: Construction of Epetra objects. Arrows indicate that the construction is one-way process.](image)

Epetra_Comp is a communicator class, it contains specific information about the parallel machine we are using. Epetra_Comp supports serial, MPI and prototype hybrid MPI/threaded parallel programming models.

Epetra_Map is a map which has the sole purpose of holding on the relation between the local and global points distributed across the processors.
Throughout this thesis points are referred to as grid points in the computational mesh, unless something else is explained.

The next step is to create Trilinos matrix and vectors, which are Epetra_CrsMatrix and Epetra_Vector respectively. These objects are as mentioned earlier the common “language” for all other Trilinos packages.

After the Epetra objects construction, the linear system $Ax = b$ can either be solved by an iterative solver in AztecOO, or added preconditioners from section 3.2. Below we show an example (main.cpp) of creating a tridiagonal matrix based on Epetra objects just introduced. The example should be straightforward and most of the commands are quite similar to standard C++.

---

**main.cpp**

```cpp
#include <iostream>
#include <Epetra_Comm.h>
#include <Epetra_Map.h>
#include <Epetra_BlockCrsMatrix.h>
#include <Epetra_Vector.h>
#include <AztecOO.h>

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv); // Initialize MPI, MpiComm
    Epetra_MpiComm Comm( MPI_COMM_WORLD );

    // ***** Map puts same number of equations on each proc *****
    int NumMyElements = 100;
    Epetra_Map Map(-1, NumMyElements, 0, Comm);
    int NumGlobalElements = Map.NumGlobalElements();

    // ***** Create an Epetra_Matrix tridiag(-1,2,-1) *****
    Epetra_CrsMatrix A(Copy, Map, 3);
    // Add rows one-at-a-time
    // Need some vectors to help
    // Off diagonal Values will always be -1, diagonal term 2
    double *Values = new double[2];
    Values[0] = -1.0; Values[1] = -1.0;
    int *Indices = new int[2];
    two = 2.0;
    int NumEntries;
    for( int i=0; i<NumMyElements; ++i ) {
        if (MyGlobalElements[i]==0) {
            Indices[0] = 1;
            NumEntries = 1;
        } else if (MyGlobalElements[i] == NumGlobalElements-1) {
            Indices[0] = NumGlobalElements-2;
            NumEntries = 1;
        } else {
            Indices[0] = MyGlobalElements[i]-1;
            NumEntries = 2;
        }
        A.InsertGlobalValues(MyGlobalElements[i], NumEntries, Values, Indices);
    }
    // Put in the diagonal entry
    A.InsertGlobalValues(MyGlobalElements[i], 1, &two, MyGlobalElements+i);
    A.FillComplete(); // Transform from GIDs to LIDs
}
```

---

Chapter 3. Basics of Trilinos
3.4. Usage of Trilinos

```
// ***** Create x and b vectors ****
Epetra_Vector x(Map);
Epetra_Vector b(Map);
b.Random(); // Fill RHS with random on the interval (-1.0, 1.0)

// ***** Create Linear Problem *****
Epetra_LinearProblem problem(&A, &x, &b);

// ***** Create/define AztecOO instance, solve *****
AztecOO solver(problem);
Solver.SetAztecOption(AZ_solver, AZ_cg);
solver.SetAztecOption(AZ_precond, AZ_Jacobi);
solver.Iterate(1000, 1.0E-8);

// ***** Report results, finish ************
cout << "Solver performed " << solver.NumIters() << " iterations."
    << " Norm of true residual = 
    " << solver.TrueResidual() << endl;

MPI_Finalize();
return 0;
```

**Line 7:** Define the local problem dimension. `NumMyElements` will be used to define an `Epetra_Map` that has 100 elements on each processor.

**Line 8:** Constructs an `Epetra_Map` object that has `NumMyElements` elements spread across the parallel (or serial) machine.

**Line 13:** Creates an `Epetra_CrsMatrix`.

**Line 24-46:** These lines insert values and indices into the matrix just instantiated. The matrix is tridiagonal with a value of 2 at each diagonal and -1 on the immediate off-diagonals.

**Line 51-53:** Once the matrix is constructed, `Epetra_Vector b` and `x` are created using the same map that determined the layout of the matrix rows. Fill `b` with random values.

**Line 56:** Now that `A`, `x` and `b` are formed, a linear problem is defined. This object will be used to define a `AztecOO` instance.

**Line 60-62:** Once the solver object is created, The conjugate gradient solver is chosen with no preconditioner. Next, call the `Iterate()` method, passing in the maximum number of iterations that can be performed and a tolerance that should be used to test for convergence.

**Line 65-69:** Print results.
For deeper understanding and training in Trilinos, see [19].

**Definition 1-to-1 Map:** A map is 1-to-1 if each global id(GID) appears only once in the map and is therefore associated with only a single processor.

Some objects in Trilinos require a 1-to-1 map. The most important ones are *Domain_Map* and *Range_Map*. *Domain_Map* is the mapping of domain objects, typically the $x$ vector in a $Ax = b$, while the *Range_Map* is the $b$ vector mapping. Since points can be represented in one processor only means that each row in a linear system of $Ax = b$ is unique and can appear only once among the processors.
Chapter 4

Extending Diffpack

This chapter answers one of the main questions:

1. Is it possible to extend Diffpack in an elegant and flexible fashion?

In order to give a detailed answer, this chapter is organized as follows. A section about Diffpack simulators is introduced. Design and implementation have to be thoroughly evaluated to avoid bottlenecks; this is described in the second section. The third section mentions some of the obstacles during the extending process. The final section will outline a Diffpack simulator example.

4.1 Diffpack simulator

For readers not familiar with Diffpack, an introduction can be found in [13, Ch.1]. When extending Diffpack, we want the implementation of Trilinos to be as easy for the user as possible. For a Diffpack user it is not required to have any knowledge of Trilinos, only a list of functions available. The user can choose Trilinos functions as though they are Diffpack functions. The basic idea to this approach is to explore Diffpack simulators from [13]. Most of the simulators that solve a PDE problem can roughly be divided into two parts: the first part contains information about the PDE, boundary conditions, the discretization and etc. to generate a linear system of form \( Ax = b \), the second part solves the generated linear system \( Ax = b \). Trilinos cannot discretize PDEs by itself; a powerful combination is to let Trilinos solve the generated linear system from Diffpack. We assume in this chapter that PDE problems are scalar PDE.

Originally, Diffpack simulators are meant for serial execution. Details about how to convert to parallel execution will not be outlined here. For
further interest and introduction to parallel Diffpack can be found in [14, Ch.1]. Although it is not mandatory to read this, it is still recommended to gain a better understanding of the parallel module of Diffpack. Most of the Diffpack simulators have the same structure:

- A global MenuSystem object that administers the call for setting up the menu and initializing the simulator.
- An instance of LinEqAdmFE that has a relation to the solver, preconditioner and the linear system.
- A define function that establishes the menu system and makes it visible for the user.
- A scan function that reads the parameters from the menu system and rebinds objects.
- The matrix storage format is compressed row. This format is best suited for matrices.

There are several methods to feed Diffpack simulators; direct input with keyboard during execution, using an input file, and a graphical user interface with input file. The input file contains information about what sort of matrix to use, type of solver to choose, the maximum number of iterations and so on. All the methods use a MenuSystem that reads the input file. An example of an input file is explained in Chapter A.1.

4.2 Design and Implementation

Since Diffpack contains a set of libraries, it is an important criterion not to modify any of these but rather build the Trilinos extension on top of Diffpack. It is desirable to code as little as possible; at the same time keep the quality and flexibility to a reasonable level.

The most profitable way to extend Diffpack is to make Trilinos classes look as similar to Diffpack classes as possible. In other words Trilinos codes are wrapped into Diffpack.

4.2.1 TrilinosItSolver

Iterative solvers of Trilinos contain methods from Krylov subspace only. For this reason it is intuitive to implement Trilinos iterative solvers as an extension to the KrylovItSolver class of Diffpack. This design is implemented as shown in Figure 4.1. By this, all the Trilinos solvers will have the same constructor as Diffpack, and can be used as a Diffpack solver. TrilinosItSolver has the same functionalities as a Diffpack solver; it solves a linear system of Diffpack, but this class uses solvers from Trilinos. The
solve function accepts a Diffpack object as parameter:

```cpp
TrilinosItSolver::solve(LinEqSystem &system)
```

When this function is called Trilinos starts working with the `LinEqSystem` object. A communicator and a map are assigned in this class, this is because no other Trilinos objects need the map beside the distributed matrix and vectors.

As mentioned in Section 3.4 Trilinos depends on a “1-to-1 map”. The linear system $Ax = b$ from Diffpack simulators can not be fed directly into Trilinos, some of the overlapping points have to be removed. Besides, Trilinos works only with Trilinos objects. Diffpack matrix and vectors can not be solved without conversion to Trilinos objects. We will outline how to construct Trilinos map, matrix and vector based on data given by Diffpack.

**Mapping**

First thing to do is to create a Trilinos communicator (`Epetra_Comm`) then a Trilinos map (`Epetra_Map`). For the map construction, the class `GridPart` is used. `GridPart` is a Diffpack class that holds the information about sub-grids in parallel computing. Two of the arrays within `GridPart` are:

```cpp
VecSimplest(VecSimple(int)) global_nrs
VecSimplest(VecSimple(int)) ib_node_ids
```

These are “array of arrays”, but this is not essential. Array `global_nrs` contains information about all the global points on their own processor, while `ib_node_ids` contains information about the interior boundary points.
By subtracting \text{ib\_node\_ids} from \text{global\_nnrs} the result should give an array of global points with no overlap, which is exactly what Epetra\_Map needs. We call this temporary difference array for \text{myGlobalPoints}. The way of thinking is illustrated in Figure 4.2.

![Figure 4.2: Construction of global mapping. Array myGlobalPoints is a result of deduction of ib\_node\_ids from global\_nnrs. Array myGlobalPoints contains no overlap among the processors.](image)

Together with the map construction, another array called \text{myLocalPoints} is created, see Figure 4.3. This is a global array of TrilinosItSolver, the purpose of this array is to make matrix and vector conversions from Diffpack more efficient.

![Figure 4.3: Construction of myLocalPoints. All index values are local. Array myLocalPoints is an array with all values not included in array ib\_node\_ids.](image)
4.2. Design and Implementation

Converting Matrices

We assume that for sparse matrix, Diffpack matrix storage format is compressed row. Reading material about this storage format can be found in [13, p.787-788].

The Trilinos matrix type used in this thesis is Epetra_CrsMatrix. A Diffpack matrix has to be converted to a Trilinos matrix. Why convert a matrix from one format to another if both are stored in compressed row? Reason for this conversion is quite simple: Diffpack matrix contains some row overlap. We want to remove all overlapping rows, such that Trilinos matrix can have “1-to-1 map”. The process is illustrated in Figure 4.4 and explained in Algorithm 4.2.1. Array myLocalPoints is as mentioned earlier a help array, numMyPoints is the number of points in their own processor, which is also the length of myLocalPoints. When converting the Diffpack matrix, myLocalPoints will help to loop through the necessary indices. Arrays irow, jcol and A are fetched from Diffpack. EpetraA is the Trilinos Epetra_CrsMatrix. To add values into Epetra_CrsMatrix we have to get help from two other temporary arrays, Values and Indices.

![Figure 4.4: Converting Diffpack matrix to Trilinos matrix. Local row number 1 and 3 are marked. We will explain row number 1 (blue dashed rectangles). Values of myLocalPoints[0] gives 3, goto irow[2]. Number of entries in this row is irow[3]-irow[2]= 2. Value of irow[2] = 6, which tells that columns indices in this row are in jcol[5] and jcol[6], store these indices in array Indices. Matrix values in this row are in A[5] and A[6], store these values in array Values. Insert these into the Trilinos matrix. See Algorithm 4.2.1.](image_url)
Algorithm 4.2.1: ADDMATRIX(LinEqMatrix &A)

\[
\begin{align*}
&\text{for } i \leftarrow 0 \text{ to } \text{numMyPoints} \\
&\quad \text{localRow} = \text{myLocalPoints}[i] \\
&\quad \text{numOfEntries} = irow[\text{localRow} + 1] - irow[\text{localRow}] \\
&\quad \text{for } j \leftarrow 0 \text{ to } \text{numOfEntries} \\
&\qquad \text{Indices}[j] = jcol[\text{localRow} + j] \\
&\qquad \text{Values}[j] = A(\text{localRow} + j) \\
&\quad \text{EpetraA.InsertMyValues}(i, \text{numOfEntries}, \text{Values}, \text{Indices}) \\
&\text{comment: Trilinos matrix, values are added one row at time.}
\end{align*}
\]

The Algorithm 4.2.1 is able to remove all the unnecessary rows. Making the matrix “1-to-1 map”. Trilinos matrix class documentation can be found in [1].

Converting Vectors

The reason for converting vectors is the same as for converting matrices; removing overlapping rows. The Trilinos vector type used is Epetra_Vector. In order to convert Diffpack vectors to Trilinos vectors, the myLocalPoints array is also used here. The procedure is illustrated in Figure 4.5 and explained in Algorithm 4.2.2.

Algorithm 4.2.2: ATTACH(LinEqVector &x, LinEqVector &b)

\[
\begin{align*}
&\text{DiffpackVecX} = x.\text{getVec()} \\
&\text{DiffpackVecB} = b.\text{getVec()} \\
&\text{for } i \leftarrow 0 \text{ to } \text{numMyPoints} \\
&\quad \text{TrilinosVecX}(i) = \text{DiffpackVecX}(\text{myLocalPoints}[i]) \\
&\quad \text{TrilinosVecB}(i) = \text{DiffpackVecB}(\text{myLocalPoints}[i])
\end{align*}
\]

Figure 4.5: Convert Diffpack vector to Trilinos vector.
To convert the Trilinos solution vector back to Diffpack:

**Algorithm 4.2.3: CONVERTTODIFFPACK(LinEqVector &x)**

```
DiffpackVecX = x.getVec()
for i ← 0 to numMyPoints
    DiffpackVecX(myLocalPoints[i]) = TrilinosVecX(i)
comm_adm− > updateInteriorBoundaryNodes(DiffpackVecX)
```

### Solving Linear system

After the matrix and vector conversions, the AztecOO solver is created. Before the actual iteration, solver type and optional preconditioner are selected. The iteration is a pure Trilinos function.

```
AztecOO::Iterate (int MaxIters, double Tolerance)
```

After the iteration, the solution vector is converted back to Diffpack format (Algorithm 4.2.3). But this vector is a smaller version of the overlapped Diffpack vector. The interior boundary points are updated by a call to a function in the SubdCommAdm class of Diffpack:

```
SubdCommAdm::updateInteriorBoundaryNodes(LinEqVector &x_)
```

#### 4.2.2 TrilinosPrecond

The Trilinos preconditioners are designed as an extension to the Precond class of Diffpack, see Figure 4.6. What makes the Trilinos preconditioners implementation unique is the flexibility. It can handle solvers from Trilinos and Diffpack. For instance, a Trilinos polynomial preconditioner such as the Least-square can be applied to the conjugate gradient solver which is member of either Trilinos (AZcg) or Diffpack (ConjGrad). Table 3.2 on page 8 shows the strength of Trilinos, it contains several preconditioners not included in Diffpack. Available preconditioners in Diffpack are described in [13, Appendix C.3].

When using a Diffpack solver, it will first initialize the preconditioner, calling the function init. In this function the declaration of TrilinosItSolver is assign and given matrix attached. For each iteration Diffpack will call the apply function:

```
TrilinosPrecond::apply (const LinEqVector & c,
LinEqVector & d,
TransposeMode tmode = NOT_TRANSPOSED)
```

This function computes the result by applying the current preconditoner to a Diffpack LinEqVector c. The result is returned in d. In Trilinos details: the vectors c and d are attached to the solver object of TrilinosPrecond.
Both \( c \) and \( d \) have to be converted to Trilinos vectors before applying preconditioning. The result vector is then converted back to Diffpack vector (Algorithm 4.2.3). This process occurs for each iteration using Diffpack solver.

When using Trilinos solver, neither init or apply functions are called. For TrilinosItSolver to determine the type of preconditioner to use, only an integer is enough. In other words, the only requirement from TrilinosPrecond is a public function that returns an integer. Since TrilinosPrecond is extended from Precond, the function getStorage is used.

\[
\text{TrilinosPrecond::getStorage()} \quad \text{const}
\]

For applying an option to the AztecOO object the SetAztecOption function is used:

\[
\text{AztecOO::SetAztecOption(int option, int value)}
\]

Below shows an example of the idea.

```c
//az_aztec_defs.h
#define AZ_precond 2
#define AZnone 0
```
4.2. Design and Implementation

4.2.3 TrilinosMonitor

The Trilinos convergence monitors are extended from the ConvMonitor class of Diffpack, see Figure 4.7.

![Figure 4.7: Trilinos convergence monitors.](image)

The idea behind the Trilinos monitors is simple. It can only work with Trilinos solvers. If the solver is an instance of Trilinos, apply the given convergence monitor. For TrilinosItSolver to determined the type of convergence monitor to use, only an integer is enough. Meaning that we only need an integer from one of the public functions in TrilinosMonitor. The function getStorage returns an integer. The idea is the same as TrilinosPrecond.

```cpp
//TrilinosPrecond.cpp
real TrilinosPrecond::getStorage() { return AZnone; }

//TrilinosItSolver.cpp
TrilinosPrecond prec;
int prectype = (int)prec.getStorage();
problem.SetAztecOption(AZ_precond, prectype);
```

```cpp
//az_aztec_defs.h
#define AZ_conv 3
#define AZr0 0

//TrilinosMonitor.cpp
real TrilinosMonitor::getStorage() { return AZr0; }

//TrilinosItSolver.cpp
TrilinosMonitor monitor;
int monitortype = (int)monitor.getStorage();
problem.SetAztecOption(AZ_conv, monitortype);
```

However, if the solver is an instance of Diffpack, the Trilinos monitor will display a warning that says only Trilinos solver can be used with this
convergence monitor, and alter the monitor to default Diffpack monitor CMRelResidual.

4.2.4 TrilinosPrm

All options that can be changed runtime by the input file are built based on “Prm” classes. These are the classes that create the Diffpack objects. For instance, LinEqSolver_prm creates the LinEqSolver object. In order to make Trilinos objects in the same fashion, three new classes have been implemented: TrilinosLinEqSolver_prm, TrilinosPrecond_prm and TrilinosConvMonitor_prm, see Figure 4.8.

![Figure 4.8: Extension of Prm classes.](image)

TrilinosLinEqSolver_prm has a constructor that adds TrilinosLinEqSolver in Figure 4.1 as a part of Diffpack LinEqSolver. Has also a create function that makes it possible to create objects in Diffpack fashion.

TrilinosPrecond_prm adds TrilinosPrecond in Figure 4.6 as a part of Diffpack Precond, and makes Trilinos preconditioner acceptable from input file. Has a defineStatic function that adds a submenu and items to the MenuSystem, has also a scan function that reads the attributes from MenuSystem.

TrilinosMonitor_prm adds TrilinosConvMonitor from Figure 4.7 as a part of Diffpack ConvMonitor. Besides create function, TrilinosMonitor_prm has also a scan function that check if solver is an instance of Diffpack or not, described in Section 4.2.3.

4.3 Integration and Obstacles

To solve a linear problem in Trilinos the user has to create the \( A \) matrix, and the \( x \) and \( b \) vectors. In most cases this will be a difficult task that depends on the problem. This is where we can take the advantages of Diffpack. When a simulator has generated a linear system \( Ax = b \), the system can either be solved in Diffpack, Trilinos or a combination of both.
A Diffpack simulator with Trilinos extensions implemented in Section 4.2 has the class hierarchy as shown in Figure 4.9. Solid lines indicate class derivation ("is a" relationship), whereas dashed lines indicate data members ("has a" relationship). Simulators accept an input file at runtime. The goal is to add Trilinos options to this file, and make these options acceptable and run Trilinos packages as if they were a part of Diffpack.

The challenging task is to make Trilinos preconditioners work as Diffpack preconditioners. This enlarges Diffpack with a suite of new preconditioners. Along with new preconditioners, it is also desirable to choose different options within each preconditioner. It is confusing to use the same parameter in Diffpack Precond_prm class for different purposes. For instance, the parameter **inner_steps** used both as polynomial order in polynomial preconditioner and as number of max level in multilevel preconditioner. The parameters provided by **Precond_prm** do not cover all the parameters needed by Trilinos.

To solve this problem, a new submenu **TrilinosPrecond_prm** (mentioned in Section 4.2.4) is added to the **MenySystem**. This submenu has parameters shown in Table B.4. This proves to be quite effective, both for the end user and the programmer. Proper names and parameters are used to keep the application organized. And numbers of desired parameters are unlimited. This idea is also extendable for Trilinos solver and convergence monitor. Since there is no need for extra parameters in any of these. The preconditioner has the only extended submenu.

Every solver inherited from **IterativeSolver** of Diffpack has a declaration to a **SubdCommAdm** object. When using Trilinos preconditioner with a Diffpack solver, the Trilinos preconditioner has a relation to **TrilinosItSolver** (see Figure 4.9, left side). The **SubdCommAdm** object in **TrilinosItSolver** is
an empty pointer. Trilinos has no idea how the matrix and vectors are distributed among the processors. The \texttt{SubdCommAdm} object has to be assigned before Trilinos can apply preconditioning. This is avoided by introducing a global declaration of \texttt{SubdCommAdm}. Global means that every other classes can get the \texttt{SubdCommAdm} object. How to assign this global variable is shown in Section 4.4.

As mentioned in Section 3.4 Trilinos has to have “1-to-1 map”. So the “grid points mapping” from Diffpack cannot have any points represented more than once among the processors. When parallel programming, the relation between local and global mapping is very important. For Diffpack this is stored in class \texttt{GridPart}, this is introduced in Section 4.2.1. The temporary array \texttt{myGlobalPoints} contains no overlap. For some scenarios this can be an obstacle. To understand this better, a deeper understanding of Diffpack has to be explained.

Generally when partitioning a computational mesh, Diffpack splits along the edge of an element. This will result in no overlapping of elements, but the grid points along the splitting edge will be represented in more than one processor, Diffpack then adds a layer of element overlap. For a uniform partition of the mesh when number of overlap is one, Diffpack is able to determine all the interior boundary points. However, for an unstructured partitioning of the computational mesh Diffpack can not determine all the interior boundary points. Hence some points are defined as interior points in more than one processor, making the map not “1-to-1”.

To handle this obstacle when occurs the solution is provided by [20]. The basic idea is to divide the computational mesh by grid points rather than elements. The partition is illustrated in Figure 4.10. Details about the partitioning method can be found in [20, Ch.5].

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{diffpack_partition.png}
\caption{A 2D finite element method. Diffpack partition of an unstructured computational mesh. (a): Element-based partition: no overlapping elements, overlapping nodes. (b): Node-based partition: no overlapping nodes, overlapping elements.}
\end{figure}
4.3. Integration and Obstacles

4.3.1 Trilinos Files

One of the goals of this thesis is to make Trilinos options a part of Diffpack options. It should not be any different using the extended Trilinos in Diffpack compared to pure Diffpack. All the files needed to make this possible are:

- initTrilinos
- MyAztecOO
- MyEpetraCrsMatrix
- TrilinosItSolver
- TrilinosPrecond
- TrilinosMonitor
- TrilinosLinEqSolver_prm
- TrilinosPrecond_prm
- TrilinosConvMonitor_prm
- Files provided by others:
  - GridFEAdT, GridPartUnstruct, GridPart
  - Ifpack, Ifpack_Preconditioner, ml_MultiLevelPreconditioner

Files explained in earlier sections are not included here. These are files that start with Trilinos-.

initTrilinos has only one function. This function has to be called before starting the Diffpack simulator to make Trilinos options valid and acceptable by Diffpack MenuSystem. Has also a global instance of SubdCommAdm mentioned in Section 4.3.

MyAztecOO inherits from AztecOO of Trilinos. This class has only one function (applyPrec), used to apply Trilinos preconditioner on each iteration when the solver is an instance of Diffpack.

MyEpetraCrsMatrix inherits from Epetra_CrsMatrix of Trilinos, has only one function (addNewMatrix), used to add new matrix values. This function is useful if the matrix is changing for each iteration.

GridFEAdT, GridPartUnstruct and GridPart are provided by [20] to partition the computational mesh by grid points.

Ifpack, Ifpack_Preconditioner and ml_MultiLevelPreconditioner contain some modifications form the original Trilinos files. Files are edited to avoid namespace collision and parameter collision.
4.3.2 Behavior of Trilinos

When Trilinos classes are successfully integrated into Diffpack, simulators can accept options from both Diffpack and Trilinos with an input file. The use of an input file is introduced in Chapter A.1 and the extended options for “convergence monitor name”, “basic method” and “preconditioning type” are listed in Table B.1, B.2 and B.3. The possible combinations of Diffpack and Trilinos are shown in Table 4.1. The scenarios are as follows:

1. Pure Diffpack.

2. Trilinos convergence monitor will be altered to default Diffpack monitor which is CMRelResidual.

3. Diffpack solver with Trilinos preconditioner.

4. Diffpack preconditioner and convergence monitor will be set to default Trilinos, which are AZnone and AZr0.

5. Trilinos convergence monitor will be altered to Diffpack monitor CMRelResidual. The rest is similar to Scenario 3.

6. Diffpack preconditioner will have the same behaviour as a default Trilinos preconditioner, AZnone. The rest is like Scenario 4.

7. Diffpack convergence monitor will have the same behaviour as a default Trilinos monitor, AZr0.

8. Pure Trilinos.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Diffpack</th>
<th>Trilinos</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>5</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>6</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>7</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>8</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

*Table 4.1: Possible combinations of Diffpack and Trilinos.*

Note that Trilinos preconditioner options have to be set by submenu TrilinosPrecond_prm and Diffpack preconditioner options have to be set by submenu Precond_prm.

To summarize what Trilinos does: for scenario 4, 6, 7 and 8, Trilinos will do the following procedure when the solve function is called:
4.4 A Diffpack example

1. Create communicator (Epetra_Comm)
2. Create map (Epetra_Map)
3. Convert matrix (Epetra_CrsMatrix)
4. Convert vectors (Epetra_Vector)
5. Initialize solver (AztecOO)
6. Apply preconditioner if any (AztecOO, ML, Ifpack)
7. Apply type of solver
8. Iterate
9. Convert the solution vector back to Diffpack

Scenario 3 and 5: The map and matrix have to be assigned. For each iteration when a Diffpack solver applies a Trilinos preconditioner with applyPrec function:

1. Attach new vectors (converting vectors for from Diffpack to Trilinos)
2. Apply prec
3. Return the result vector (convert Trilinos vector back to Diffpack)

4.4 A Diffpack example

Code of initTrilinos.cpp is listed to gain a better understanding of this section.

```cpp
#include <diffpack.h>
#include <trilinos.h>

// Header files omitted

void initTrilinos(int argc, const char* argv[], bool hide_casename_files)
{
    // register Trilinos_prm as official prm of Diffpack:
    LinEqSolver_prm::registerPrmSubclass (*new TrilinosLinEqSolver_prm());
    Precond_prm::registerPrmSubclass (*new TrilinosPrecond_prm());
    ConvMonitor_prm::registerPrmSubclass (*new TrilinosConvMonitor_prm());
}

// global variable
Handle(SubdCommAdm) globalCommAdm;

void attachCommAdmForTrilinos (const SubdCommAdm &adm_) {
    globalCommAdm.rebind(adm_);
}
```

A standard sequential Diffpack simulator Poisson1 from [13] is used. The simulator has the purpose of solving a the boundary-value problem numerically by the finite element method. Parallelizing this simulator is detailed in [14, Ch.1]. Apart from including files from Section 4.3.1, a few lines have to be added in Poisson1 before the simulator is functional with the new extensions. Note that this could be applied to other Diffpack simulators as well, the procedure is the same. First file to edit is main.cpp. Include the necessary files:
#include "initTrilinos.h"
#include "TrilinosPrecond_prm.h"

Right after `initDiffack(argc, argv)`, Trilinos initialize function is called to make the Trilinos options valid.

    initTrilinos(argc, argv);

The `defineStatic` function from `TrilinosPrecond_Prm` has to be added before `global_menu.multiloop`. This will add a submenu of Trilinos preconditioner to the `MenuSystem` and becomes visible for the user:

    TrilinosPrecond_prm::defineStatic(global_menu, 1);

Next file to edit is `Poisson1.cpp`. First, include the `initTrilinos` file.

    #include "initTrilinos.h"

In the `scan` function right after the `GridPartAdm` object is assigned:

    #ifdef DP_PARALLEL_LA
      gp_adm->prepareCommunication(*dof);
      lineq->attachCommAdm(*gp_adm);
      attachCommAdmForTrilinos(*gp_adm);
    #endif

This assigns a global instance of `SubCommAdm` object. The `attachCommAdmForTrilinos` function is in `initTrilinos` file. This `SubCommAdm` object is needed by `TrilinosItSolver` in Scenario 3 and 5 from Table 4.1.

When these few lines are added, the simulator should work with Trilinos functions. Execute the application to see the available options, or just feed directly into the input file:

      sub Precond_prm
        set preconditioning type = AZls
        ok
        ...
      ...
      sub TrilinosPrecond_prm
        set poly ord = 2
        ok
Chapter 5

Benchmarking

This chapter will look into the performance of Diffpack, Trilinos and the combination of Diffpack and Trilinos. The cluster used in this chapter is the Linux cluster *chilopodus.simula.no*. Topics such as speedup, efficiency and overhead are discussed.

<table>
<thead>
<tr>
<th>chilopodus.simula.no</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Dual Socket Itanium2</td>
</tr>
<tr>
<td>Core Frequency</td>
<td>1300 MHz</td>
</tr>
<tr>
<td>Cache</td>
<td>16 KB L1, 256 KB L2 3 MB L3</td>
</tr>
<tr>
<td>System Memory</td>
<td>4 GB per CPU</td>
</tr>
<tr>
<td>Number of CPUs</td>
<td>24</td>
</tr>
<tr>
<td>Operating System</td>
<td>Debian GNU/Linux 3.1</td>
</tr>
<tr>
<td>Kernel</td>
<td>Linux 2.6.8 ia64 (SMP)</td>
</tr>
<tr>
<td>Queue system</td>
<td>Torque 1.2.0 (p0)</td>
</tr>
<tr>
<td>MPI Implementation</td>
<td>MPICH 1.2.6</td>
</tr>
<tr>
<td>Interconnect</td>
<td>Gigabit Ethernet</td>
</tr>
</tbody>
</table>

Table 5.1: Properties of *chilopodus.simula.no*.

When using Trilinos solver the Diffpack matrix and vectors have to converted to Trilinos matrix and vectors (Scenario 4, 6, 7 and 8 in Table 4.1). This applies to use of Trilinos preconditioner also (Scenario 3 and 5 in Table 4.1). For each iteration there are conversions of vectors. This process can be classified as overhead. It is interesting to see if Trilinos solver is faster than Diffpack solver, with matrix and vector conversion time included. The Poisson1 simulator introduced in Section 4.4 is used in this chapter. The simulator solves a boundary values problem numerically by using the finite element method to generate a linear system $Ax = b$. The iterative solvers compared in this chapter are the conjugate gradient and the generalized minimal residual.
5.1 The Conjugate Gradient Method

Table 5.2 is measurement of time taken from a solver starts until the solver converges. The problem is a unit cube domain with [39,39,39], [59,59,59] and [79,79,79] elements in each direction, and two 3D unstructured grids: heart-muscle.grid and fine refined-fixed.grid which contain 11306 total points and 28283 total points respectively. Table 5.2 is divided into 3 parts:

Part 1: Time of the conjugate gradient method in Diffpack, ConjGrad. The time is based on wall clock time of the solve function:

```c
double start, end;
start = proc_manager->getTime();
lineq->solve();  // solve linear system
end = proc_manager->getTime();
s_o << "\nTotal Solve Time: " << (end-start) << " secs\n";
```

Part 2: Similar to Part 1. Time is based on the solve function. The solver is the conjugate gradient method from Trilinos, AZcg. This time includes conversions of matrix and vectors.

Part 3: Time of pure Trilinos solve function, Iterate. Time used to convert matrix and vectors is not included.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Number of CPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>[39,39,39]</td>
<td>5.84375</td>
</tr>
<tr>
<td>[59,59,59]</td>
<td>33.4258</td>
</tr>
<tr>
<td>heart-muscle.grid</td>
<td>0.67578</td>
</tr>
<tr>
<td>fine_refined-fixed.grid</td>
<td>7.50000</td>
</tr>
<tr>
<td>[39,39,39]</td>
<td>4.12109</td>
</tr>
<tr>
<td>heart-muscle.grid</td>
<td>0.45312</td>
</tr>
<tr>
<td>fine_refined-fixed.grid</td>
<td>5.15234</td>
</tr>
<tr>
<td>[39,39,39]</td>
<td>3.80859</td>
</tr>
<tr>
<td>heart-muscle.grid</td>
<td>0.39453</td>
</tr>
<tr>
<td>fine_refined-fixed.grid</td>
<td>5.00000</td>
</tr>
</tbody>
</table>

Table 5.2: Wall clock time of Poinsson1 with conjugate gradient method in Diffpack and Trilinos, with no preconditioning.
5.1. The Conjugate Gradient Method

There is almost linear speedup for large problems. We will take a look at the measurement of $[79,79,79]$ grid from Table 5.2. The problem is a 3D unit cube domain and problem range is $[0,1] \times [0,1] \times [0,1]$ in x, y and z direction. Element type is ElmT4n3D; each element is a tetrahedron with 4 points in 3D.

```
set gridfile= P=PreproBox | d=3 [0,1]x[0,1]x[0,1] | d=3 e=ElmT4n3D [79,79,79] [1,1,1]
```

This results in a linear system with 80 points in each direction: $80^3 \rightarrow 512000$ unknowns. Solving time is illustrated in Figure 5.1.

![Figure 5.1: Time to solve a linear system with 512000 unknowns.](image)

Both solvers have decreasing solver time as the number of processors increases. In overall it looks like Trilinos has a faster conjugate gradient solver than Diffpack. Overhead time taken to convert Diffpack matrix to Trilinos matrix and vectors is also small using right below 2.5 seconds at most. Speedup and efficiency are illustrated in Figure 5.2 and 5.3.
Chapter 5. Benchmarking

Figure 5.2: Trilinos has almost perfect speedup, while Diffpack is just below.

Figure 5.3: Good speedup results in good efficiency, 1 is optimal.
5.2 The Generalized Minimal Residual Method

Table 5.3 is like Table 5.2, divided into 3 parts: pure Diffpack, Trilinos with conversion time and pure Trilinos. The problem is continued 3D unit cube domain with element $E_{1mT4n3D}$ split into $[59,59,59]$ elements.

<table>
<thead>
<tr>
<th>Part</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part 1</td>
<td>171.164</td>
<td>89.320</td>
<td>46.832</td>
<td>25.984</td>
<td>14.640</td>
</tr>
<tr>
<td>Part 2</td>
<td>81.160</td>
<td>42.062</td>
<td>21.839</td>
<td>11.516</td>
<td>6.305</td>
</tr>
<tr>
<td>Part 3</td>
<td>80.141</td>
<td>41.234</td>
<td>21.379</td>
<td>11.242</td>
<td>5.926</td>
</tr>
</tbody>
</table>

Table 5.3: Solver time of the generalized minimal residual in Diffpack and Trilinos, with no preconditioning.

The generalized minimal residual method is also faster in Trilinos than Diffpack. Overhead time for matrix and vector conversions is almost not noticeable, just like Table 5.2. Speedup and efficiency of the generalized minimal residual method are illustrated in Figure 5.5 and 5.6.
Chapter 5. Benchmarking

Figure 5.5: The generalized minimal residual method has a lower speedup than the conjugate gradient method.

Figure 5.6: Efficiency is also reduced compared to the CG method.
5.3 Trilinos Preconditioner

How effective is Diffpack solver with Trilinos preconditioner? The solver is once again conjugate gradient with a multilevel preconditioner. The options in the input file are as follows:

```plaintext
sub LinEqSolver_prm
  set basic method = ConjGrad
  set max iterations = 1000
ok

sub Precond_prm
  set preconditioning type = MLSA
ok

sub TrilinosPrecond_prm
  set ML output = 0
  set ML max level = 10
  set ML aggregation type = Uncoupled-MIS
  set ML smoother type = symmetric Gauss-Seidel
  set ML coarse type = Amesos-KLU
ok
```

Complete Trilinos options for the input file are listed in Appendix B.

<table>
<thead>
<tr>
<th>Description</th>
<th>Number of CPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Time saved</td>
<td>36%</td>
</tr>
</tbody>
</table>

**Table 5.4:** Diffpack conjugate gradient solver with Trilinos preconditioner. "Time saved" is based on time gained compared to Table 5.2.

Time saved using multilevel preconditioners compared to no preconditioning is 34-38%. Reason for such improvement lies within the number of iterations. Normally with no preconditioning, the conjugate gradient will use 359 iterations, but with multilevel preconditioner the numbers of iterations are considerably reduced to 17.

On the other hand, not all the preconditioners give improved results. Some preconditioners, such as polynomial preconditioners will increase the total solve time, but decrease the number of iterations.
5.4 Boussinesq Simulator

Boussinesq wave simulator from [13] is used to measure Trilinos performance on a time dependent problem. For each iteration, there are two linear systems that need to be solved, with matrix and vectors continuous changing. There are some interesting questions:

- How much faster is use of Trilinos preconditioner, compared to pure Diffpack? Since matrices and vectors are changing for each iteration and between the two linear systems, it is expected that time to convert matrix and vectors is more noticeable.

- What are the speedup and efficiency of Diffpack solver in combination of Trilinos preconditioner?

The input file of Boussinesq simulator has the following properties:

```
set gridfile = P=PreproBox | d=2 | [0,60]x[0,30] | d=2 e=ElmT3n2D [200,200] [1,1]
```

The problem is a unit square domain with triangle element in 2D, and 200 elements in each direction. Time step is 0.25 and goes from 0 to 20.

```
set time parameters = dt=0.25 [0,20]
```

Table 5.5 contains time measurement of Boussinesq wave simulator in five parts.

Part 1. Time of `timeLoop` function: Diffpack solver with no preconditioning.

Part 2. Time of `timeLoop` function: Diffpack solver with Trilinos preconditioner, MLSA from Section 5.3.

Part 3. Time of `init` and `applyPrec` function within a Diffpack solver when using Trilinos preconditioner. This is time to apply preconditioner, including conversion.

Part 4. Time for Trilinos to apply preconditioning. No conversion time included.

Part 5. Difference of Part 4. and Part 3; overhead time for conversions of matrices and vectors.

Since there are two linear systems in each time step result in: 2x matrix conversions and 6x vector conversions. In total of $2 \times 4 \times 20 = 160$ matrix conversions and $6 \times 4 \times 20 = 480$ vector conversions. The elements are 200 in $x$ and $y$ directions will result 40401 unknowns. Part 5. in Table 5.5 is overhead time for converting matrices and vectors to/from Trilinos. As the total time of `timeLoop` decreases (Part 2, Table 5.5), so does overhead time. Figure 5.7 illustrated the percentage of the solver time used to convert matrices and vectors. With multilevel preconditioner "MLSA" the number of iterations are reduced from 4-5 down to 1 for each time step. To summarize, speedup and efficiency of Diffpack solver with a Trilinos preconditioner based on `timeLoop` function are shown in Figure 5.8.
### 5.4. Boussinesq Simulator

<table>
<thead>
<tr>
<th>Number of CPUs</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part 1</td>
<td>147.965</td>
<td>77.8047</td>
<td>42.3906</td>
<td>25.4023</td>
<td>17.3789</td>
</tr>
<tr>
<td>Part 2</td>
<td>191.469</td>
<td>102.488</td>
<td>55.1797</td>
<td>34.6016</td>
<td>24.9883</td>
</tr>
<tr>
<td>Part 3</td>
<td>51.3086</td>
<td>27.5234</td>
<td>14.1712</td>
<td>9.83984</td>
<td>8.13672</td>
</tr>
<tr>
<td>Part 4</td>
<td>37.0230</td>
<td>20.6914</td>
<td>10.2695</td>
<td>7.36719</td>
<td>6.33203</td>
</tr>
<tr>
<td>Part 5</td>
<td>14.2856</td>
<td>6.83200</td>
<td>3.84770</td>
<td>2.47265</td>
<td>1.80469</td>
</tr>
</tbody>
</table>

**Table 5.5:** Solver time of Boussinesq wave simulator.

#### Figure 5.7: The percentage of solver time used to conversions; overhead time.

Calculated by dividing time in Part 5 by time in Part 2 in Table 5.5. Time to convert matrices and vectors is around 7% of the total solver time.
Chapter 5. Benchmarking

**Figure 5.8:** Speedup and efficiency of Diffpack solver with Trilinos preconditioner in time dependent PDE problem.
Chapter 6

Conclusions and Future Work

6.1 Conclusions

This chapter summarizes the thesis. The main questions posed in the introduction:

1. Is it possible to extend Diffpack in an elegant and flexible fashion?
2. Is there any profit integrating Trilinos into Diffpack?

**Question 1.** Throughout this thesis, we have proved that it is possible to extend Diffpack in an elegant and flexible fashion. Diffpack classes have a hierarchic structure, which makes it easy to comprehend. Most of the classes and functions are virtual. The implementations of Trilinos are added as new layers on top of Diffpack. None of the Diffpack classes are changed at any level. The “Prm” classes described in Section 4.2.4 provide the last elegant touch. Trilinos objects are built in the same way as Diffpack. For a user, it should not be any difference using Trilinos functions compared to pure Diffpack. By this we can conclude that Diffpack is easy extendable. Not just with Trilinos, but other packages of interest also.

**Question 2.** Several factors are involved when integrating Trilinos. First of all, we have extended Diffpack with a suite of new preconditioners not found in Diffpack before. And secondly, new iterative solvers are added. Even though all Trilinos solvers exist within Diffpack, some of the solvers have proved to be faster than Diffpack. Trilinos is under continuous development, whenever new releases are installed, Diffpack will be able to use the newest versions with their functionalities.
6.2 Limitations

Although Diffpack has been extended with Trilinos, there are several limitation factors that are worth mentioning:

- Class \texttt{DegFreeFE} in Diffpack takes care of the relation between degrees of freedom in a field representation and the ordering of the equations and unknowns in the linear system. If the numbering of computational mesh point does not match the numbering of rows in the linear system, mapping to Trilinos will fail.

- The main difference between a scalar and vector PDE is that vector PDE has more than one unknown per point. If the number of unknowns per point is $d$, will result in a linear system with $\text{gridpoints} \times d$ rows. Since the numbers of grid points do not match the number of rows in the linear system, this will also cause problem when mapping to Trilinos.

6.3 Future Work

Limitations from last section can be solved, but it is expected a lot of work and time for this. If this is accomplished in an elegant fashion, it will strengthen the extension of Diffpack with Trilinos considerably.

Diffpack solvers in Boussinesq simulator introduced in Section 5.4 can work with Trilinos preconditioners. If the solver is an instance of Trilinos the application will crash and displays error messages. It would be a good addition if the simulator can work with Trilinos solvers also. But on the other hand, all iterative solvers in Trilinos already exist in Diffpack, so there is no huge profit in such work.

The Trilinos version used in this project is 6.0.19. Since the beginning there has been new releases of Trilinos, the latest version is 7.0.8. Along with new releases, there are several new packages. It is interesting to look at the new packages, and explore the possibilities for integrating new packages of interest into Diffpack.

Trilinos convergence monitors implemented in Section 4.2.3 have a drawback; they work only with Trilinos solvers. By this we can conclude that the implementation is not fully completed. It should be possible for Diffpack to use Trilinos convergence monitor also. The implementation in Section 4.2.3 can be used as framework for future development.
Appendix A

Diffpack Supplementary

A.1 Diffpack Input File

Diffpack simulators use an input file to feed desirable parameters from the user. This provides a flexible and easy-to-use interface. This chapter gives an introduction to an input file. No assumption of earlier knowledge is required. But on the other hand this is not a full introduction to Diffpack input files, just an example. It should point out the basic ideas.

```text
! problem domain, type of element
set gridfile = P=PreproBox | d=3 [0,1]x[0,1]x[0,1] | d=3 e=ElmT4n3D [19,19,19] [1,1,1]
sub GridPart_prm !submenu gridpart
    set grid source type = UniformPartition !partition type
    set subdomain division = d=3 [0,1]x[0,1]x[0,1] in:[0:4]x[0:1]x[0:1]
    set use triangle elements = ON
ok
sub LinEqAdmFE !submenu for linear system/solver
    sub LinEqSolver_prm !submenu for linear solver
        set basic method = ConjGrad !iterative method
        set max iterations = 1000 !max number of iterations
        exit LinEqSolver_prm
    exit MatSparse !submenu for matrix storage
ok
sub Precond_prm !submenu for preconitioner
    set preconditioning type = PrecJacobi !preconditioner type
    set RILU relaxation parameter = 0.0 !preconditioner parameter
    set (S)SOR relaxation parameter = 1.0 !number of inner steps
    exit Precond_prm
ok
sub DefineConvMonitor #1 !submenu for convergence monitor
    sub ConvMonitorList_prm
        define ConvMonitor #1
        set #1: convergence tolerance = 1.0e-8
        set #1: convergence monitor name = CMAbsResidual
        exit ConvMonitor #1
        exit ConvMonitorList_prm
ok
ok
ok
ok
```

End of code
From the input file: set gridfile contains information about the computational mesh. In this case the problem is in 3D and problem range is 
\([0,1] \times [0,1] \times [0,1]\) in x,y and z direction. Element type is ElmT4n3D: each element is a tetrahedron with 4 points in 3D. set grid source type determines how the grid in the problem domain is divided, since the problem is a unit cube, UniformPartition is chosen. set sub domain division tells how to partition the uniform grid. In this example, the x direction is split into four parts. When running this input file it has to be executed on four nodes in order to fulfil the partition criteria.

For many problems a unit square or cube is not sufficient. For instance computation of an unstructured grid heart-muscle-grid, this can not be divided uniformly. The solution to is to use GlobalGrid to the parameter set grid source type.

By default set basic method = GaussElim which is a direct solver and not suited for parallel computing. For iterative parallel solver conjugate gradient (ConjGrad) is to be preferred for a symmetric positive definite matrix. More info about iterative solvers in Diffpack see [13, Appendix D.3].

Precond_Prm contains info about the preconditioner. By default no preconditioner is applied. Diffpack preconditioners are described in [13, Appendix D.5]. ConvMonitor_Prm contains info about the convergence monitor. Diffpack stopping criteria and convergence monitors are described in [13, Appendix D.6].

The rest of input.i is straightforward. The name of the variables tells much about what they are for and their properties.
When creating a Diffpack folder with `Mkdir` command, all needed make files are created also. However, these make files are only compatible with Diffpack. To include Trilinos packages, edit `.cmake2` file. Installation of Trilinos can be found in [10, Ch.1].

```bash
# This .cmake2 file contains application specific customization of
# the general Makefile. Additional customization is found in .cmake1

# Name of the executable file:
APPL := app

# Modifications of .cmake1/.cmake2 are intended to be performed by
# advanced users. Some make variables (e.g. NUMT) must be set in .cmake1.
# First .cmake1 is included, then MakeHeaders, then MakeFlags and then
# .cmake2.
# The directory of local copy of Trilinos
TRILINOS_HOME = /home/.../trilinos-6.0.19

# The folder with compiled Trilinos
TRILINOS_ARCH = LINUX_MPI
TRILINOS_LIB = $(TRILINOS_HOME)/$(TRILINOS_ARCH)

# Compile and link with some external software packages:
INCLUDEDIRS += -I$(TRILINOS_LIB)/include/
LDPATH += -L -L$(TRILINOS_LIB)/lib -L/usr/lib/mpich/lib
TMP := -lteuchos -lepetra_test -lml -laztecoo -lifpack -lteuchos \
        -lepetra -lepetra -lameos -leptraext \
        -lepetra -lteuchos -lg2c -lm -lunwind -lmpi -lmpich -lmpichfsup \
        -lmpich -lpmpich -lpmpich -lpmf -lpmc -lpmc -lpmf -lpmc
LIBS := $(TMP) $(LIBS)

# link with additional system libraries:
SYSLIBS += -ljpeg
```

Appendix B

Extended Trilinos Options

As mentioned in Appendix A.1 Diffpack simulator uses a file as input. The goal of this thesis is to make Trilinos options a part of Diffpack.

When extending Diffpack, the main parameters to change in the input file are:

- set basic method
- set preconditioning type
- set #1: convergence monitor name

Besides accepting the standard Diffpack options, new extended options for convergence monitor name, basic method and preconditioning type are listed in Table B.1, B.2 and B.3 respectively.

<table>
<thead>
<tr>
<th>Command</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZr0</td>
<td>$| r^k | / | r^\text{ref} |$(default)</td>
</tr>
<tr>
<td>AZrhs</td>
<td>$| r^k | / | b |$</td>
</tr>
<tr>
<td>AZAnorm</td>
<td>$| r^k | / | A |_{\infty}$</td>
</tr>
<tr>
<td>AZnoscaled</td>
<td>$| r^k |$</td>
</tr>
<tr>
<td>AZsol</td>
<td>$| x^k |<em>{\infty} / (| A |</em>{\infty} + | x^k |<em>{1} + | b |</em>{\infty})$</td>
</tr>
</tbody>
</table>

Table B.1: Extended valid options of set #1: convergence monitor name.

As mentioned in Section 4.3, the TrilinosPrecond_prm class has a submenu, the parameters are shown in Table B.4.
<table>
<thead>
<tr>
<th>Command</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZcg</td>
<td>Conjugate gradient.</td>
</tr>
<tr>
<td>AZcgcn</td>
<td>Conjugate gradient with condition number estimation. Similar to AZcg. Additionally computes extreme eigenvalue estimates using the generated Lanczos matrix.</td>
</tr>
<tr>
<td>AZgmres</td>
<td>Restarted generalized minimal residual.</td>
</tr>
<tr>
<td>AZgmrescn</td>
<td>Restarted GMRES with condition number. Similar to AZgmres. Additionally computes extreme eigenvalue estimates using the generated Hessenberg matrix.</td>
</tr>
<tr>
<td>AZcgs</td>
<td>Conjugate gradient squared.</td>
</tr>
<tr>
<td>AZtfqmr</td>
<td>Transpose-free quasi-minimal residual.</td>
</tr>
<tr>
<td>AZbicgstab</td>
<td>Bi-conjugate gradient with stabilization.</td>
</tr>
</tbody>
</table>

Table B.2: Extended valid options of set basic method.
<table>
<thead>
<tr>
<th>Command</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZNone</td>
<td>No preconditioning. (default)</td>
</tr>
<tr>
<td>AZJac</td>
<td>k step Jacobi. Default step: 1. No. step can be changed with set poly ord.</td>
</tr>
<tr>
<td>AZNeu</td>
<td>Polynomial Neumann series. Default polynomial order: 1. Polynomial order can be changed with set poly ord.</td>
</tr>
<tr>
<td>AZls</td>
<td>Polynomial Least-squares series. Default polynomial order: 1. Polynomial order can be changed with set poly ord.</td>
</tr>
<tr>
<td>AZSGS</td>
<td>Non-overlapping domain decomposition k step symmetric Gauss-Seidel. Default step: 1. No. step can be changed with set poly ord.</td>
</tr>
<tr>
<td>AZILU</td>
<td>Domain decomposition preconditioner (additive Schwarz). Default overlapping order: 0, can be changed by set overlap. (Applies to AZILU, AZBILU, AZBILUT and AZRILU also). Using ilu(k) where k can be changed by set fill level.</td>
</tr>
<tr>
<td>AZILUT</td>
<td>The drop tolerance given by set omega and fill-in is given by set fill level.</td>
</tr>
<tr>
<td>AZRILU</td>
<td>Using rilu(k,ω), where ω can be changed by set omega, and k by set fill level.</td>
</tr>
<tr>
<td>AZBILU</td>
<td>Using block ilu(k). k can be changed by set fill level.</td>
</tr>
<tr>
<td>AZICC</td>
<td>Incomplete Cholesky, icc(k). k can be changed by set fill level.</td>
</tr>
<tr>
<td>IfpackPR</td>
<td>Point (damped) relaxation preconditioner set by set relax type for Jacobi(default), Gauss-Seidel, or symmetric Gauss-Seidel. Damping factor is set by set relax damping factor. User can specify the number of Jacobi steps (sweeps) by set relax sweep. The number of overlap is provided by set partition overlap (applies to all IFPACK preconditioner).</td>
</tr>
<tr>
<td>IfpackBR</td>
<td>Similar to IfpackPR. Block relaxation preconditioner.</td>
</tr>
<tr>
<td>IfpackIC</td>
<td>Incomplete Cholesky factorization with dropping based on the level-of-fill set by set fill level.</td>
</tr>
<tr>
<td>IfpackICT</td>
<td>Incomplete Cholesky factorization, with dropping based on threshold set by set threshold.</td>
</tr>
<tr>
<td>IfpackILU</td>
<td>Incomplete LU factorization with dropping based on the level-of-fill set by set fill level.</td>
</tr>
<tr>
<td>IfpackILUT</td>
<td>Incomplete LU factorization with dropping based on threshold set by set threshold.</td>
</tr>
<tr>
<td>IfpackAmesos</td>
<td>Complete LU factorization on each sub domain. Solver type set by set amesos solver type.</td>
</tr>
<tr>
<td>MLDD</td>
<td>2-level domain decomposition preconditioners based on aggregation. These parameters applies to all ML preconditioner: set ML max level, set ML aggregation type and set smoother and set ML coarse type.</td>
</tr>
<tr>
<td>MLDDML</td>
<td>3-level domain decomposition preconditioners, with coarser spaces defined by aggregation.</td>
</tr>
<tr>
<td>MLSA</td>
<td>Classical smoothed aggregation preconditioners.</td>
</tr>
<tr>
<td>MLMaxwell</td>
<td>Edge-element AMG for Maxwell equations.</td>
</tr>
</tbody>
</table>

Table B.3: Extended valid options of set preconditioning type.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>poly ord</td>
<td>0</td>
<td>Number of steps for Jacobi and polynomial order for polynomial preconditioners. Used in AztecOO</td>
</tr>
<tr>
<td>overlap</td>
<td>0</td>
<td>Number of overlap for AztecOO DD preconditioner</td>
</tr>
<tr>
<td>fill level</td>
<td>0</td>
<td>The level of fill-in for incomplete factorizations. Used in AztecOO and IFPACK.</td>
</tr>
<tr>
<td>omega</td>
<td>1.0</td>
<td>Damping or relaxation parameter used for incomplete factorizations. Used in AztecOO and IFPACK.</td>
</tr>
<tr>
<td>threshold</td>
<td>1.0</td>
<td>Incomplete factorization with dropping based on threshold. Used in AztecOO and IFPACK.</td>
</tr>
<tr>
<td>relax type</td>
<td>Jacobi</td>
<td>Specifies the type of point and block relaxation scheme. Used in IFPACK. Valid options: Jacobi/Gauss-Seidel/symmetric Gauss-Seidel</td>
</tr>
<tr>
<td>relax sweep</td>
<td>1</td>
<td>Specifies the number of sweeps in the application of point relaxation schemes. Used in IFPACK.</td>
</tr>
<tr>
<td>relax damping factor</td>
<td>1.0</td>
<td>Specifies the damping factor in the application of point relaxation schemes. Used in IFPACK.</td>
</tr>
<tr>
<td>partition overlap</td>
<td>0</td>
<td>Specifies the overlap among the blocks which can differ from the overlap among the processors. Note that only the Jacobi block relaxation scheme can take advantage of non-zero overlaps. Used in IFPACK.</td>
</tr>
<tr>
<td>amesos solver type</td>
<td>Amesos_Klu</td>
<td>Defines the Amesos solver to be used by class IfpackAmesos. Used in IFPACK. Valid options: Amesos_Lapack /Amesos_Klu/ Amesos_Umpack/ Amesos_Superlu/ Amesos_Kumps/ Amesos_Dscpack</td>
</tr>
<tr>
<td>ML output</td>
<td>5</td>
<td>ML output level, 0 being silent and 10 very verbose</td>
</tr>
<tr>
<td>ML max level</td>
<td>10</td>
<td>Maximum number of levels. Used in ML.</td>
</tr>
<tr>
<td>ML aggregation type</td>
<td>Uncoupled</td>
<td>Defines the aggregation scheme. Used in ML. Valid options: Uncoupled/ MIS/ Uncoupled-MIS/ METIS/ ParMETIS</td>
</tr>
<tr>
<td>ML smoother type</td>
<td>Jacobi</td>
<td>Defines the smoother type. Used in ML. Valid options: Jacobi/ Gauss-Seidel/ symmetric Gauss-Seidel/ Aztec/ MLS</td>
</tr>
<tr>
<td>ML coarse type</td>
<td>Amesos-KLU</td>
<td>Defines the coarse solver. Used in ML. Valid options: Jacobi/ Gauss-Seidel/ MLS/ Hiptmair/ Amesos-KLU/ Amesos-Superlu/ Amesos-UMFPACK/ Amesos-Superludist/ Amesos-MUMPS</td>
</tr>
</tbody>
</table>

Table B.4: Parameters of TrilinosPrecond_prm.
Bibliography


