UNIVERSITY OF OSLO
Department of Informatics

Extending Diffpack with PETSc solvers and Preconditioners

Mads Fredrik Skoge Hoel

May 2, 2008

Extending Diffpack with PETSc solvers and Preconditioners

Mads Fredrik Skoge Hoel

May 2, 2008

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Parallel computing refers to solving a problem concurrently on multiple processors. This is motivated by increasing performance of a single computer. The challenge with parallel computing is that it is more complicated to program, test and debug in comparison with a sequential program. With parallel computing becoming popular and available to the public, the demand for high level tools to simplify program development exists and is increasing.

Diffpack and PETSc are two high level libraries that simplify development of parallel numerical software, in particular software for solving partial differential equations. The two libraries differ in features. Diffpack has tools for automating the whole process of setting up the linear system, solving and report generation. PETSc has more emphasis on solving problems and have a lot of linear solvers and preconditioners not available in Diffpack. By extending Diffpack with PETSc linear solvers and preconditioners we get the best of both worlds, the flexibility of Diffpack combined with the solvers and preconditioners offered by PETSc.

In this master thesis we will look at how to extend Diffpack with parallel linear algebra solvers (Krylov Subspace Methods) and preconditioners from PETSc.

What we hope to answer the following questions:

1. Can Diffpack be extended with PETSc Krylov Subspace Solvers and Preconditioners in such a way that no modification of the Diffpack library is required?

2. Does application of PETSc solvers and preconditioners require modification of an existing Diffpack Simulator?

3. Can the Krylov Subspace Solvers and Preconditioners of PETSc outperform those already existing in Diffpack?

1.1 Acknowledgments

Big thanks to Prof. Xing Cai for great counseling, support and particularly for allowing me the opportunity of writing a master thesis at Simula Research Laboratory. Thank you Simula staff and social committee for providing us

students, me included, with a great environment and making us a part of the arrangements. I thank my fellow Simula students Anders Knatten, Guo Wei Ma, Magnus Vikstrm, Martin Tingstad, Kim Kalland and Jacob Libak for good company both socially and professionally.

1.2 Organization of the thesis

This thesis starts off in Chapter 2 with giving a brief introduction and overview of features and concepts of parallel computing that will aid us in extending Diffpack with PETSc.

In Chapter 3 we introduce new classes, discuss and define methods of converting matrices and vectors needed to use the PETSc solvers and preconditioners.

In Chapter 4 we look at the performance of the linear solvers and preconditioners introduced along with the overhead of converting the matrices and vectors.

Finally in Chapter 5 we summarize the work and answer the questions given in the introduction based on the work done in Chapter 3 and the results we got in Chapter 4.

Chapter 2

Background

This chapter present technical details needed in this thesis.

2.1 Partial differential equations

Partial differential equation is an equation where we want to find a function describing/modeling an underlying physical problem. These kinds of problems can be challenging, or even unsolvable by human means.

The numerical approach to solving the partial differential equation is to solve an approximate problem, which ultimately leads to a linear system that can be solved by a computer. This way, the original problem is simplified, with a likely loss of quality in the solution. This way we are able to solve equations with a higher order of dimensions, more parameters on more complicated domains, with the bottleneck being the computers currently available.

2.1.1 Finite element method

The finite element method is good for solving PDEs having complicated domains. But has several other benefits for our purpose in this thesis. There are some considerations that must be done when running a parallel program using the finite element method. We will in this section give an overview on the finite element method. This overview is not meant to be a replacement to what can be found in a book on the subject, like [9], but instead serve as aid in understanding what must be done when we integrate PETSc solvers into a parallel program using Diffpacks finite element framework.

The finite element method is a method of turning a partial differential equation into an approximate problem that can be solved by a computer. A problem such as the Poisson equation is first transformed into its weak form. This weak form consist of a sum of basis functions and coefficients. By finding the coefficients in this weak form, we get an approximate solution to the original problem.

The weak form ultimately gets transformed into a linear system that can be solved by a computer. The type of element affects the pattern of the matrix ,

The basis functions in the finite element method could in reality be any kind of function as long as it fulfills certain requirements; it must be 1 in one node and zero in all other nodes; it must be nonzero over the elements to which this

node belongs. This makes the product of the basis functions vanish except over a local patch of elements. This effect in turn affects the entries of the matrix in the corresponding linear system (Figure 2.1). This is one of the benefits to the finite element method; the matrices from the finite element method become sparse, very suitable for sparse matrices which in turn is highly suitable for iterative solvers. That can utilize the number of non zeros to get excellent performance.

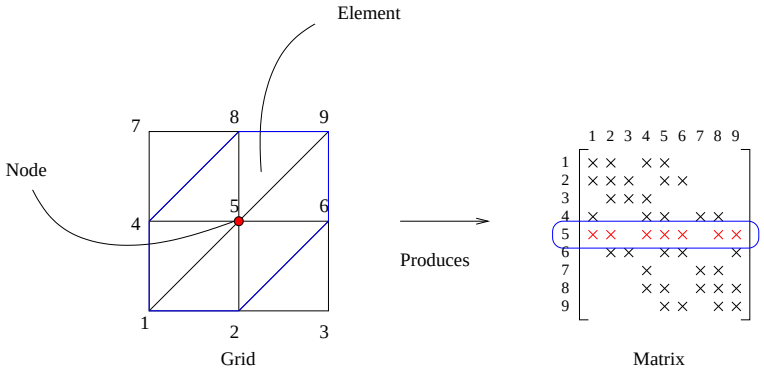

Figure 2.1: Elements affect the matrix pattern. The blue line defines the elements within the local patch of elements where the product of basis function belonging to node 5 is nonzero. A row in the matrix corresponds to a single node in the grid. For example, node 5 corresponds to row 5.

2.1.2 MPI

In parallel computing, as with other parallel programming, there is a need to communicate between processors in a cluster. A standardized approach of doing this is known as the Message Passing Interface (MPI).

While working with higher level libraries such as Diffpack and PETSc we need not work directly with MPI.

2.2 Krylov Subspace Methods

Unlike direct solvers like gauss elimination, iterative solvers like the Krylov Subspace Solvers are quite suitable for parallel computing[6]. With little communication, when using the finite element method, these methods are suitable for parallel applications.

Krylov subspace methods consist of three types of linear algebra operations:

- Matrix-vector multiplication: $\mathbf{w} = \mathbf{A} \cdot \mathbf{u}$
- Vector addition: $\mathbf{w} = \mathbf{u} + \alpha \mathbf{v}$
- Inner product: $c = \mathbf{u} \cdot \mathbf{v}$

These operations require little to no communication and can be performed in parallel[6].

2.2.1 Convergence criteria

Krylov subspace methods, as with other iterative methods, comes up with an approximate solution after a series of iterations. The key to getting good results is to know when to stop the iterations. And without knowing the analytical solution in advance, this is likely to be challenging. For this purpose libraries such as Diffpack and PETSc come with predefined convergence tests.

2.3 Partitioning

For good parallel performance the work load must be distributed as evenly as possible among processors. Data is commonly divided into partitions at grid level, preferably before the linear system is constructed as it is often based on the grid.

There exists many different partitioning algorithms today; some are fast and produce poor quality in partitions, others are slow but produce higher quality partitions. A fine balance between the two is often sought.

Diffpack supports two different partitioning strategies, structured partitioning and unstructured partitioning (ParMETIS)[6]. The unstructured partitioning method uses ParMETIS which is preferred to the structured partitioning method since it is fully automatic, and provide good work load balancing on unstructured finite element grids.

For a full introduction and overview of partitioning algorithms and software see [11].

2.4 Diffpack

Diffpack[2] is numerical library with an emphasis on solving partial differential equations[9].

Diffpack allows the scientist to focus on solving the PDE and less on the programming problem. The scientist is able to experiment with different kinds of grids, elements, linear solvers, preconditioners etc. without having to edit and recompile the code. This is possible by using a Diffpack simulators. By using a Diffpack simulator the only things the scientist need to provide is problem specific information. Some predefined simulators come with the Diffpack software, like the Poisson1 simulator, which we will be using in this thesis. The Poisson1 simulator is a standard Diffpack simulator, and has been used to introduce Diffpack simulators in [9, ch. 3].

Diffpack has also several benefits for a programmer. Diffpack has predefined classes to simplify the task of writing numerical applications, including new Diffpack simulators. These classes can take care of tasks such as grid generation, assembly of elements, solving the linear system, error handling, report generation.

2.4.1 Diffpack simulator

The Diffpack simulator is a user defined class inheriting from SimCase or one of the SimCase base classes. The user provides problem specific informations through this class, letting the Diffpack library do the rest. Running the program

the user is prompted with a menu to allow the user to experiment on how the problem is solved.

Diffpack differentiates between two different Simulator classes, the finite difference simulator (FDM) and the finite element simulator (FEM). These behave mostly in the same way, but the problem specific definitions are of course different.

2.4.2 Parallel Diffpack

Diffpack was originally a uniprocessor library. Some years later the Parallel Toolbox [1] was created, to provide Diffpack with support for solving PDEs on a parallel computer. For the moment, the parallel toolbox mainly supports the finite element simulator.

Thanks to the object oriented design of Diffpack, parallel support was added in an elegant way, with close to no modification to the original library [6]. The way the Parallel Toolbox was designed allowed the user only to add only a few lines to make their existing finite element simulator parallel.

2.5 PETSc

PETSc (Portable, Extensible Toolkit for Scientific Computation), features a lot of solvers and preconditioners, with the intent of solving partial differential equations. Though PETSc is written in the relatively low level language C, like Diffpack it is designed at a high level to be easy to use, tweak and experiment with. Some of these solvers work with both serial and parallel applications. The Krylov subspace solvers of PETSc are among these.

Another attractive feature of PETSc is the interfaces to third party libraries. PETSc has a lot of interfaces to third party libraries and tools. One of the libraries, Hypre[**?**], is certainly of interest to us as it extends PETSc with high performance preconditioners.

The Krylov solvers of PETSc provide support for both serial and parallel applications.

2.6 PETSc Solvers

Though PETSc is lacking features that make Diffpack , it makes up for in variety of solvers. PETSc separates between 3 classes of solvers; nonlinear solvers, time steppers and Krylov subspace methods.

It would be interesting to investigate the nonlinear solver and time steppers and see how them can extend Diffpack. But due to the size of the PETSc library, we will be limiting this thesis we will consider the Krylov subspace solvers.

2.6.1 Krylov subspace methods

PETSc has a lot of Krylov type solvers, some are already a part of Diffpack. Though a lot of them are variants of some Krylov method, it is easy to see that Diffpack can be further enriched. The most interesting part will be on the efficiency of the methods with the extra overhead of a Diffpack-PETSc interface.

Method	*PETSc*	*Diffpack*
BiCGStab	X	X
BiCGStabL	X	-
Chebychev	X	-
ConjGrad	X	X
CGS	X	X
GMRES	X	X
LGMRES	X	-
LSQR	X	-
MINRES	X	X
Orthomin	-	X
Richardson	X	-
RTCQMR	X	X
ConjRes	X	-
SymMinRes	X	X
Symmlq	-	X
RTCQMR	X	-
TFQMR	X	X

2.7 Preconditioners of PETSc

In this section we list the preconditioners that are accessible from PETSc. PETSc has a lot of preconditioners, some are not true parallel preconditioners and some need some external package, and some are not supported by the `MATMPIAIJ` format.

The listing presented below is based on[3]. Question marks in the table indicates a feature that is currently not described in the documentation.

Preconditioners of PETSc			
Method	*Ext.package*	*Parallel*	*MPIAIJ*
PCASM	-	X	X
PCBJACOBI	-	X	-
PCCHOLESKY	-	X	-
PCEISENSTAT	-	X	X
PCGALERKIN	-	X	X
PCHYPRE	X	X	X
PCICC	-	-	-
PCILU	-	-	-
PCJACOBI	-	X	X
PCKSP	-	X	X
PCLU	-	-	-
PCMAT	-	?	?
PCMG	-	X	X
PCML	-	X	X
PCNN	-	X	-
PCOPENMP	-	?	?
PCPROMETHEUS	X	X	-
PCREDUNDANT	-	X	X
PCSAMG	X	?	?
PCSOR	-	X	X
PCSPAI	X	X	X

Chapter 3

Extending Diffpack with PETSc

In this chapter we will see how we can take advantage of Diffpack object oriented design to extend Diffpack, in an elegant way with PETSc solvers and preconditioners.

Extending Diffpack with PETSc allows us to increase the number of solvers and preconditioners currently available in Diffpack. It is also attractive for a Diffpack user to have a unified interface to both libraries, making it easier to experiment with different kinds of solvers and preconditioners without having to create a separate simulator for each library.

PETSc has a lot more to offer than just Krylov solvers and preconditioners, but because of the size of the PETSc library, we are going to only add support for the Krylov subspace methods and the preconditioners. But the ideas and principles applied here, can be transferred to further extending Diffpack with more features from PETSc.

3.1 Using Diffpack with PETSc

There are a few things to consider when using Diffpack with PETSc. On the lower level we must consider how to convert matrices and vectors. On the more higher level we must create classes that represent these solvers as well as figure out where to place the PETSc solvers and preconditioners.

3.2 The Poisson1 Simulator

The `Poisson1` simulator is one of many simulators accompanying the Diffpack documentation, many of which have been described in the book [9]. They are intended to work as examples on Diffpack usage. But because of their flexibility, and object oriented design, they can be easily modified and extended to serve as a basis to solve new problems.

The `Poisson1` simulator is a finite element simulator solving the Poisson problem. It makes use of the highest level of classes in Diffpack, the `LinEqAdmFE` and the `MenuSystem`, among others.

Though `Poisson1` is a sequential simulator it can be easily be made into a parallel simulator by using the parallel toolbox, this has been done by X. Cai in [6].

3.3 New classes

We want to be able to use PETSc solvers and preconditioners in the same fashion as with Diffpacks solvers and preconditioners. To experiment with different kinds of Diffpack solvers and preconditioners the user does not need to edit and recompile the code. They are able to be easily changed between different solvers and preconditioners at runtime via the `MenuSystem`.

When using Diffpack in this fashion, a preconditioner and solver of a particular type are never declared in the code. An object of a base class is used instead. For preconditioners this is `Precond`, and `LinEqSolver` for solvers.

When the application is started, Diffpack reads the input from the standard input, and with the aid of a parameter class creates the correct type of solver and preconditioner based on the options specified.

3.3.1 New class hierarchies

To make the PETSc solvers and preconditioners compatible with Diffpack classes, in particular the high level interfaces/classes such as `LinEqAdmFE`, we need to extend one of the classes in the `LinEqSolver` and the `Precond` hierarchy.

class PetscSolver A natural base class to extend with the PETSc Krylov Subspace methods is the `KrylovItSolver`, which defines the interface to Diffpacks existing Krylov Subspace solvers. All the `KSP` solvers of PETSc are added as sub classes of PetscSolver.

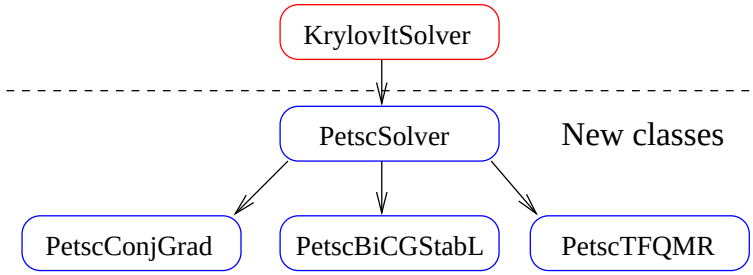

Figure 3.1: Where PETSc solvers are added to Diffpack hierarchy

class PetscPrecond `Precond` is the base class for the preconditioners of Diffpack. Here we introduce class `PetscPrecond` extending `Precond`, to serve as a base class for the PETSc preconditioners.

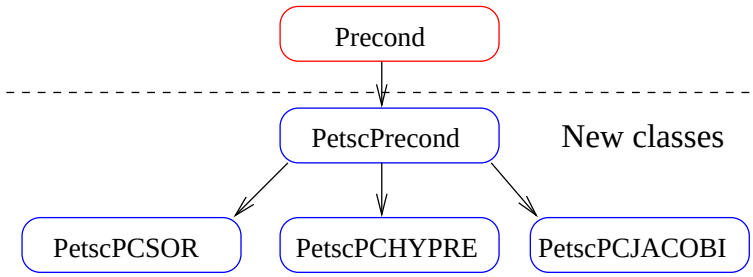

Figure 3.2: Where PETSc preconditioners are added to Diffpack hierarchy

3.3.2 Extending class hierarchy of Diffpack

In [8] H. P. Langtangen explains how to extend a class hierarchy in Diffpack, without recompiling the libraries. To be able to do so, we need to create a parameter (`prm`) subclass for the base class in the hierarchy we want to add. We have two hierarchies we want to add, `PetscSolver` and `PetscPrecond`. For this purpose we create `PetscSolver_prm` for the PETSc solvers and `PetscPrecond_prm` for the PETSc preconditioners.

The parameter classes for `Precond` preconditioners and `LinEqSolver` solvers are `Precond_prm` and `LinEqSolver_prm`. For this purpose we introduce new prm classes `PetscSolver_prm` and `PetscPrecond_prm` for our newly created `PetscSolver` and `PetscPrecond`, respectively.

Each of these new `prm` classes will take care of creating the correct PETSc solver and preconditioner based on runtime information.

3.3.3 PETSc in object oriented programming

The object oriented design of PETSc, for the most part, fits nicely into object oriented programming. There is one subtle, yet important issue of PETSc that needs to be taken care of.

PETSc needs to be initialized and finalized in the correct order, and PETSc objects needs to be destroyed before PETSc is finalized. If PETSc is initialized after MPI and parallel Diffpack, it needs to be finalized before MPI and parallel Diffpack. This can be remedied by encapsulating the PETSc finalize call in a `finally` clause. This however does not solve the issue with destruction of PETSc objects when used in destructors, because these objects may be destroyed after main is exited.

One solution is to create a class which takes care of initializing and finalizing the libraries, then create a global instance of this object.

For this purpose we introduce a new class `SubSystemManager`, a name and idea borrowed from the Dolfin[7] project.

3.4 Dealing with convergence monitors

In the most interesting problems, we do not know the analytical solution, so the more commonly used stopping criteria is based on the absolute and relative residual[9, p. 815]. Diffpack gives us the ability to choose among a wide range

of convergence tests. PETSc on the other hand, has a default convergence test, that is a mixture of two convergence tests; absolute and relative residual convergence test [4, p. 71].

$$||r_k||_2 \leq max(rtol * ||b||_2, atol) \tag{3.1}$$

where $r_k = b - Ax_k$

To truly incorporate PETSc solvers into Diffpack, instead of adding support for this mixed convergence test, we will add support for Diffpacks most common convergence monitors the `CMRelResidual` and `CMAbsResidual`. This way we avoid adding a convergence monitor that does not work with Diffpack, and we make the PETSc solver more compatible with Diffpack.

The way to do this is to check using a `dynamic_cast` inside the `PetscSolver` class if the convergence monitor is of type `CMRelResidual` and if it is not, we will assume that it is of type `CMAbsResidual`.

If a `CMAbsResidual` is used we set the relative tolerance to zero. And if it is a `CMRelResidual` we set the absolute tolerance to zero and use these functions to make PETSc use `KSPDefaultConvergedSetUIRNorm` and `KSPSetNormType(ksp, KSP_NORM_UNPRECONDITIONED)`. `KSPDefaultConvergedSetUIRNorm` sets the default norm to be $||B*(b-A*(initialguess))||$ and `KSPDefaultConvergedSetUIRNorm` tells PETSc to use an unpreconditioned norm instead.

3.5 Converting matrices and vectors

To be able to call the solvers and preconditioners of PETSc we need to convert the matrices and vectors that make up the linear system. But in order to successfully convert the matrices and vectors we must first deal with two issues; overlapping boundary nodes/ghost rows and renumbering the nodes.

Diffpack introduces some overlapping boundary nodes when setting up and assembling the element matrices. In PETSc however these nodes can only be represented in one process. So we must remove/drop all relation to these nodes when constructing the linear system to PETSc.

PETSc uses a different node numbering as in Diffpack. There is currently no way, at least not documented, to tell PETSc to use a different mapping. So we must renumber the nodes to prevent PETSc from repartitioning the nodes.

3.5.1 Removing overlapping boundary nodes

When Diffpack has partitioned a grid among several processes there are nodes that exist in several processes. These internal boundary nodes are necessary when using the finite element method to get contributions from elements that lie on the boundaries between two or more processes.

These boundary nodes are represented in at least two processes, however in PETSc the nodes can only belong to one process. So we have to remove these internal boundary nodes.

Removing the boundary nodes can be done once we have set the level of overlapping elements to one. In this scenario the internal boundary nodes on one process ends up in the grid belonging to its neighbor process. This way, we do not lose any elements, and at the same time, we get internal boundary nodes that can be removed, without loosing any elements (Figure 3.3).

Getting an exact overlap of one element, however, is not ensured when using the element based partitioning of Diffpack. But can be ensured by using Tingstad's version of class `GridPartUnstruct` [12].

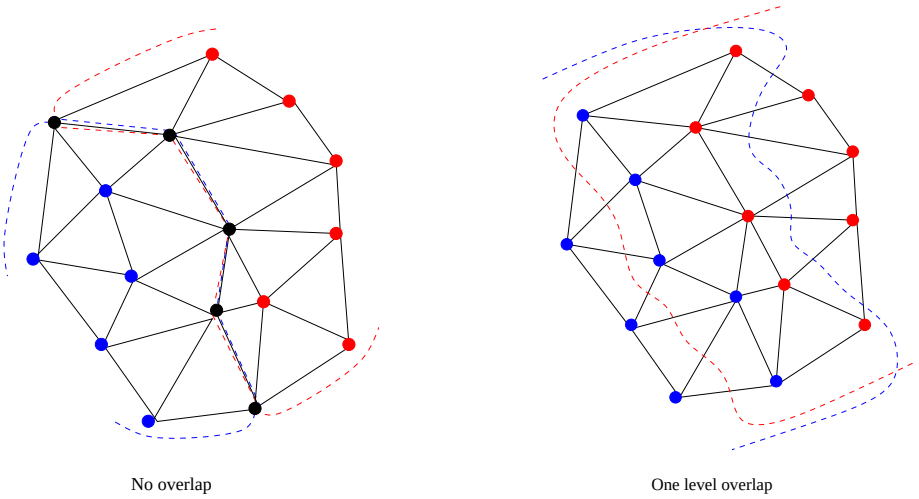

Figure 3.3: Different levels of overlap. Figure on the left (no overlap), internal boundary nodes cannot be removed without loosing elements. Figure on the right (1 level overlap), internal boundary nodes can be safely removed.

With one level of overlapping elements the nodes that are not owned by a given process are easily identified by the vector `ib_node_ids`, which is located in the `GridPart` class. Each entry in `ib_node_ids` is a local node number of an internal boundary node.

We can use the `ib_node_ids` vector to remove the interior boundary nodes, a technique described in [10].

3.5.2 Renumbering the nodes

Diffpack has a natural ordering of the nodes in the grid. With natural ordering we mean nodes keep their existing global numbers after grid partitioning. PETSc on the other hand uses a different ordering, a so called PETSc ordering (see fig. 3.4). In PETSc ordering, a grid with N nodes will be evenly and continuously divided among P processes. The ownership, given by `PetscSplitOwnership`, is determined by this simple formula: `N/size + ((N % size) > rank)`. For example if we are looking at N=1000 nodes, size=3 processes, process 0 would own the 334 first nodes, process 1 the next 333 nodes, and process 2 the remaining 333 nodes.

Now onto the problem of having different orderings. If values are entered into the linear system using natural ordering, PETSc will start reshuffling the data until the linear system has a PETSc ordering. This is totally unnecessary, and is likely to lead to a different problem, as well as it might become a great performance loss.

To be able to renumber the nodes, we must figure out what the global node numbering in Diffpack corresponds to the global node numbering in PETSc.

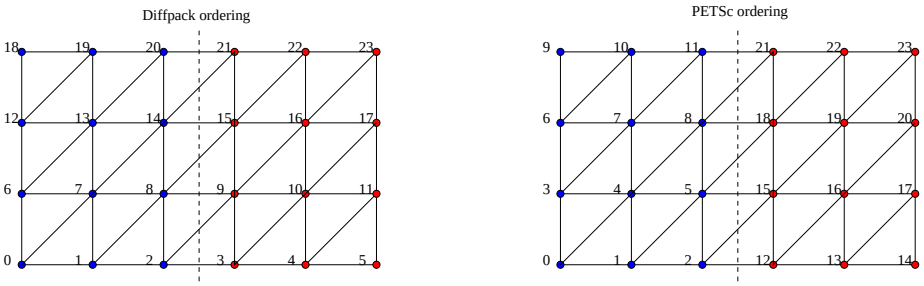

Figure 3.4: Different orderings of Diffpack and PETSc

Diffpacks node numbering is contained in vector `global_nnrs`, and like the interior boundary nodes `ib_node_ids` they are located in class `GridPart`. Using `global_nnrs` we can construct the PETSc equivalent, `petsc_nnrs`, which we can use we to construct the global PETSc matrix.

3.5.3 Creating a PETSc node numbering

To generate the PETSc node numbering array `petsc_nnrs` we can use a PETSc object, named `AO`, short for application ordering. `AO` objects are created to help map between different numberings of different applications and libraries. This object is collectively created, which we use to change the ordering of `global_nnrs`, by constructing `petsc_nnrs`. But in order to be able to properly create an `AO`, we must have an array of the global node numbers owned by a given processor; `global_nnrs` without the internal boundary nodes. Let us call this index set `diffpack_ordering`. Just to be sure, we make all the Diffpack numbering have base 0, opposed to being base 1.

Generating Diffpack ordering

```
void SubSystemManager :: generateDiffpackNumbering(
    const VecSimple(int)& global_nnrs,
    const VecSimple(int)& ib_node_ids,
    VecSimple(int)& dpack_order)
{
    // In this code we are removing the overlap, to get a
    // list of nodes that are uniquely owned by a process.
    // global_nnrs and ib_node_ids are objects of type
    // VecSimple(int) containing global node numbers and
    // interior boundary nodes
    int start = 0; int stop = ib_node_ids(1); int k = 1;
    for(int i = 1; i < ib_node_ids.size(); i++)
    {
        for(int j = start + 1; j < stop; j++)
            dpack_order(k++) = global_nnrs(j) - 1;
        start = ib_node_ids(i);
        stop  = ib_node_ids(i+1);
    }

    for(int i = stop + 1; i <= global_nnrs.size(); i++)
        dpack_order(k++) = global_nnrs(i) - 1;
}
```

Then once this is done, we use the function `AOApplicationToPetsc`, which returns `petsc_nnrs`, the PETSc equivalent of Diffpacks `global_nnrs`. `petsc_nnrs` is really nothing more than a 1 to 1 mapping of `global_nnrs`, and since a local node maps onto a global node via `global_nnrs` we have a one to one mapping between local node number and `petsc_nnrs`.

Generating petsc_nnrs

```
int SubSystemManager:: generatePetscNnrs(
    const VecSimple(int)& global_nnrs,
    const VecSimple(int)& ib_node_ids,
    VecSimple(int)& petsc_nnrs)
{
    // This function generates petsc_nnrs, the equivalent
    // of global_nnrs.
    if(mappingIsDone)
        return 1;

    VecSimple(int) dpack_order;
    generateDiffpackNumbering(global_nnrs, ib_node_ids,
        dpack_order);
    AO app_ordering;
    AOCreateMapping(PETSC_COMM_WORLD, dpack_order.size(),
        &dpack_order(1), PETSC_NULL, &app_ordering);

    petsc_nnrs.redim(global_nnrs.size());
    for(int i = 1; i <= petsc_nnrs.size(); i++)
        petsc_nnrs(i) = global_nnrs(i) - 1;
    AOApplicationToPetsc(app_ordering, petsc_nnrs.size(),
        petsc_nnrs.getPtr0());

    AODestroy(app_ordering);
    return 0;
}
```

Using `petsc_nnrs` we can start converting the linear system. But before we do that we need to preallocate the data structures, which if not done correctly might lead to a great performance loss.

3.5.4 Preallocating PETSc

Efficient use of PETSc requires preallocation of the matrices, this is to reduce the amount of allocations and copies when we insert values into a matrix. We will consider `MATMPIAIJ`, which is the format we will be using.

The `MATMPIAIJ` matrix has a diagonal and an off-diagonal portion for a given process. The diagonal portion corresponds to the nodes owned by a given process, while the off-diagonal portion corresponds to the nodes owned by another process (Figure 3.5).

Preallocation of a `MATMPIAIJ` matrix is done when creating the matrix using the function `MatCreateMPIAIJ` [5]. This function needs to know the number of nonzeros on the diagonal and off-diagonal portion of the part of the matrix owned by a given process. It leaves us with two options on specifying the nonzeros. We can either specify the maximum number of nonzeros on any row. Or we can pass an array with the exact number of nonzeros; one entry per row. The first option is certainly the easiest, but to minimize the memory allocation we will go for the latter option. The latter option is conceptually easy, but the implementation details are complicated by the bookkeeping. As we will see it

is quite feasible since we already have a parallel Diffpack matrix we can use to determine the number of nonzeros.

Figure 3.5 is an illustration of the format with the number of nonzeros. The blue numbers represents entries on the diagonal, the red numbers on the off-diagonal portion. `d_nnz` and `o_nnz` is the array specifying the number of nonzeros on the diagonal and the off-diagonal portion. From the figure can see here that CPU/process 0 on the diagonal portion has 2 nonzeros on the first and 1 nonzero on the second row, and on the off-diagonal portion 1 and 2 nonzeros on the first and second row respectively.

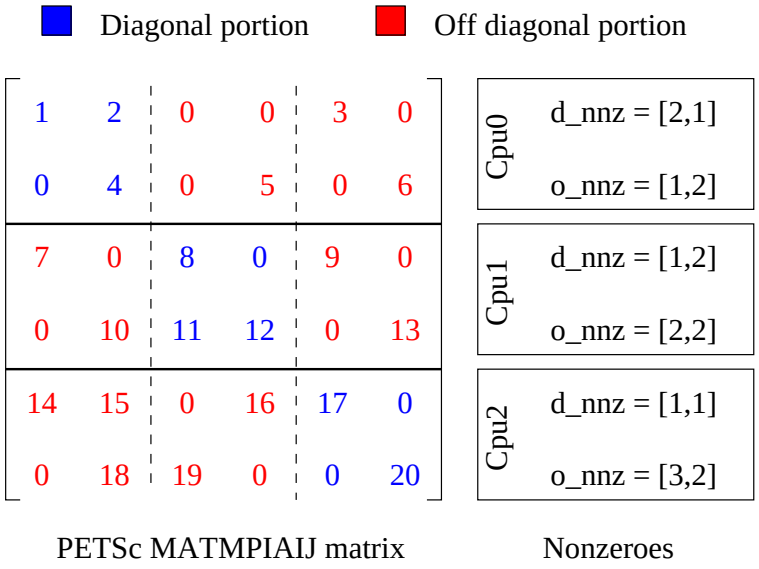

Figure 3.5: Diagonal and off diagonal portions of a PETSc matrix. `d_nnz` and `o_nnz` are arrays specifying number of nonzeros per row on the diagonal and off-diagonal portion of the matrix

To determine the number of nonzeros we loop over the column array in a Diffpack MatSparse matrix, counting the number of node numbers that is not an interior boundary node as a diagonal nonzero and the number of node numbers that is a interior boundary node. This is equivalent to checking the columns in `jcol` array of the MatSparse matrix belonging to a row owned by a given process for occurrences of local interior boundary node numbers defined by `ib_node_ids`. Listed below is an algorithm that does this and demonstrated in (figure TODO: figure).

Determine diagonal (d_nnz) and off diagonal (o_nnz) nonzeros

```
int MatVecHandler:: determineNNZ(
    MatSparse(real)& dpMat,
    VecSimple(int)& d_nnz,
    VecSimple(int)& o_nnz) const
{
...
    int start = 0; int stop = ib_node_ids(1);
    int value = 0; int row = 1;
    for(int i = 1; i < ib_node_ids.size(); i++)
    {
        for(int j = start + 1; j < stop; j++)
        {
            for(int k = pattern.irow(j); k < pattern.irow(j+1); k++)
            {
                value = pattern.jcol(k);
                if(bsearch(&value, ib_node_ids.getPtr0(),
                            ib_node_ids.size(), sizeof(int), int_cmp))
                {
                    ++o_nnz(row);
                }
                else
                    ++d_nnz(row);
            }
            row++;
        }
        start = ib_node_ids(i);
        stop  = ib_node_ids(i+1);
    }

    for(int j = stop + 1; j <= global_nnrs.size(); j++)
    {
        for(int k = pattern.irow(j); k < pattern.irow(j+1); k++)
        {
            value = pattern.jcol(k);
            if(bsearch(&value, ib_node_ids.getPtr0(),
                ib_node_ids.size(), sizeof(int), int_cmp))
            {
                ++o_nnz(row);
            }
            else
                ++d_nnz(row);
        }
        row++;
    }
}
}
```

In the code above we used the method `bsearch(...)`, which is a function from the C standard library, to perform binary search. An alternative method of finding internal boundary nodes in the jcol `ib_node_ids` is using a hash map. Though hash map takes up memory, it is a lot more efficient than binary search. Which we can consider later if this code becomes a bottleneck.

Verifying that the preallocation is correct can be done when running an application by passing `-info` to PETSc as described in the user manual.

3.5.5 Constructing the PETSc matrix and vector

After having created the PETSc global node numbering `petsc_nnrs`, and counted the number of nonzeros we can begin creating the PETSc matrix. First creating the matrix using `MatCreateMPIAIJ` using the arrays with diagonal and off-diagonal nonzeros.

Filling the PETSc matrix with values we loop over the rows of the Diffpack matrix, skipping interior boundary nodes. Using the mapping created earlier, `petsc_nnrs`, we map each column entry, filling a row buffer. The row buffer are then inserted into the matrix into the correct global row. Inserting rows using `MatSetValues` we are ensure that the values are put into the correct place: the diagonal portion or the off diagonal portion of the matrix.

A finishing call to the functions `MatAssemblyBegin` and `MatAssemblyEnd` and the matrix can be used.

In a similarly fashion the PETSc vector is created with `VecCreateMPI`. Values are inserted with `VecSetValues`, one row at a time. Finishing the construction with `VecAssemblyBegin`/`VecAssemblyEnd`.

See code on the next page for a demonstration.

Creation of Matrix and Vector

```
int MatVecHandler:: createPetscMatAndVec(LinEqSystem& system, Mat& A,
                                         Vec& x, Vec& b)
...
  // petsc_nnrs is the PETSc equivalent of global_nnrs
VecSimple(int) row_buffer(getMaxRowLength(pattern));
int row_start = 0, row_end = 0;
MatGetOwnershipRange(A, &row_start, &row_end);
int row_len = 0; int g_row = row_start;
for(int i = 1; i < ib_node_ids.size(); i++)
{
    for(int j = start + 1; j < stop; j++)
    {
    row_len = pattern.irow(j+1) - pattern.irow(j);
    row_ptr = pattern.irow(j);
    for(int k = pattern.irow(j); k < pattern.irow(j+1); k++)
    {
        row_buffer(l++) = petsc_nnrs(pattern.jcol(k));
    }
      MatSetValues(A, 1, &g_row, row_len, row_buffer.getPtr0(),
        &dpMat(row_ptr), INSERT_VALUES);
      VecSetValues(b, 1, &g_row, &dpVec(j), INSERT_VALUES);
      VecSetValues(x, 1, &g_row, &dpVecX(j), INSERT_VALUES);

      g_row++; l=1;
  }
  start = ib_node_ids(i);
  stop  = ib_node_ids(i+1);
}

for(int j = stop + 1; j <= global_nnrs.size(); j++)
{
    for(int k = pattern.irow(j); k < pattern.irow(j+1); k++)
    {
        row_len = pattern.irow(j+1) - pattern.irow(j);
        row_ptr = pattern.irow(j);
        for(int k = pattern.irow(j); k < pattern.irow(j+1); k++)
        {
            row_buffer(l++) = petsc_nnrs(pattern.jcol(k));
        }
        MatSetValues(A, 1, &g_row, row_len, row_buffer.getPtr0(),
            &dpMat(row_ptr), INSERT_VALUES);
        VecSetValues(b, 1, &g_row, &dpVec(j), INSERT_VALUES);
        VecSetValues(x, 1, &g_row, &dpVecX(j), INSERT_VALUES);

        g_row++; l=1;
    }
}
...
```

As with getting info about the success of the preallocation we can pass the command line argument -info to PETSc, to verify that values were put into the correct process[4].

3.5.6 Converting vectors between Diffpack and PETSc

To be able to return the solution to the Diffpack simulator we need to be able to convert the solution vector from the PETSc solver to the Diffpack solution vector in the `LinEqSystem`. We also need to convert vectors when we apply a PETSc preconditioner using a Diffpack solver, vica versa.

We can convert a PETSc vector to Diffpack by getting the pointer to the underlying data, filling the vector in the right places, ie. ignoring the internal boundary nodes. Finishing it off with a call to `GridPartAdm::updateInteriorBoundaryNodes(diffpackVec)` to ensure that the interior boundary nodes are update correctly.

Converting vectors from PETSc to Diffpack

```
void MatVecHandler:: petscToDiffpack(const Vec& petscVec,
                                     Vec(real)& diffpackVec)
{
    double* values = NULL;
    int petsc_vec_size = -1;

    VecGetArray(petscVec, &values);
    VecGetSize(petscVec, &petsc_vec_size);
    if(diffpackVec.size() != global_nnrs.size())
        diffpackVec.redim(global_nnrs.size());
    int start = 1, stop = ib_node_ids(1);
    int petsc_row = 0;
    for(int nnr = start; nnr < stop; nnr++)
        diffpackVec(nnr) = values[petsc_row++];

    for(int id = 1; id < ib_node_ids.size(); id++)
    {
        start = ib_node_ids(id) + 1;
        stop  = ib_node_ids(id + 1);
        for(int nnr = start; nnr < stop; nnr++)
        {
            diffpackVec(nnr) = values[petsc_row++];
        }
    }

    for(int nnr = stop + 1; nnr <= diffpackVec.size(); nnr++)
        diffpackVec(nnr) = values[petsc_row++];
    gp_adm->updateInteriorBoundaryNodes(diffpackVec);
    VecRestoreArray(petscVec, &values);
}
```

Converting a Diffpack vector to a PETSc vector is done in almost the same way, only this time we do not need to update the interior boundary nodes.

Converting vectors from Diffpack to PETSc

```
void MatVecHandler :: diffpackToPetsc(const Vec(real)& diffpackVec,
                                      Vec& petscVec) const
{
    TAU_PROFILE("Conversion", " ", CONVERSION);
    if(petscVec == NULL)
        createPetscVec(diffpackVec, petscVec);

    PetscScalar* values;
    VecGetArray(petscVec, &values);
    int i = 0;
    for(int nnr = 1; nnr <= ib_node_ids(1) - 1; nnr++)
    {
        values[i] = diffpackVec(nnr);
        ++i;
    }

    int nnr_start, nnr_stop;
    for(int id_nr = 1; id_nr <= ib_node_ids.size() - 1; id_nr++)
    {
        nnr_start = ib_node_ids(id_nr)     + 1;
        nnr_stop  = ib_node_ids(id_nr + 1);
        for(int nnr = nnr_start; nnr < nnr_stop; nnr++)
        {
            values[i] = diffpackVec(nnr);
            ++i;
        }
    }
    for(int nnr = nnr_stop + 1; nnr <= global_nnrs.size(); nnr++)
    {
        values[i] = diffpackVec(nnr);
        ++i;
    }
    VecRestoreArray(petscVec, &values);
}
```

Chapter 4

Experiments

In the previous chapter we have seen how Diffpack can be extended with Krylov subspace methods and preconditioners of PETSc. We defined methods of converting Diffpacks `MatSparse(real)` matrix and `Vec(real)` vector to the PETSc `MATMPIAIJ` matrix and to `VECMPI` vector, respectively. We also introduced some classes to make the PETSc solvers and preconditioners available to the Diffpack user, with minimal user effort needed.

Our strategy in converting the linear system from Diffpack to PETSc before solving it with a PETSc solver or applying a preconditioner results in little modification of an existing parallel Diffpack simulator. This also introduces some overhead, both in memory and with time it takes to solve the system.

In this chapter we will look at the performance of using Diffpack with PETSc. More specifically we will look at the performance of two types of scenarios that can occur:

Test A: Same solver/preconditioner If both libraries provide the same linear solver and preconditioner, will the PETSc solver outperform the Diffpack solver, or will it be dominated by the conversion time?

Test B: Different solver/preconditioner If there is significant overhead, will PETSc still have a superior solver or preconditioner?

4.1 Application example

In this example we modify the `Poisson1` simulator to gain access to the PETSc solvers and preconditioner classes we defined in Chapter 3.

The first step is to make the `Poisson1` simulator parallel, this has been described in [6].

After making the simulator parallel, we can add the lines needed by the PETSc extension.

In the file containing the `Poisson1.cpp` we first include the `SubSystemManager.h` header file to gain access to the global instance of `SubSystemManager`, which is named`global_manager`.

```
#include <SubSystemManager.h>
```

In the `Poisson1 ::scan` method we add the following line after `gp_adm` has been been initialized. This is needed to make global node numbering and the internal boundary nodes accessible to the PETSc classes:

```
global_manager->attach(*gp_adm);
```

Then in the main.cpp file we include the header file `initPETSc.h`

Then we add a call to `initPETSc()` after Diffpack and Parallel Diffpack has been initialized in the main method, this call initializes PETSc.

Then to make sure the libraries get destroyed in the correct order, we remove the call to `closeDpParallelLA` that finalizes Parallel Diffpack.

4.2 Test environment

The tests in this thesis are conducted on a Linux cluster named Chilopodus. The cluster consists of 24 nodes, with each node having the following characteristics. The GNU g++ compiler version 4.2.3 with `-O3` optimization flag was used to

CPU brand	2x Itanium 2 (Madison)
Clock	1300 Mhz
Cache	16 KiB L1, 256 KiB L2, 3 MiB L3
Memory	4 GB shared
Bus	400 Mhz
OS	Linux/GNU
Network interface	Gigabit Ethernet
MPI library	MPICH v1.2.7
Queue system	Torque (OpenPBS) v.2.2.1

Table 4.1: Specification for each node in the Chilopodus cluster

compile the libraries and the test programs.

4.3 How testing is performed

In the first scenario where we look at using the same solver and preconditioner time will be measured during solve, which include exclusive time and inclusive. The inclusive time is the time it takes to execute `lineq->solve` and exclusive time will only involve the call to the PETSc solver `KSPSolve`. To make sure timing has minimal overhead, we will only be using we will only be using `MPI_Wtime`.

In the other parts we will be using TAU[13] to measure the time spent during solve and conversion. TAU is a profiling tool capable of profiling parallel programs, measuring both time and memory usage.

4.3.1 Test simulator

In the tests we will be using the Poisson1 simulator we had a look at earlier in this thesis. The Poisson1 simulator is a classic Diffpack simulator that accom-

panies Diffpack library and has been used in [9] to explain the how to program Diffpack simulators.

4.3.2 Convergence criteria

Unless specified, in all the tests we will be using the default convergence criteria of Diffpack. The default convergence criteria consists of the convergence monitor `CMRelResidual` with a convergence tolerance of $1.0e-4$.

4.3.3 Measuring time with TAU

TAU provides us with lots of tools useful in profiling parallel programs and can interface with third party libraries and tools such as PAPI. We, however, will only be using it to measure the overall memory usage and the time of the three stages we defined in the previous section: the setup stage, the solver stage, and the conversion stage.

To measure the time we put TAU statements, that begin with `TAU`, into various places in the code. The statements stack, so one can safely have statements nested without having the time measurement overlap.

TAU is initialized with:

```
int main (int argc, const char* argv[])
{
    ...
    TAU_PROFILE_INIT(arg_l,arg_s);
    TAU_PROFILE_SET_NODE(myid);

    Poisson1 simulator;
    global_menu.multipleLoop(simulator);
}
```

The solve stage is measured by:

```
void Poisson1:: solveProblem () // main routine of class Poisson1
{
    ...
    TAU_PROFILE_TIMER(t1, "Solve", " ", TAU_USER);
    TAU_PROFILE_START(t1);
    lineq->solve();
    TAU_PROFILE_STOP(t1);
    ...
}
```

The conversion stage is measured by placing this statement at the beginning of the functions converting the matrices and vectors between Diffpack and PETSc.

```
TAU_PROFILE("Conversion", " ", CONVERSION);
```

To measure overall memory usage we use `TAU_TRACK_MEMORY()`. `TAU_TRACK_MEMORY()` enables memory tracking of memory, and an interrupt is generated every 10 seconds to measure the memory. The memory it measures is the memory allocated on the heap.

```
int main (int argc, const char* argv[])
{
    ...
    TAU_TRACK_MEMORY();
    ...
}
```

Just to make sure we get the memory involved in conversion, we do place a `TAU_TRACK_MEMORY_HERE()` at the end of `PetscSolver:: solve(...)`

4.4 Reporting speedup

In parallel computing there are two quantities, *speedup* and *efficiency*, that indicate the quality of a parallel program[6]:

$$S(P) = \frac{T(1)}{T(P)} \quad \text{Speedup}$$

$$\eta(P) = \frac{S(P)}{P} \quad \text{Efficiency}$$

To be able to measure true speedup we need to have the execution time with one CPU. In this thesis we have not developed support for converting Diffpack matrices to PETSc sequential sparse matrices (`MATSEQAIJ`), so we have to be a little creative. Though true speedup would be preferable.

The ideal situation is when we get linear speedup:

$$S(P) = P \tag{4.1}$$

We could assume that $S(1) \approx T(2)/2$. And calculate speedup, however if the solver or preconditioner for some reason differs greatly between sequential and parallel, our use of speedup would be totally wrong, and it could lead a being confused or in a worse case misunderstanding.

Instead we chose to calculate the relative improvement when doubling the CPUs, by calling it scaling. This would not be misunderstood with the usual definition with speedup.

So in an ideal situation we get perfect speedup if we get a relative improvement of 100%, and super linear speedup if more than that. For instance if the execution time with 2 CPUs is 120 seconds and 4 CPUs is 60 seconds, then the scaling becomes 2, indicating linear speedup.

4.5 Test A: Same solver/preconditioner

If a solver is present in both libraries, whom will be faster. Intuitively Diffpack will be faster because with PETSc we will have the overhead of converting the linear system.

For this test the choice is Conjugate Gradient, since it is part of both libraries and performed best among the Krylov Subspace solvers we tested, and we tested all solvers provided by both libraries. By best we mean; the solver that used the least amount of time, with the fewest number of iterations and with the best error. Though BiCGStab used fewer iterations to get a residual of $1.0e-4$ the error in the solution was higher than with Conjugate Gradient.

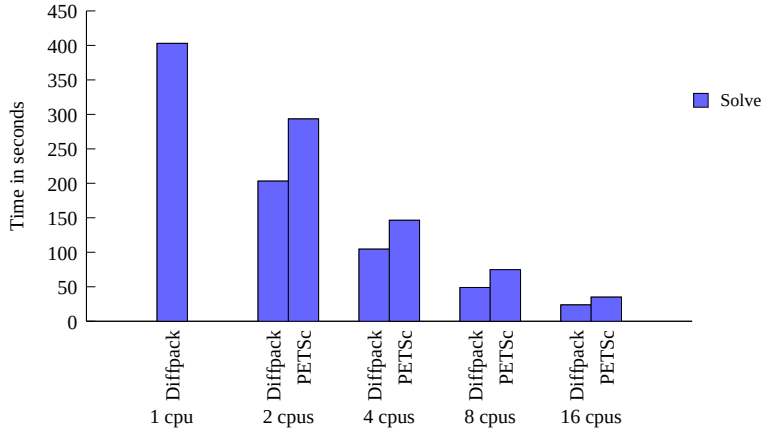

Figure 4.1: Conjugate Gradient in both libraries without preconditioning

4.5.1 Conjugate gradient

Test		Time		Result	
Cpus	Solv	Solve	Scaling	Iter	Error
1	Diffpack	402.9	N/A	1406	9.78e-05
2	Diffpack	203.5	1.98	1406	9.79e-05
	PETSc	293.4	N/A	1406	9.79e-05
4	Diffpack	104.3	1.95	1406	9.78e-05
	PETSc	146.5	2.0	1406	9.78e-05
8	Diffpack	49.0	2.12	1406	9.78e-05
	PETSc	74.7	1.96	1406	9.78e-05
16	Diffpack	23.7	2.07	1406	9.78e-05
	PETSc	34.9	2.14	1406	9.77e-05

Table 4.2: Efficiency of conjugate gradient without preconditioning. Time is measured in seconds. Solve time includes the time it takes to convert the linear system. A scaling of 2 corresponds to linear speedup.

From the test results we can see that both libraries scales perfectly, however Diffpack is considerably faster compared to PETSc. Both libraries have close to theoretical speedup, and in some cases super-linear speedup, which can be explained by improved cache performance due to increased number of processors.

The error and number of iterations give an indication that both implementations are quite similar, if not equal. The execution time however is not, with PETSc being approximately 50% slower. Exactly why PETSc performed much worse than Diffpack in this test is hard to tell, however by placing $MPI_W time$ inside the solve function of class `PetscSolver`:

```
double t1, t2, total_time; t1 = t2 = total_time = 0.0;
t1 = MPI_Wtime();
  ierr = KSPSolve(ksp, petscVec, petscLinsolVec); CHKERRQ(ierr);
t2 = MPI_Wtime() - t1;
```

```
MPI_Reduce(&t2, &total_time, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

, we were able to measure the exclusive time used by the PETSc library. Thus eliminating the doubt that our work is being responsible for the poor performance.

CPUs	Preconditioner	Time		
		Inclusive	Exclusive	Overhead
4	None	297.68	295.44	2.24
	SOR	77.25	75.12	2.12
8	None	148.96	147.16	1.8
	SOR	42.61	40.98	1.63
16	None	74.96	73.55	1.44
	SOR	22.38	21.12	1.26
16	None	35.99	34.87	1.12
	SOR	11.52	10.51	1.01

Table 4.3: Total time it takes to solve the system (inclusive) and the time the PETSc library uses to solve the linear system (exclusive). Time is measured in seconds.

4.5.2 Conjugate Gradient with SOR preconditioning

In this test we will compare the performance of preconditioned Conjugate Gradient. Preconditioning method used in this test is SOR which is also part of both libraries.

Listed below in table 4.4 we observe that Diffpack and PETSc have nearly equal performance in runtime. It is quite interesting to note that while Diffpack outperformed PETSc in the previous test without preconditioning, they are quite on par with SOR preconditioning. PETSc use a few iterations less than Diffpack, which is due to PETSc applying the preconditioner before solving the linear system. Though PETSc uses fewer iterations, it has bigger error.

The poor scaling in comparison to the uniprocessor gives an indication that none of the libraries has a true parallel SOR preconditioner. In PETSc case this can be confirmed by the documentation for SOR[5], stating that parallel SOR in PETSc corresponds to a block Jacobi with SOR preconditioning on each block. However by looking at the parallel performance comparing it to the case with 2 processors we notice that they scale equally *well*.

4.5.3 Overhead

Here we give an overview on the overhead of the preconditioning in the previous test with Conjugate Gradient and SOR preconditioning. This has been measured as described earlier by using TAU.

From the measurements listed in table 4.5 we notice that though the overhead on each processor decreases, the relative difference increases. Though the relative overhead increases, it can still be considered small.

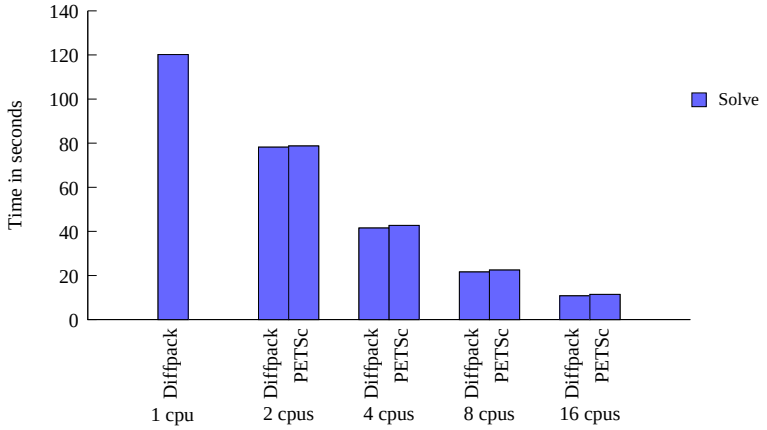

Figure 4.2: Conjugate Gradient with (S)SOR provided by both libraries. Time is measured in seconds. Grid size is 1000x1000 triangle elements

Test		Time		Result	
Cpus	Solv	Solve	Scaling	Iter	Error
1	Diffpack	120.2	N/A	172	8.55e-04
2	Diffpack	78.2	1.54	215	7.05e-03
	PETSc	78.7	2	214	7.17e-03
4	Diffpack	41.5	1.88	233	3.65e-03
	PETSc	42.7	1.84	230	4.05e-03
8	Diffpack	21.6	1.92	244	3.40e-03
	PETSc	22.4	1.9	240	3.90e-03
16	Diffpack	10.8	2.0	253	3.09e-03
	PETSc	11.4	1.96	249	3.51e-03

Table 4.4: Conjugate gradient with SOR preconditioning. Grid consists of 1000x1000 Triangle elements. Solve time includes conversion time between Diffpack and PETSc.

4.6 Test B: Different kinds of preconditioners

A more realistic scenario is to use a preconditioner that is not available in both libraries. In this test we will look at the performance of using Conjugate Gradient with preconditioners that are not available in both Diffpack and PETSc. Among those we choose a preconditioner that has good performance. The preconditioners chosen are based on trying out the various preconditioners selecting those that yielded the best results.

Selecting a PETSc preconditioner Some of the preconditioners were not tested as they, either did not support the `MATMPIAIJ` format, or simply were not parallel. We also did not test all the preconditioners from external packages, such as FFTW (Fastest Fourier Transform in the West) which could have yielded superb results. The one package we did test was HYPRE, where we got superior performance from the HYPRE BoomerAMG compared to the other

CPUs	Total time	Overhead	Relative
2	76.99	2.09	2.7%
4	42.54	1.64	3.8%
8	22.1	1.31	5.9%
16	11.43	1.03	9%

Table 4.5: Overhead of conversion between Diffpack and PETSc for the test involving Conjugate Gradient with SOR preconditioning

preconditioners we did test in PETSc and Diffpack.

Selecting a Diffpack preconditioner In Diffpack, the preconditioner of those tested, the RILU with a relaxation parameter of 0.8 performed best. Though SSOR preconditioner with a relaxation parameter of 1.8 had a nice performance, it did not scale as well in parallel compared to the RILU preconditioner.

ILU is one of the preconditioners in PETSc that is not supported in parallel. The RILU preconditioner in Diffpack is supported in parallel, so it will be interesting to see how it performs when we apply it to a PETSc linear system.

Of the preconditioners available from PETSc, the BoomerAMG preconditioner in the HYPRE package performed superior to the ones we did test.

4.6.1 RILU versus BoomerAMG

In this test we compare the performance of the BoomerAMG preconditioner to the RILU preconditioner, we also compare the performance and overhead of applying the preconditioners of one library to the solver of the other library. Time measurements for this test was done using TAU.

Listed below in table 4.6 are the test results.

From looking at the results we can safely say that BoomerAMG is superior in any way, it has superior execution time, error and iterations in comparison to the performance of the RILU preconditioner. BoomerAMG scales nicely with a close to halving of execution time when doubling the number of processors. However error increases a little with increasing number of CPUs, though this is negligible since the number of iterations is so low.

While the execution time with Diffpack/BoomerAMG had the lowest execution time and iterations, it used one less iteration but bigger error compared to PETSc/BoomerAMG. However if we view the iterations per time we notice that PETSc/BoomerAMG has slightly higher iterations/second. Though this difference is negligible when compared to the overall time.

RILU preconditioner performed nicely with almost perfect scaling. However the iterations increases as the number of processors increase. On the other hand the error decreases, so this is no indication loss of accuracy with increasing number of processors.

The execution time of PETSc/RILU is a lot worse execution time than Diffpack/RILU, however PETSc/RILU uses a lot more iterations, especially on 2 CPUs where the difference in iterations are as high as 10.6%. But this difference diminishes with increasing number of processors where time becomes

more comparable. This is not surprising since vectors are converted within the preconditioner with each iteration. The overhead in conversion time in PETSc/RILU is also highest among the other combination, but is still small and greatly reduced with increasing number of processors. On 2 CPUs the overhead accounts for 7.1%, still relatively small. The error is slightly better with PETSc/RILU, but this is likely due to PETSc/RILU using more iterations.

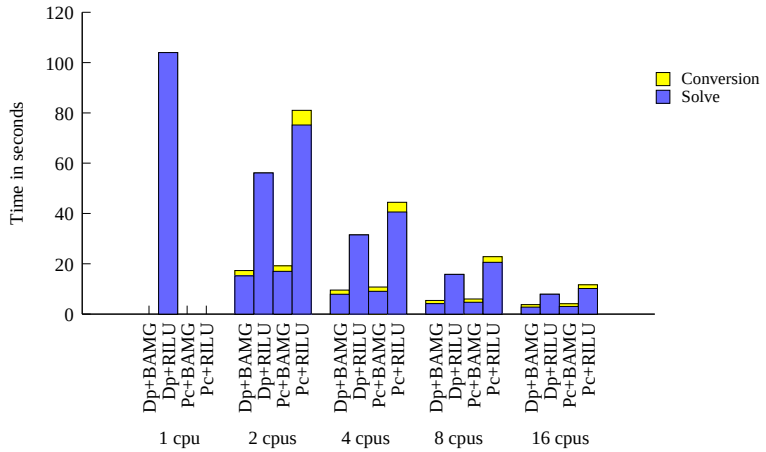

Figure 4.3: Speedup of the conjugate gradient

4.6.2 Memory

To get an overview on the memory usage in using PETSc with Diffpack we use TAU. TAU has the capability of sampling the memory on the heap. Triggering TAU to take a sample of the heap memory can be done in two ways; Periodic triggering with `TAU_TRACK_MEMORY` and explicit triggering at a given execution point with `TAU_TRACK_MEMORY_HERE`. The last statement is preferable since we get more control over where the memory is tracked. However, since we do not have access to the source code of Diffpack we will rely on using the periodic memory tracking mainly.

The periodic memory tracking, defaults to sampling the memory every 10 seconds, which can be modified with `TAU_SET_INTERRUPT_INTERVAL(value)`. We will be using both TAU triggering options, and we will set the periodic memory tracking down to 1 second, which was the lowest we could set.

The explicit triggering points are placed in before and after `lineq->solve`.

The problem with the periodic tracking, it is kind of unreliable since there is limited we can do to control when and where memory is tracked. This is especially true with the superior performance we got with CG/BoomerAMG. We had to set the convergence tolerance down to $1.0e-30$ to get more than 20 iterations. One thing we could do though, is control where the periodic tracking starts, so we place `TAU_TRACK_MEMORY` right before `lineq->solve`.

The full input files can be seen in the appendix. But we give a summary in

Test			Time			Result	
CPUs	Solver	Preconditioner	Solve	Conv	Total	Iter	Error
1	Diffpack	RILU	104.0	N/A	104.0	178	5.87e-04
2	Diffpack	RILU	56.1	N/A	56.1	187	6.44e-03
	PETSc	RILU	75.2	5.8	81.0	207	4.40e-03
	PETSc	BoomerAMG	17.0	2.2	19.2	4	1.61e-05
	Diffpack	BoomerAMG	15.2	2.1	17.3	3	3.08e-04
4	Diffpack	RILU	31.5	N/A	31.5	199	4.30e-03
	PETSc	RILU	40.6	3.8	44.5	216	2.22e-03
	PETSc	BoomerAMG	9.1	1.7	10.8	4	4.44e-05
	Diffpack	BoomerAMG	7.9	1.7	9.6	3	5.63e-04
8	Diffpack	RILU	15.8	N/A	15.8	207	3.68e-03
	PETSc	RILU	20.6	2.2	22.8	220	2.13e-03
	PETSc	BoomerAMG	4.7	1.3	6.0	4	2.02e-05
	Diffpack	BoomerAMG	4.2	1.3	5.4	3	4.67e-04
16	Diffpack	RILU	7.9	N/A	7.9	215	3.28e-03
	PETSc	RILU	10.2	1.5	11.7	226	2.03e-03
	PETSc	BoomerAMG	3.1	1.0	4.1	4	4.60e-05
	Diffpack	BoomerAMG	2.8	1.0	3.8	3	7.32e-04

Table 4.6: Different kinds of combinations of preconditioners not available in both libraries. The solver type used in both libraries is Conjugate Gradient. The RILU preconditioner is only available in the Diffpack library. The BoomerAMG preconditioner is only available in PETSc. Time is reported in seconds. Solve time is the time it takes to solve the linear system. Conv time involves anything related to converting data structures between the libraries. Tot is the total time used to convert and solve the linear system.

In this test we have not conducted any time measurements to avoid them being affected by the memory tracking. Though it is not the purpose of this test, an indication of performance can be viewed at the number of samples taken between the tests.

In table 4.7 we can view the results of the memory test. The measurements with BoomerAMG may deviate some because of the few samples in particular with 16 processors. But the measurements give an estimate about the memory usage of the various combinations of library solvers and preconditioners.

From the table, we can see that as the number of CPUs increase, so does memory. The memory leap between one and two CPUs is the highest, giving an indication that not all memory usage is distributed evenly, most likely with objects that are needed by all the CPUs. Aside from the leap from a single CPU to two CPUs , the memory is close to constant with increasing number of CPUs.

Diffpack/RILU had the least amount of memory usage. This is to be expected as no extra matrices and vectors are introduced than are already present in the simulator.

What is kind of surprising is that PETSc/RILU uses less memory than PETSc/BoomerAMG, since the PETSc/RILU involves more data structures; including a Diffpack linear system, a PETSc linear system and the Diffpack and PETSc preconditioner data structures. Though it may be that BoomerAMG

uses more memory than RILU. PETSc/BoomerAMG is the only one whose memory decreases with increasing number of CPUs, though not by much, but the number of samples are so low that this cannot be confirmed.

Diffpack/BoomerAMG used the most memory of those tested. This may be a confirmation that BoomerAMG generally uses more memory than RILU.

A summary of the memory usage is given in Figure

Test			Memory	
CPUs	Solver	Preconditioner	#Samples	Memory
1	Diffpack	RILU	455	336
2	Diffpack	RILU	243	429
	PETSc	RILU	319	567
	PETSc	BoomerAMG	70	678
	Diffpack	BoomerAMG	72	669
4	Diffpack	RILU	123	433
	PETSc	RILU	161	573
	PETSc	BoomerAMG	38	662
	Diffpack	BoomerAMG	61	717
8	Diffpack	RILU	65	440
	PETSc	RILU	83	577
	PETSc	BoomerAMG	22	627
	Diffpack	BoomerAMG	46	722
16	Diffpack	RILU	32	452
	PETSc	RILU	43	584
	PETSc	BoomerAMG	15	609
	Diffpack	BoomerAMG	40	742

Table 4.7: Different kinds of combinations of preconditioners not available in both libraries. This table summarizes memory usage. Memory usage is measured in MB. Memory usage is the average of all the samples, which has been summed over all the CPUs in the test. Grid size 1000x1000 is the same as in the previous tests

Chapter 5

Conclusion

In this master thesis we wanted to investigate the possibility of extending Diffpack with PETSc. We were faced with these questions:

1. Can Diffpack be extended with PETSc Krylov Subspace Solvers and Preconditioners in such a way that no modification of the Diffpack library is required?
2. Does application of PETSc solvers and preconditioners require modification of an existing Diffpack simulator?
3. Can the Krylov Subspace Solvers and Preconditioners of PETSc outperform those already existing in Diffpack?

To answer these questions we defined and introduced new class hierarchies `PetscSolver` and `PetscPrecond`. These new classes were added to the Diffpack class hierarchy.

To make these new solvers and preconditioners compatible with the Diffpack library we developed algorithms on how to convert the matrices and vectors between Diffpack and PETSc.

We increased the user experience by not only making PETSc's preconditioners available to Diffpack solvers, but also the ability to use Diffpack preconditioners with the PETSc solvers, thus allowing the user to further experiment with different kinds of combinations of solvers and preconditioners. Because experimentation with different kinds of options is an important aspect of Diffpack.

By the same philosophy we changed the default convergence criteria of PETSc Krylov Subspace methods to support the most common convergence criteria of Diffpack Krylov Subspace methods.

After which we tested and compared the performance of CG from both libraries without preconditioning, with the same preconditioner (SOR) and with different preconditioners when used with both libraries. Where we found that with CG without preconditioning that PETSc was much slower than Diffpack. We also discovered that CG with (S)SOR preconditioning were equally good, including the overhead. We tested different combinations of preconditioners with CG and measured the time it took to solve and convert the system, in addition to the memory usage.

Can Diffpack be extended with PETSc Krylov Subspace Solvers and Preconditioners in such a way that no modification of the Diffpack library is required? -

Yes, Diffpack can be extended with PETSc solvers and preconditioners without having to modify the libraries.

Does application of PETSc solvers and preconditioners require modification of an existing Diffpack Simulator? - To be able to successfully use the PETSc solvers and preconditioners after extending Diffpacks class hierarchy requires removal of one statement and addition of two statements and one include preprocessor directive.

Can the Krylov Subspace Solvers and Preconditioners of PETSc outperform those already existing in Diffpack? - In some cases, if both libraries supply the same preconditioner and solver the efficiency is at best on par with Diffpack. But if PETSc has a solver and preconditioner that is not part of Diffpack, - then yes PETSc can out perform Diffpack, despite the extra overhead of converting the linear system. The over head can be considered low; in our experiments the overhead was between 1-3 seconds and less than 15%.

5.1 Future work

Future work to be done is to add support for uniprocessor PETSc with Diffpack and add support for vector PDEs, the work in this thesis only covers parallel and scalar PDEs.

Adding support for uniprocessor, is equivalent of adding support for the PETSc serial sparse matrix, `MATSEQAIJ`. Algorithms to convert the Diffpack matrix to this format needs to be developed. Since `MATSEQAIJ` is just a simpler version of `MATMPIAIJ`, the algorithms developed in this thesis can be simplified for this purpose:

Preallocation of the `MATSEQAIJ` matrix we need only count the number of non zeros per row in the Diffpack matrix; there is no off-diagonal portion to consider.

Transferring data from a Diffpack matrix to a `MATSEQAIJ` can be done directly as there is no need to drop internal boundary nodes as they do not exist for uniprocessor simulations.

This is a minor issue, but should be done; implement the solvers and preconditioners from third party PETSc libraries as stand alone classes. For example the Hypre package was added to Diffpack as a standalone preconditioner PetscPCHYPRE. It would be better to provide a separate preconditioner class for each of the preconditioners.

Bibliography

[1] Diffpack parallel toolbox. `http://diffpack.com/products/parallel1/para1_main.html`.

[2] Diffpack software package. `http://www.diffpack.com`.

[3] Summary of sparse linear solvers available from PETSc. http://www-unix.mcs.anl.gov/petsc/petsc-2/documentation/linearsolvertable.html.

[4] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[5] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. Online Documentation for PETSc. `http://www-unix.mcs.anl.gov/petsc/petsc-as/snapshots/petsc-current/docs/`. 10. Jan. 2008.

[6] X. Cai, E. Acklam, H. P. Langtangen, and A. Tveito. *Advanced Topics in Compuational Partial Differential Equations*, chapter 1. Springer, 2003.

[7] Johan Hoffman, Johan Jansson, Anders Logg, and Garth N. Wells. Dolfin homepage, 2008. http://www.fenics.org/wiki/DOLFIN.

[8] H. P. Langtangen. Tips and frequently asked questions about diffpack. Technical Report 1, Numerical Objects, 1991.

[9] Hans Petter Langtangen. *Computational Partial Differential Equations - Numerical Methods and Diffpack Programming*. Textbooks in Computational Science and Engineering. Springer, 2nd edition, 2003.

[10] Guo Wei Ma. An Extension of Parallel Diffpack with Trilinos, 2007.

[11] James D. Teresco, Karen D. Devine, and Joseph E. Flaherty. Partitioning and dynamic load balancing for the numerical solution of partial differential equations.

[12] Martin Burheim Tingstad. Improving Inter-subdomain Communication and Load-balancing for the Parallel Diffpack Library, 2007.

[13] Los Alamos National Laboratory University of Oregon and Research Centre Jlich. TAU (tuning and analysis utilities), 2008. http://www.cs.uoregon.edu/research/tau/home.php.

Appendix A

Makefiles

The standard Diffpack Makefiles can be created with the Mkdir command. `Mkdir mydirectory` populates the specified directory mydirectory with these files: `Makefile`, `cmake1` and `cmake2`.

The file `Makefile` is the main Makefile, it includes the two other Makefiles, `.cmake1` and `.cmake2`. The easiest way of affecting the way the Makefile compiles and links is through `.cmake1` and `.cmake2`.

`.cmake1` is used to specify additional compile flags, while `.cmake2` is used to specify additional libraries to compile and link against.

Provided in the next two sections are examples of `.cmake1` and `.cmake2`, used in this thesis to compile and link with PETSc and TAU.

A.1 Makefiles for parallel Diffpack with PETSc

.cmake1

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

# The packages to be used by this application/library:
PACKAGES = $(DPR) $(LAR) $(BTR)
PROJECTROOT = $(PDPR)

CXXUF += -fPIC
```

.cmake1

```
# 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 line below includes a PETSc make file that defines variables needed to
# compile and link with PETSc. We need access to two of them: $PETSC_INCLUDE
# and ${PETSC_KSP_LIB}. The contents of these two variables can be entered here
# manually, however by including the make file below this Makefile will be
# A. Platform independant
# B. Easy to change between different installations of PETSc: for instance with
# or without optimalization
# C. Likely to be compatible with future releases of PETSc

include ${PETSC_DIR}/bmake/common/variables

# Compile and link with some external software packages:
INCLUDEDIRS += ${PETSC_INCLUDE}
LDPATH      += -L/usr/lib/mpich/lib ${PETSC_KSP_LIB}
LIBS        += ${PETSC_KSP_LIB}
```

A.2 Makefiles for parallel Diffpack with PETSc and TAU

.cmake1

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

# The packages to be used by this application/library:
PACKAGES = $(DPR) $(LAR) $(BTR)
PROJECTROOT = $(PDPR)

CXXUF += -fPIC -DPROFILING_ON  -DTAU_STDCXXLIB -DTAU_GNU
CXXUF += -DTAU_DOT_H_LESS_HEADERS -fPIC -DTAU_MPI
CXXUF += -DTAU_MPI_THREADED -DTAU_LARGEFILE  -D_LARGEFILE64_SOURCE
CXXUF +=  -DTAU_WEAK_MPI_INIT -DMPICH_IGNORE_CXX_SEEK
```

.cmake1

```
# 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.
#
#

include ${PETSC_DIR}/bmake/common/variables

# Compile and link with some external software packages:
INCLUDEDIRS += ${PETSC_INCLUDE} -I/home/mfhoel/src/tau-2.17/include
LDPATH      += -L/usr/lib/mpich/lib ${PETSC_KSP_LIB}
LDPATH      += -L/home/mfhoel/src/tau-2.17/ia64/lib
LIBS        += ${PETSC_KSP_LIB} -lTauMpi-mpi-pdt -ltau-mpi-pdt
```

Appendix B

Installing PETSc

PETSc was installed with HYPRE in a few steps: First set this system variable (and put them in `/.bashrc`):

```
export PETSC_DIR=/home/yourusername/petsc-2.3.3-p12-opt
```

Now one may have to log out and in for the variable to be loaded *or* enter `source /.bashrc`. Finally PETSc can be installed:

```
yourusername@chilopodus:~/petsc-2.3.3-p12-opt$ ./configure \
--with-mpi-dir=/usr/lib/mpich/ \
--download-f-blas-lapack=1 \
--with-clanguage=cxx --with-debugging=0 CXXOPTFLAGS=-O3 COPTFLAGS=-O3 \
FOPTFLAGS=- O3 --shared=0 --with-shared=0 --download-hypre=1
mfhoel@chilopodus:~/petsc-2.3.3-p12-opt$ make all test
```

You may run into trouble with using the Makefile in Appendix A now that PETSc was installed using MPICXX. Edit the which is located in `$PETSC_DIR/bmake/$PETSC_ARCH/petsc_conf` file and change `mpicxx` with `g++`.

You may want to make a backup copy of the `petsc_conf` file.

In this section we have a command that may be used to change mpicxx to g++. First to ensure that it will work this command will print the proposed changes (from inside the directory with the petsc_conf file):

```
sed -n "\|mpicxx| s|\(^[^=]*\).*mpicxx|\1 = "`which g++`"|p" petscconf
```

Now execute the next command if you are satisfied with the proposed changes:

```
sed -i "\|mpicxx| s|\(^[^=]*\).*mpicxx|\1 = "`which g++`"|" petscconf
```

Appendix C

Installing TAU

TAU was installed on Simulas Linux cluster Chilopodus, which is an ia64 architecture.

Before installing TAU, it is necessary to install Program Database Toolkit (PDT), a package some of TAUs functionality depends on. PDT was installed in `/home/mfhoel/src/pdtoolkit-3.12`.

Installing TAU is pretty straightforward. We ran into a problem though with MPICH version 1, which ended in hanging application on more than 2 cpus. TAU caught `SIGUSR1` signals generated from MPICH. MPICH uses SIGUSR1 to for internal communication. To fix this, it was necessary to comment out these lines inside `tau-2.17/src/Profile/MetaData.cpp`:

```
  /* register SIGUSR1 handler */
//  if (signal(SIGUSR1, tauSignalHandler) == SIG_ERR) {
//    perror("failed to register TAU profile dump signal handler");
//  }
//
//  if (signal(SIGUSR2, tauToggleInstrumentationHandler) == SIG_ERR) {
//    perror("failed to register TAU instrumentation toggle signal handler");
//  }
```

On Chilopodus TAU was successfully installed in two steps:

```
./configure -mpi -c++=g++ -fortran=gnu -pdt=/home/mfhoel/src/pdtoolkit-3.12
make clean install
```