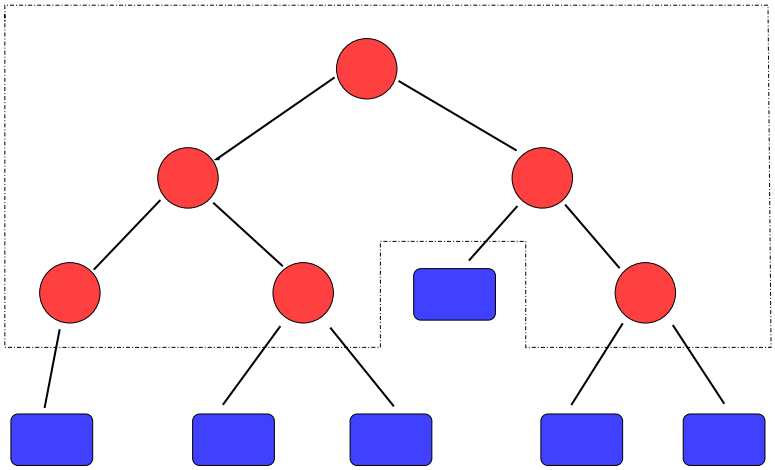

IMPLEMENTING A COMPONENT BASED PARALLEL DISTRIBUTED FINITE ELEMENT SOLVER

FILIP JOHANSSON and FREDRIK HANSSON

Structural Mechanics

Master's Dissertation

Department of Construction Sciences
Structural Mechanics

ISRN LUTVDG/TVSM--08/5155--SE (1-89)
ISSN 0281-6679

IMPLEMENTING A COMPONENT BASED PARALLEL DISTRIBUTED FINITE ELEMENT SOLVER

Master's Dissertation by

FILIP JOHANSSON and FREDRIK HANSSON

Supervisors:

Jonas Lindemann, PhD,
LUNARC, Lund

Examiner:

Ola Dahlblom, Professor,
Div. of Structural Mechanics

Printed by KFS I Lund AB, Lund, Sweden, March, 2008.

For information, address:
Division of Structural Mechanics, LTH, Lund University, Box 118, SE-221 00 Lund, Sweden.
Homepage: http://www.byggmek.lth.se

Abstract

Parallel computing is becoming more and more important in modern Finite Element Software. As problems grow larger, computation on a single processor may not be fast enough. To overcome this problem, one can utilize parallel programming using e.g SMP-machines or clusters. Furthermore, when local compute resources are scarce, it can be quite convenient to take advantage of non-local resources for performing the calculations.

This thesis is a collaboration between the Division of Structural Mechanics, LTH and StruSoft AB in Malmö/Budapest. The aim is to address the issues above, specifically looking at the structural analysis program FEM-Design developed by StruSoft. There are several ways to parallelize code, focus will be on OpenMP, PETSc and Intel MKL. These methods have been studied in order to conclude which one is the most suitable for existing Finite Element applications. In the end Intel MKL was chosen and implemented. Regarding the distributed computations, a realistic client/server application was developed using the Internet Communications Engine (Ice).

The parallel properties of the implementation was studied and also, during a visit to the StruSoft Budapest branch, the implementation was integrated into FEM-Design. The results were astonishing, reaching speedups of a factor up to 360 compared to the original solver. Also, the scaling was almost linear for the implemented solver.

Acknowledgements

First of all we would like to thank our supervisor Jonas Lindemann for his support during the thesis process and many interesting discussions. We would also like to thank all the people at StruSoft in Malmö and Budapest. Specially Arpad Tornyos for his technical support during our visit to Budapest. Tamas Racz for his help with communicating with the people in Budapest and his overall involvement in the work. Also we owe Håkan Hansson a great deal of thanks for making this thesis possible. Futhermore we would like to thank Kent Persson at the Divison of Structural Mechanics for inspiring us to use Intel MKL. Also we would like to thank the opponents Mikael Månsson and Magnus Tingne for their many useful comments on this report.

Contents

Chapter 1

Introduction

Parallel computing is becoming more and more important in modern Finite Element Software. As problems grow larger, computation on a single processor may not be fast enough. To overcome this problem, one can utilize parallel programming using e.g smp-machines or clusters. Furthermore, when local compute resources are scarce, it can be efficient to take advantage of non-local resources for performing the calculations. The latter part is often referred to as distributed computations.

The focus is on parallelizing existing Finite Element Codes. For this purpose some different techniques have been studied, these are OpenMP, PETSc and Intel MKL. Also, the possibility of distributing the calculations to non-local resources using e.g. the Internet Communications Engine (Ice) is investigated. This thesis is a collaboration between the Division of Structural Mechanics, LTH and StruSoft AB in Malmö/Budapest. The aim is to address the issues above, specifically looking at the structural analysis program FEM-Design developed by StruSoft.

The structure of the report is as follows: Chapter 2 gives a brief description of the tools for parallelization mentioned above. Chapter 3 describes the process of implementing the parallel solver. In chapter 4, the integration of the parallel solver into FEM-Design is discussed. Chapter 5 gives an overview of distributed programming tools, specifically focusing on Ice. Chapter 6 treats the implementation of the distributed interface for the parallel solver. In chapter 7, some conclusions from the present work are discussed, also, some suggestions for future work are given. Most of the code written in this project is available in appendix A and B.

Chapter 2

Parallel programming

In this section some approaches to parallel programming will be reviewed. The purpose of parallel programming is to divide tasks between different CPUs in order to reduce the computational time. When doing this it is important how the problem is divided and how the different processes (threads) communicate and synchronize with each other. One of the most used tools for handling this communication is the Message Passing Interface, MPI for short. The advantages of parallel programming are twofold, more CPU power can be utilized, also, the total amount of memory is increased with more CPU:s. The latter being very significant for larger problems. Symmetric multi processing (SMP) systems like dual/quad core are becoming more and more popular, but far from all programs actually use the parallel capacity of these systems. As the CPU clock speed curve is starting to level out, parallel programming is becoming more and more important.

In this thesis, three common approaches to parallel programming are studied, in the context of a finite element solver:

OpenMP [5] A set of compiler directives that can be directly inserted in to existing code.

PETSc [15] A library for solving Partial Differential Equations using MPI.

Intel MKL [4] A library of threaded math routines based on OpenMP. It is highly optimized for Intel processors.

It should be noted that the techniques studied are applied to medium sized finite element problems. Larger finite element problems often require some kind of domain decomposition scheme in addition to the techniques described in this report.

2.1 OpenMP

OpenMP stands for Open specifications for Multi Processing. It is a collaborative work between interested parties from the hardware and software industry, government and academia. Essentially, OpenMP is an API for parallelizing existing code, giving the programmer full control over the parallelization. It is comprised of three primary API components, these are

- Compiler directives
- Runtime library routines
- Environment variables.

OpenMP is mainly designed for symmetric multiprocessing (SMP) architectures i.e. systems with multiple CPUs on a shared memory. OpenMP is available for Fortran(77, 90 and 95) and C/C++. It has been implemented on many platforms including most Unix platforms and Windows NT. A full description of the OpenMP specification can be found in [5].

2.1.1 Programming model

OpenMP uses a shared memory process consisting of multiple threads. The fork-join model of parallel execution is utilized, see figure 2.1. The OpenMP program is initially

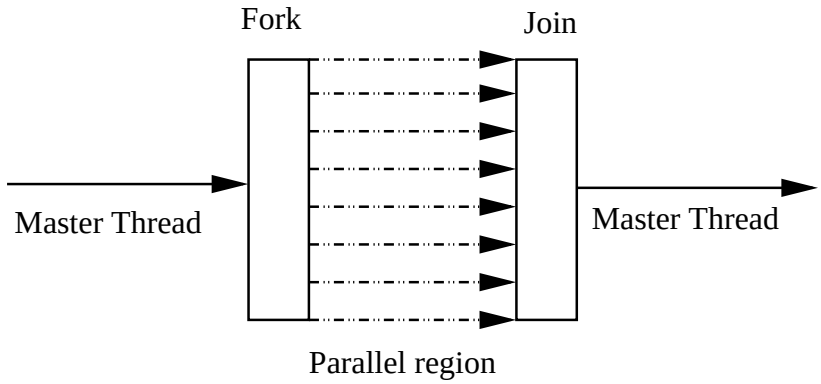

Figure 2.1: The master thread creates a team of parallel threads

executed as a single process by the *master thread*. When entering a parallel region, the master thread creates a team of parallel threads, this is called *fork*. The statements in the program that are enclosed by the parallel region are then executed in parallel among the threads in the team. When all threads have finished their work in the parallel region, they synchronize and terminate, leaving only the master thread, this is called *join*. In the parallel region, OpenMPs memory model dictates that all variables except loop variables by default are shared by the different threads. One can alter this by specifying variables as private or shared before entering the parallel region. The model also supports nested parallelism i.e. it is possible to have parallel regions within other parallel regions.

2.1.2 Usage

OpenMP is used by inserting special compiler directives or runtime library routine calls into the existing code. The compiler directives are inserted as special comments (beginning with !$OMP in Fortran 90). Examples of directives are

!$OMP PARALLEL/ !$OMP END PARALLEL begin and end parallel region

!$OMP DO/!$OMP END DO this is a work sharing construct. It divides the enclosed loop between the different threads

Other constructs in OpenMP include:

- Static/dynamic sheduling of work-divison constructs.
- The possibility to divide code in different parallel running sections
- Synchronization tools like barriers and the possibilty to specify sections of code within parallel regions to be executed serially.
- Tools to lock/unlock certain parts of data structures such as matrices/vectors.

and much more, but obviously, a full description of OpenMP cannot be made here. A nice guide to using OpenMP can be found in [6].

2.1.3 Code examples

In 2.1, the basic usage of OpenMP is illustrated. This program is a simple example that allocates two large arrays and performs some simple calculations on them. First, **x** is initialized, which is done serially. The value of the environment variable *OMP_NUM_THREADS*, which should be set to the number of processors for best performance, is printed using the OpenMP method *omp_get_num_threads()*. The program then enters the parallel region where **a** is calculated. Here, the work is divided equally between the threads, but it is also possible to divide the work in different proportions using the *!$OMP SCHEDULE* directive.

Listing 2.1: test.f90

```
program main
   use omp_lib
   integer :: i, j
   integer(8) :: size = 5e7
   real(8) :: start_time, finish_time
   real(8), dimension(:), allocatable :: a , x

   allocate(a(size))
   allocate(x(size))

   do i=1,size
    x(i) = (i+2)/1000
   end do

   start_time = omp_get_wtime()

    write(*,'(A,I)') 'Maximum number of threads: ',
     &omp_get_max_threads()

   !$OMP PARALLEL do
   do i=1,size
      a(i) = sin ( x(i) / 2 )
   end do
```

```
    !$OMP END PARALLEL do

    deallocate(a)
    deallocate(x)

    finish_time = omp_get_wtime()

    write(*,'(A,F)') 'Time: ', finish_time-start_time

end program main
```

In this example, the result in one calculation did not depend on the result from another iteration, in the following example this is not the case.

```
!$OMP PARALLEL do shared(a)
! Parallel section executed by all threads
do i=1,size-1
    a(i) = a(i+1)
end do
! All threads join master thread and disband
!$OMP END PARALLEL do
```

This is because $a(i+1)$ is needed in its unmodified form for the correct result to be obtained, however since the loop is now divided between the CPUs we cannot be certain of this. The above situation is an example of a so called *race condition*, meaning that the result of the code depends on the thread scheduling and the speed of each processor. In some cases, there are workarounds to these problems. For example, one could use some OpenMP directives to enforce synchronization between threads, or one could modify the loop itself. Of course, it is necessary to evaluate the cost in terms of time and memory in order to see, if it is worth it or not.

2.2 PETSc

The Portable, Extensible Toolkit for Scientific Computations [15] is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations. It employs the MPI standard for all message-passing communication. The original purpose of PETSc was to enable its users to easily experiment between many different models, solving methods and discretizations and to eliminate the MPI from MPI programming. It is a freely available research code usable from C/C++, Fortran 77/90 and Python. PETSc has run problems with over 500 million unknowns, also it has run efficiently on 6,000 processors. It has also been used in applications running at over 2 Teraflops. An overview of some of the components of PETSc can be seen in figure 2.2. This figure illustrates the hierarchical structure of PETSc, an important feature of the package is the possibility to start at a high level and work your way down in level of abstraction. The essential components of PETSc are:

Vec Provides the vector operations required for setting up and solving large-scale linear

and nonlinear problems. Includes easy-to-use parallel scatter and gather operations, as well as special-purpose code for handling ghost points for regular data structures.

Mat A large suite of data structures and code for the manipulation of parallel sparse matrices. Includes four different parallel matrix data structures, each appropriate for a different class of problems.

PC A collection of sequential and parallel preconditioners, including (sequential) ILU(k), LU, and (both sequential and parallel) block Jacobi, overlapping additive Schwarz methods and (through BlockSolve95) ILU(0) and ICC(0).

KSP Parallel implementations of many popular Krylov subspace iterative methods, including GMRES, CG, CGS, Bi-CG-Stab, two variants of TFQMR, CR, and LSQR. All are coded so that they are immediately usable with any preconditioners and any matrix data structures, including matrix-free methods.

SNES Data-structure-neutral implementations of Newton-like methods for nonlinear systems. Includes both line search and trust region techniques with a single interface. Employs by default the above data structures and linear solvers. Users can set custom monitoring routines, convergence criteria, etc.

TS Code for the time evolution of solutions of PDEs. In addition, provides pseudo-transient continuation techniques for computing steady-state solutions.

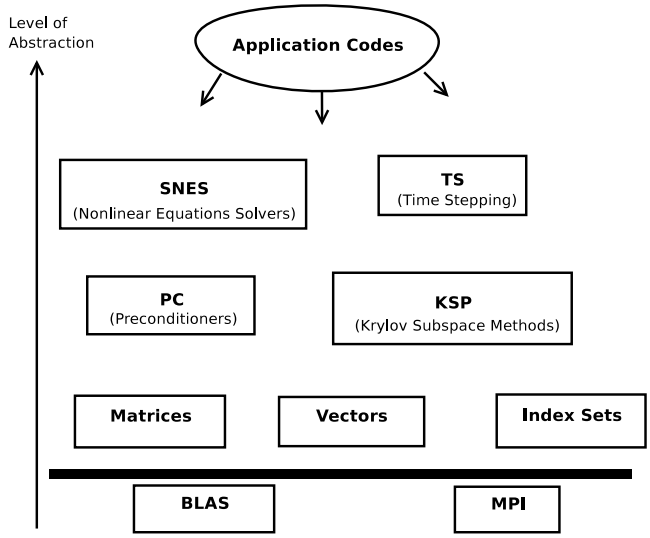

Figure 2.2: The hierarchical structure of the PETSc library, as described in the PETSc manual[3].

In figure 2.3 a closer look is taken at the parallel numerical parts of PETSc. As you can see PETSc offers a wide range of iterative Krylov Subspace solvers and preconditioners. Also the package offers the possibility to store system matrices in many different formats.

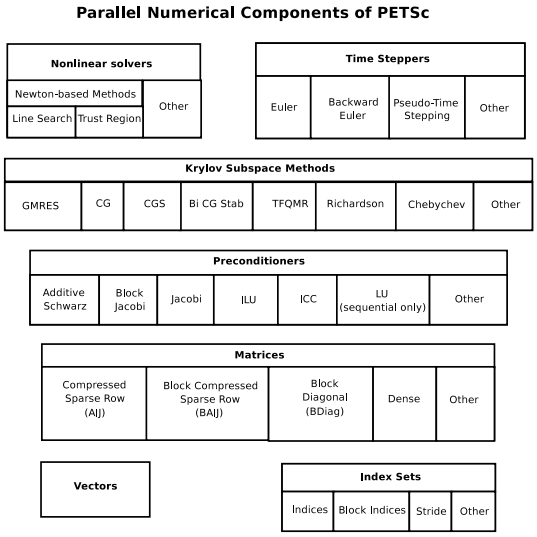

Figure 2.3: An overview of the numerical libraries of PETSc, as described in the PETSc manual[15].

2.2.1 Programming model

PETSc employs the distributed memory model. Each process has its own *address space* in memory and when needed, information is passed to other processes via MPI. To be more specific in e.g. a finite element problem, each process will own a contiguous subset of rows of the system matrix and will primarily work on this subset, sending/receiving information from/to other processes only when needed. How to perform this subdivision is of course a different subject, and for that reason PETSc has integrated support for reordering packages like ParMetis. If the already included packages does not include the needed features, external packages can easily be integrated. This programming model makes PETSc especially suitable for clusters.

2.2.2 Usage

Unlike OpenMP, PETSc is not easily integrated in existing codes. PETSc is a set of library interfaces, with its own matrix and vector structures, assembly and solving methods which are organized in an object oriented way. The user declares the system matrices and vectors, and then uses PETSc to assemble, precondition and solve the problem by envoking calls to PETSc methods. Through the whole process, there is ample possibility for the user to customize and control the solution process. It is also possible to supply most options at the command line, which makes experimenting between different methods very easy. An example of how PETSc can be used is shown in the next section.

2.2.3 Code examples

In section 1.4 of the PETSc manual [15] there are examples that illustrate the basic usage of PETSc. These examples are written in the C programming language, but using PETSc in Fortran code is not very different The linear system is solved using *KSP*, the interface to the preconditioners, Krylov subspace methods and direct linear solvers of PETSc.

2.3 Intel MKL

The Intel Math Kernel Library, MKL, provides Fortran routines and functions that perform operations on vectors and matrices including sparse matrices. It has support for both C and Fortran interfaces. It is highly optimized for Intel processors but also works well on other archtectures. Features include:

- Linear Algebra in the form of BLAS/LAPACK, scaLAPACK routines and sparse solvers.
- Fast Fourier Transforms.
- Vector math library.
- Vector random number generators.
- LINPACK benchmark packages.

These features covers most functions needed in a modern finite element solver. BLAS, FFT and vector math are threaded using OpenMP. Intel MKL also includes an efficient parallel solver, PARDISO, which is described in the next section.

2.3.1 The PARDISO solver

The PARDISO solver was originally developed by Olaf Schenk (et al) at the university of Basel. The solver has been licensed to Intel, whom in their turn have optimized it for Intel CPUs (though it still runs efficiently on e.g. AMD architectures). It is a high-performance, robust, memory efficient and easy to use software for solving large sparse symmetric and unsymmetric linear systems of equations on shared memory multiprocessors. Parallelization is achieved using OpenMP. The PARDISO solver uses a direct LU algorithm (of course handling multiple right hand sides), but it also implements an iterative conjugate gradient method. The behaviour of the latter is not studied in this project. As seen in figure 2.4 the solver supports a wide range of sparse matrices. It has both an out of- and in-core version. The out of core version writes the factorized matrices to temporary files on the hdd. The solver has four distinct phases:

1. Reordering and symbolic factorization.
2. Numeric factorization.
3. Back substitution.
4. Memory release.

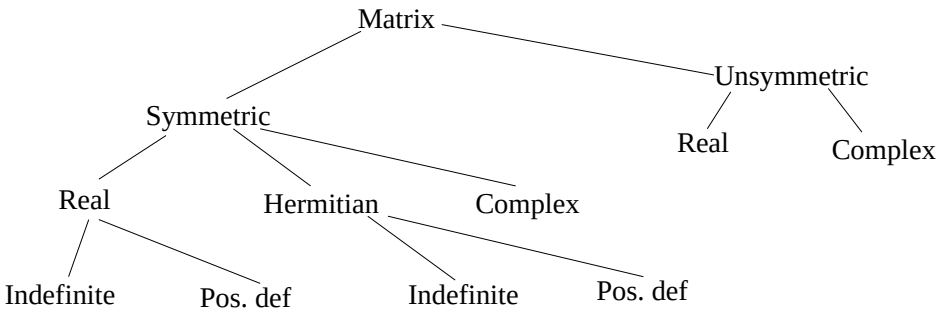

Figure 2.4: Sparse matrices supported by PARDISO, as described in the Intel MKL manual [4].

The reordering consists of a permutation vector being calculated, also, during the first phase, an elimination tree is calculated for the next phase. PARDISO uses supernodes with a two-level dynamic sheduling method to divide work between threads during the factorization phase. This method is quite advanced and is described by figure 2.5. The

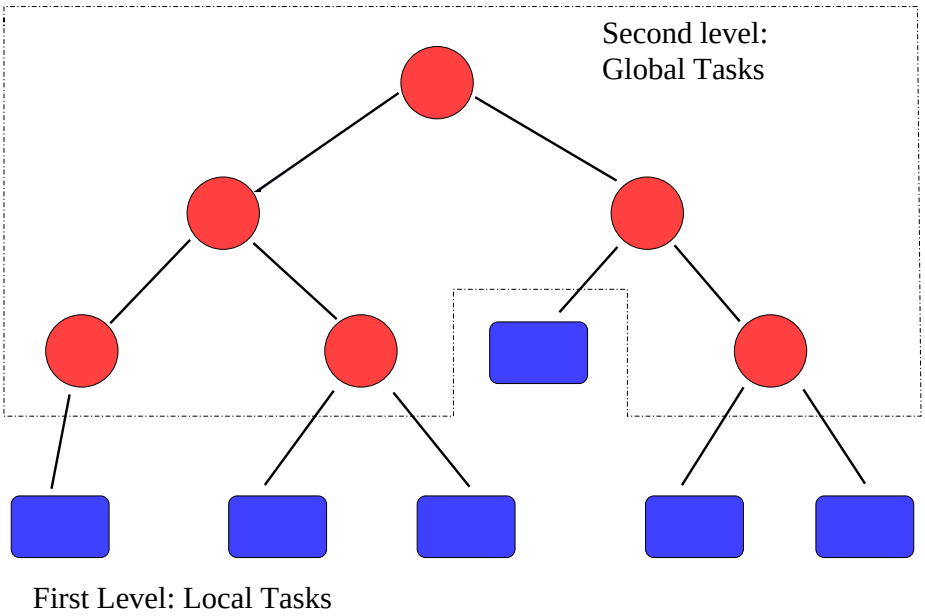

Figure 2.5: A mapping of a hypothetical elimination tree with the two-level scheduling over four processors as image described in the article [1] by Olaf Schenk and Klaus Gärtner.

idea is to have all processors working as much as possible, all the time. The first level consists of nodes which are independent, the second level is comprised of nodes which are not independent and therefore must be global to each process. The tasks in the first level is managed by a queue Q_s and in the second by a queue Q_r. Suppose now that one process is executing a task in the second level, and suppose furthermore that this process now has to wait for some other process to finish another task before finishing the work: In this situation, the CPU should not wait in an idle state, because it will inhibit the performance of the solver. In PARDISO, the process would in this situation put this node back in Q_r and pick another node from the queue and try factorizing this

instead. The pseudo-code for the factorization in PARDISO can be seen in [1]. This is an example of locks in OpenMP being used. The next phase of the PARDISO solution is back substitution, in this step PARDISO uses perturbed pivoting when encountering numerical troubles. Refining the solution during the back substitution to compensate for the "errors" in the factorized matrices. Information on the number of perturbations and iterative refinements can be queried using special functions. In the final phase of the solution structures allocated by PARDISO are freed, e.g. the factorized matrices.

2.3.2 Storage format

For sparse matrices, it is more efficient to store only the non-zero elements. This assumes that the sparsity is large i.e. the number of non-zeros is a small percentage of the total number of elements. There are a number of storage schemes for sparse matrices. The basic technique is to compress the non-zero elements into a linear array and provide arrays to describe the location of the non-zeros in the original matrix K.

The PARDISO solver uses the Compressed Sparse Row (CSR) format for sparse matrices. This is a row major format i.e. the the compression of K into a linear is done by walking across each row and store each non-zero element in the order they appear in the walk. For symmetric matrices, only non zero elements of the upper triangular half of the matrix are stored.

The CSR storage format consists of three arrays

values This array contains the non-zero elements of K. The elements are stored in the order they appear when walking across the rows in K.

ia element i of this array gives the index of the *values* array that contains the first non-zero element of row i in K. The number of non-zeros of the i-th row is equal to $ia(i+1) - ia(i)$, since the non-zeros are stored consecutively. An additional entry is added to the end of ia in order to have this relationship to hold for the last row in K. This makes the length of ia equal to the number of rows in $K + 1$.

ja element i of this array gives the number of the column corresponding to the element $values(i)$.

2.3.3 PARDISO usage

PARDISO is used by calling subroutines in Fortran or C. The Fortran interface of PARDISO is

```
SUBROUTINE pardiso(pt, maxfct, mnum, mtype, phase, n, a, ia, ja
    ,
                   perm, nrhs, iparm, msglvl, b, x, error)

INTEGER*8 pt(64)

integer maxfct, mnum, mtype, phase, n, nrhs, error, ia(*), ja
    (*),
```

```
perm(*), iparm(*)

real(8) a(*), b(n,nrhs), x(n,nrhs)
```

mtype This parameter defines the matrix type. The values of mtype supported by PARDISO are
1: real and structurally symmetric matrix 2: real and symmetric positive definite matrix -2: real and symmetric indefinite matrix 3: complex and structurally symmetric matrix 4: complex and Hermitian positive definite matrix -4: complex and Hermitian indefinite matrix 6: complex and symmetric matrix 11: real and unsymmetric matrix 13: complex and unsymmetric matrix

iparm Array of dimension 64 used to pass various parameters to PARDISO and to return some useful information after the execution of the solver. If $iparm(1) = 0$, then PARDISO fills $iparm$ with default values. There is no default value for $iparm(3)$, the number of processors to use, and this value has to be supplied by the user.

phase The execution phase of the solver.

n The number of equations.

a Array containing the non-zeros.

ia Array containing starting indices of each row.

ja Array containing the column indices for the corresponding value in a.

perm This array, of dimension n, holds the permutation vector. If A is the original matrix and $B = PAP^T$ the permuted matrix, then row i of A is the $perm(i)$ row of B. The permutation vector is only accessed if $iparm(5) = 1$.

nrhs The number of right-hand sides that need to be solved for.

msglvl If $msglvl = 0$ then PARDISO generates no output, if msglvl = 1 the solver prints statistical information.

b This array of dimension $(n, nrhs)$ contains the right hand side vector/matrix B. On output, it contains the solution if $iparm(6) = 1$.

x On output, it contains the solution if $iparm(6) = 0$.

This interface is given for 64-bit architectures. For 32-bit architectures, the argument pt(64) must be defined as INTEGER*4.

2.3.4 Code example

The code example in Listing 2.2 is taken from the MKL manual and shows an example of how to use the PARDISO solver. The program computes the solution of a sparse

symmetric linear system $Ax = b$ where

$$A = \begin{pmatrix} 7.0 & 0.0 & 1.0 & 0.0 & 0.0 & 2.0 & 7.0 & 0.0 \\ 0.0 & -4.0 & 8.0 & 0.0 & 2.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 8.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 5.0 \\ 0.0 & 0.0 & 0.0 & 7.0 & 0.0 & 0.0 & 9.0 & 0.0 \\ 0.0 & 2.0 & 0.0 & 0.0 & 5.0 & 1.0 & 5.0 & 0.0 \\ 2.0 & 0.0 & 0.0 & 0.0 & 1.0 & -1.0 & 0.0 & 5.0 \\ 7.0 & 0.0 & 0.0 & 9.0 & 5.0 & 0.0 & 11.0 & 0.0 \\ 0.0 & 0.0 & 5.0 & 0.0 & 0.0 & 5.0 & 0.0 & 5.0 \end{pmatrix}$$

and

$$b = \begin{pmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{pmatrix}$$

First, the system of equations is defined. In the next step the *iparm* array is initialized. The number of processors to be used is set to the number of processors on the machine for best performance. The solver has three execution steps. In the first step the solver performs a fill-reduction analysis and symbolic factorization. The *phase* parameter is set to 11 and symbolic factorization is performed. In the second step the solver performs the numerical factorization, with the *phase* parameter set to 22, the solver then performs only numerical factorization and no solve. In the last step the the system is solved. Finally the used memory is released.

Listing 2.2: mkl_exempel.f90

```
!
  ------------------------------------------------------------------------

! Example program to show the use of the "PARDISO" routine
! for symmetric linear systems
!
  ------------------------------------------------------------------------

! This program can be downloaded from the following site:
! http://www.computational.unibas.ch/cs/scicomp
!
! (C) Olaf Schenk, Department of Computer Science,
! University of Basel, Switzerland.
! Email: olaf.schenk@unibas.ch
!
!
  ------------------------------------------------------------------------
```

```
PROGRAM mkl_exempel

use omp_lib


IMPLICIT NONE
!.. Internal solver memory pointer for 64-bit architectures
!.. INTEGER*8 pt(64)
!.. Internal solver memory pointer for 32-bit architectures
!.. INTEGER*4 pt(64)
!.. This is OK in both cases
INTEGER*8 pt(64)
!.. All other variables
INTEGER maxfct, mnum, mtype, phase, n, nrhs, error, msglvl
INTEGER iparm(64)
INTEGER ia(9)
INTEGER ja(18)
REAL*8 a(18)
REAL*8 b(8)
REAL*8 x(8)

INTEGER i, idum
REAL*8 waltime1, waltime2, ddum
!integer omp_get_max_threads
!external omp_get_max_threads
!.. Fill all arrays containing matrix data.
DATA n /8/, nrhs /1/, maxfct /1/, mnum /1/
DATA ia /1,5,8,10,12,15,17,18,19/
ja = (/1, 3, 6,7,2,3, 5,3, 8,4, 7,5,6,7,6, 8,7,8/)
a = (/7.d0, 1.d0, 2.d0,7.d0,-4.d0,8.d0, 2.d0,1.d0,
     & 5.d0,7.d0, 9.d0,5.d0,1.d0,5.d0,-1.d0, 5.d0,11.d0,5.d0/)

!..
!.. Set up PARDISO control parameter
!..
do i = 1, 64
iparm(i) = 0
end do
iparm(1) = 1 ! no solver default
iparm(2) = 2 ! fill-in reordering from METIS

iparm(3) = omp_get_max_threads() ! numbers of processors,
                                 ! value of OMP_NUM_THREADS
iparm(4) = 0 ! no iterative-direct algorithm
iparm(5) = 0 ! no user fill-in reducing permutation
iparm(6) = 0 ! =0 solution on the first n compoments of x
iparm(7) = 16 ! default logical fortran unit number for output
iparm(8) = 9 ! numbers of iterative refinement steps
iparm(9) = 0 ! not in use
iparm(10) = 13 ! perturbe the pivot elements with 1E-13
iparm(11) = 1 ! use nonsymmetric permutation and scaling MPS
iparm(12) = 0 ! not in use
```

```
iparm(13) = 0 ! not in use
iparm(14) = 0 ! Output: number of perturbed pivots
iparm(15) = 0 ! not in use
iparm(16) = 0 ! not in use
iparm(17) = 0 ! not in use
iparm(18) = -1 ! Output: number of nonzeros in the factor LU
iparm(19) = -1 ! Output: Mflops for LU factorization
iparm(20) = 0 ! Output: Numbers of CG Iterations
error = 0 ! initialize error flag
msglvl = 0 ! don't print statistical information
mtype = -2 ! unsymmetric matrix symmetric, indefinite, no
    pivoting

!.. Initiliaze the internal solver memory pointer. This is only
! necessary for the FIRST call of the PARDISO solver.

do i = 1, 64
pt(i) = 0
end do

!.. Reordering and Symbolic Factorization, This step also
    allocates
! all memory that is necessary for the factorization

phase = 11 ! only reordering and symbolic factorization
CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
              &idum, nrhs, iparm, msglvl, ddum, ddum, error)
WRITE(*,*) 'Reordering completed ... '
IF (error .NE. 0) THEN

WRITE(*,*) 'The following ERROR was detected: ', error
STOP
END IF
WRITE(*,*) 'Number of nonzeros in factors = ',iparm(18)
WRITE(*,*) 'Number of factorization MFLOPS = ',iparm(19)

!.. Factorization.
phase = 22 ! only factorization

CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
              &idum, nrhs, iparm, msglvl, ddum, ddum, error)
WRITE(*,*) 'Factorization completed ... '
IF (error .NE. 0) THEN
WRITE(*,*) 'The following ERROR was detected: ', error
STOP
ENDIF

!.. Back substitution and iterative refinement
iparm(8) = 2 ! max numbers of iterative refinement steps
phase = 33 ! only factorization
do i = 1, n
b(i) = 1.d0
end do
```

```
CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
              &idum, nrhs, iparm, msglvl, b, x, error)
WRITE(*,*) 'Solve completed ... '
WRITE(*,*) 'The solution of the system is '
DO i = 1, n
WRITE(*,*) ' x(',i,') = ', x(i)
END DO

!.. Termination and release of memory
phase = -1 ! release internal memory
CALL pardiso (pt, maxfct, mnum, mtype, phase, n, ddum, idum,
              &idum, idum, nrhs, iparm, msglvl, ddum, ddum,
                  error)
END
```

Chapter 3

Parallel Solver for Plane Stress

This section describes the implementation of a parallel solver using the techniques described earlier. The three different approaches described earlier are evaluated with emphasis on the needs of a medium sized finite element software company. The simple 2D plane stress application used as the base for implemetation is also described, finally, the scaling, speed, and accuracy of the chosen method is evaluated in test examples.

3.1 Original Application

The implementation started with an existing application for 2D plane stress calculations using linear triangular elements. This application was comprised of a user interface written in Python, linked to a calculation engine written in Fortran. The calculation engine used a symmetric banded matrix format with a gaussian solver. The Triangle mesh generator [7] written by Jonathan Shewchuck was used to generate the finite element mesh in the text examples.

A dependency diagram of the different modules can be seen in figure 3.1 The different

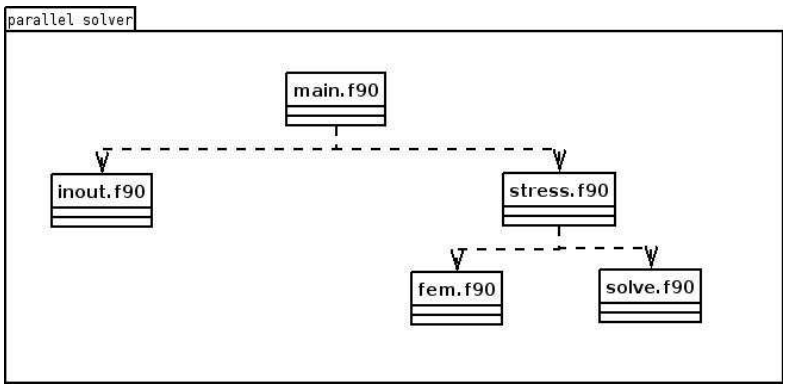

Figure 3.1: Dependency diagram for the original solver.

modules are:

inout read problem definition from input files.

fem element matrices, assembly functions, format conversion.

solve solve the problem, uses the Pardiso solver.

stress allocates system matrices, controls assembly process, calls the format conversion and calls the solve routine, finally it deallocates the system matrices.

main Allocate topology/coordinate matrices. Read problem definition via inout, perform calculations via stress.

The source code for this application can be found in appendix. In the following sections this code will be implemented as a parallel version using the techniques described in the initial chapters.

3.2 OpenMP Implementation

The first technique studied was OpenMP. For larger Finite Element problems, the most time consuming part is the solving process. It is therefore the part that this work focuses on. This section covers an OpenMP implementation of the existing serial Finite Element application.

3.2.1 Original Serial Solver

The solver that is used in the original application is a gaussian solver. It handles an arbitrary number of load cases and it is banded i.e. it expects the system matrix to be stored in a banded format. This format is used to conserve storage space in large Finite Element systems. Figure 3.2 illustrates the banded matrix format. The first column of

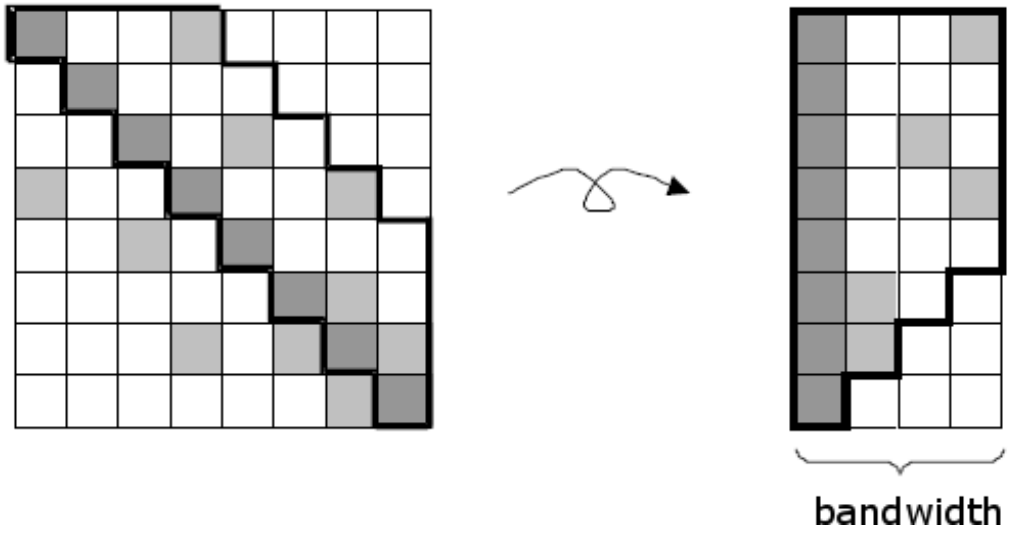

Figure 3.2: Illustration of the banded matrix format taken from [14].

the banded matrix contains the diagonal elements. Each row contains *bandwidth* elements starting from, and including, the diagonal element. The solver also expects the matrix to be symmetric as only the upper triangular part of the matrix is stored.

The subroutine starts with saving the right hand side vectors r, that was given on input, into x. The second loop takes care of the forward reduction of the matrix and finally the system is solved by back substitution.

3.2.2 Parallelizing the Solver using OpenMP

To parallize the existing code OpenMP directives will be added at suitable locations.

The first loop, saving the right hand side vectors, is very straightforward to parallelize with OpenMP.

```
!$omp parallel do
do n=1,nsize
   if(ifix(n).eq.0) then
   do ll=1,nlcs
        x(n,ll)=r(n,ll)
   end do
   endif
end do
!$omp end parallel do
```

The compiler directive !$*omp parallel do* is added at the beginning and !$*omp end parallel do* at the end of the loop to split it up in several threads. To parallelize the forward reduction of the matrix, however, is more complicated. Beginning with the first inner loop one can see that the value of kk is depending on the other iterations.

```
!$omp parallel do private(kk, jj)
do ii=n,nx
   kk=ii-n+1 !modified incrementation
   do jj=1,nlcs
        r(ii,jj)=r(ii,jj)-s(n,kk)*x(n,jj)
   end do
end do
!$omp end parallel do
```

To overcome this kk has to be labeled *private* which means that it is visible to one thread only. But now, when changes are seen by one thread only, the expression $kk = kk + 1$ will not be correct, so we change it to $kk = ii - n + 1$. This will work because ii and n are visible to all threads in the parallel region. Also jj is a local variable in the parallel region and also has to be labeled *private*. The second loop of the reduction part is not as complicated as the first one.

```
!$omp parallel do private(ilc)
do jeq=1,nsize
```

```
        if(ifix(jeq).eq.0) then
          do ilc=1,nlcs
            r(jeq,ilc)=x(jeq,ilc)
            x(jeq,ilc)=0.0d0
          end do
        endif
      end do
      !$omp end parallel do
```

ilc is a variable in the parallel region and is therefore labeled *private*.

3.2.3 Results with OpenMP

The OpenMP implementation was tested on a very simple model, consisting of a two dimensional rectangular plate divided into triangular elements by the Triangle mesh generator. The architecture was a 4 Intel Xeon 5160 @ 3.0 Ghz with 4 GB RAM. The original application was quite limited, only managing to calculate problems with up to approximately 5000 degrees of freedom before it began swapping on the hard drive. This is due to the very big bandwidth. When splitting up a region into several threads, the threads has to be forked and this causes an overhead in the program. This overhead is in fact so large that it in these relatively small problems are calculated even faster by the original solver. To overcome this problem, one could either develop an OpenMP parallelized solver from scratch instead of parallelizing extisting routines.

3.3 PARDISO Implementation

The next approach taken was to implement the finite element application using the Intel MKL solver PARDISO. First, the subroutine *bandsolve_omp* was replaced with the subroutine *pardiso_solve*, shown in appendix B in the *solve* module. This subroutine basically contains the code code in section 2.3.4, but with some modifications. The matrices in this case are real, symmetric and positive definite, thus the variable *mtype* is set to 2. Instead of initialize the number of equations, the CSR-representation of the system matrix, the solution matrix and the right-hand-side matrix in the solver code, these are given as parameters to *pardiso_solve* from the *execute* subroutine in the *stress* module.

Next, the code was adapted to the new solve routine.The banded matrices must be replaced with matrices stored in the CSR format, which is the default format used by the PARDISO solver. There are several ways to overcome this. Assemble directly into the CSR format or assemble into a different format and then do a conversion to CSR. Different techniques for this are described in the following sections.

3.3.1 Banded Solution

In this approach the system matrix was, as before, assembled into the banded format then converted to the CSR format in order to use the PARDISO solver. For this purpose a

conversion subroutine, BANDtoCSR, shown in appendix B, was written. It stores the banded system matrix K into the arrays $Kvec$, ia and ja. The subroutine simply iterates over each row and every non zero element encountered is stored in $Kvec$. The variable nnz is increased for every non zero element encountered. To take advantage of the symmetry, only the upper triangle part of the matrix is stored, and for this purpose the *col_idx* variable is used. It starts counting the column index from the diagonal on each row, and when a non zero element is encountered the value of *col_idx* is inserted at index nnz in ja. When the first non zero element of a row i is encountered, i.e. when the element at index i of ja is empty, this element is set to the value of nnz i.e. the index in K of the first non zero element of row i. Finally the dummy entry of ia is set to $nnz+1$. The code listing 3.1 shows the iteration.

Listing 3.1: Format conversion from banded to CSR

```
ia = 0
do i=1, nnd*2
        col_idx = i
        do j=1, bw
                if(K(i,j) .ne. 0) then
                        nnz = nnz+1 !increment non-zero
                             counter
                        Kvec(nnz) = K(i,j) !insert the
                            non-zero
                       !element
                        ja(nnz) = col_idx !the inserted
                             non-zeros
                      !column index
                        if (ia(i) .eq. 0) then
                                ia(i) = nnz
                        end if
                end if
                col_idx = col_idx+1
        end do
end do
ia(nnd*2+1) = nnz+1 !Add special last entry
```

The only part of the matrix relevant for the PARDISO solver is the submatrix with the elements corresponding to the degrees of freedom that are not prescribed. The subroutine $submatrixCSR$ performs an in-place extraction of this submatrix. It also calculates an vector, $updof$, containing the unprescribed degrees of freedom. The purpose of $updof$ is to extract the elements of the displacement vector and force vector that corresponds to the unprescribed degrees of freedom. $submatrixCSR$ is shown it's entirety in appendix B in the *fem* module. First the vector *nbr_del_col_vec* is calculated. Element i in *nbr_del_col_vec* tells how many columns from column 0 to $i-1$ that corresponds to a prescribed degree of freedom i.e. how many columns that will not be extracted to the submatrix. The column index of the submatrix can then be calculated as $column\ index\ of\ the\ original\ matrix - nbr\_del\_col\_vec(i)$. When *nbr_del_col_vec* is calculated the subroutine loops over all rows of K and performs the in-place extraction of the submatrix. The extraction code is shown in listing 3.2.

Listing 3.2: Submatrix extraction

```
nnz_sub = 0 ! initialize number of non-zeros in submatrix
```

```
allocate(nbr_del_col_vec(neq))
nbr_del_col_vec = 0
nbr_del_col = 0
do i=1, neq
        if (Pre(i) .ne. 0) then
                nbr_del_col = nbr_del_col + 1
        end if
        nbr_del_col_vec(i) = nbr_del_col
end do
do i=1, neq !loop over all rows of K
        if ( Pre(i) .eq. 0 ) then !this dof is unprescribed
                row = row+1 !the submatrix has a new row
                updof(row) = i !dof i is unprescribed
                tmp_rowstart = nnz_sub+1 !index where row in
                     !submatrix starts
                do j=1, ia(i+1)-ia(i)!loop over columns in this
                     row
                        col = ja( ia(i) + j - 1 )
                        if ( Pre( col ) .eq. 0 ) then
                                nnz_sub = nnz_sub +1
                                K(nnz_sub) = K( ia(i) + j-1 )
                                ja(nnz_sub)=col-nbr_del_col_vec
                                    (col)
                        end if
                end do
                ia(row) = tmp_rowstart
         end if
end do
ia(row+1) = nnz_sub+1 !Add special last entry
deallocate(nbr_del_col_vec)
```

At this point, when the system matrix was stored in the CSR format, one simply had to call the subroutine *pardiso_solve* to solve the system. The results are shown in the next section.

3.3.2 Results with Banded Solution

The code was tested on the same model as in the OpenMP section and used the same architecture. Even though this solution performed better than the OpenMP implementation, it was actually very limited. The program managed to solve problems with up to about 6000 degrees of freedom, but for problems with some ten thousands of degrees of freedom, it had to run for hours before returning a solution. The time consuming parts of the application was not the solver part, instead it was the assembling of the matrix, the format conversion and the extraction of the submatrix and force vector. Even though we used the memory efficient banded matrix format, the system matrix consumed a lot of memory. The main reason for this is that the bandwidth for the topology generated by Triangle is not very efficient. This resulted in a very memory consuming system matrix even for relatively small problems. In figure 3.3 the bandwidth for different problem sizes are shown. As the reader can see, the bandwidth for a system matrix with 6382 equations

was 6200 i.e. almost full bandwidth. This resulted in extremely inefficient calculations.

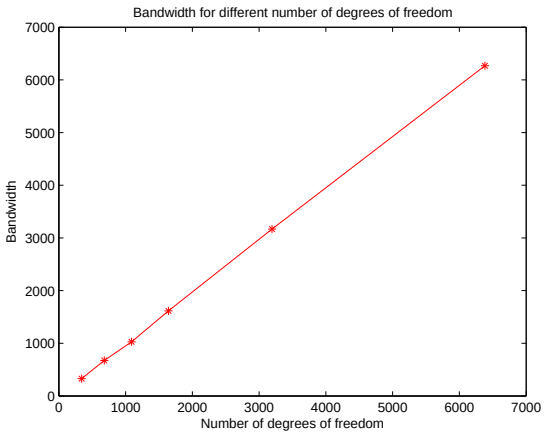

Figure 3.3: Bandwidths for different number of degrees of freedom.

The relatively bad bandwidth is due to the numbering scheme used by Triangle. When generating a mesh with Triangle, the geometry is defined in an input file. In this file, the edge nodes are listed. When the mesh is generated for the geometry, the numbering of the edge nodes are the same for all meshes generated i.e. they are forced into one position and this results in the relatively bad bandwidth. To overcome this, some kind of renumbering of the topology has to be done.

3.3.3 Renumbered Banded Solution

To reduce the bandwidth the *TRIANGULATION_RCM* [16] was used. It is an executable FORTRAN 90 program, using double precision arithmetic, which computes the reverse Cuthill-McKee(RCM, an algorithm for reordering nodes in a graph) reordering for nodes in a triangulation composed of 3-node or 6-node triangles. The user supplies a node file and a element file, containing the coordinates of the nodes, and the indices of the nodes that make up each triangle. Either 3-node or 6-node triangles may be used. The program reads the data, carries out the reordering algorithm and produces new node- and element files corresponding to the reordered nodes.

To utilize *TRIANGULATION_RCM*, a program that reads the node- and element files produced by Triangle and then produces new node- and element files to be read by *TRIANGULATION_RCM*, was written. Some changes was also made in the *inout* module of the application to make it read files in the new format. The bandwidth is now significantly reduced and more memory efficient. The results are shown in the next section.

3.3.4 Results with Renumbered Banded Solution

Using the renumbering algorithm, the bandwidth became approximately five times smaller. Of course, this resulted in more effective calculations because a more memory conserving system matrix was produced. The size of the system matrix is proportinal to $numberofelements*$

bandwidth, thus the application managed to solve problems approximately five times larger than with the previous one (i.e. approximately 60 000 degrees of freedom). Still, this was to limited because the size of the problems in real applications are up a million degrees of freedom. A better approach is needed.

3.3.5 Coordinate Format Solution

To store the system matrix in the banded format was not the optimal solution. Even when the nodes where renumbered to minimize the bandwidth, there where extremely many zeros kept in memory. The optimal solution would be to assemble directly into the CSR format or in a similar, memory conserving, matrix format and then convert to CSR. Because it complicated to efficiently assemble directly into the CSR format, latter alternative was chosen. The format that was chosen to assemble in, is the coordinate (COO) format, a matrix format similar to the CSR format and very easy to assemble into. Like the CSR format, it consists of three vectors:

K This array contains the non-zero elements of the system matrix A. The elements are stored in the order they appear when walking across the rows in A.

ik element i of this array gives the number of the row corresponding to the element $K(i)$.

jk element i of this array gives the number of the column corresponding to the element $K(i)$.

The assembly process is rewritten so that the element matrices are assembled directly into the COO format. To optimize the time complexity, the submatrix relevant for the PARDISO solver is directly calculated, instead of first calculating the full system matrix and then extracting the submatrix. As in section the *nbr_del_col_vec* vector is calculated to get the number of columns that will not be extracted. This vector is calculated by the subroutine *get_nbr_del_col_vec*, in the *fem* module, by looping through the *Pre* vector. This subroutine also calculates the *updof* vector, containing the unprescribed degrees of freedom, to extract the force- and displacement vectors relevant for the PARDISO solver. Furthermore the elements, corresponding to the prescribed degrees of freedom, of the force vector are also calculated by *assemCOO*. The code in listing 3.3 shows the calculation.

Listing 3.3: Calculation of get_nbr_del_col_vec

```
nbr_del_col_vec = 0

do i=1,size(Pre)
        if (Pre(i) .ne. 0) then
                nbr_del_col = nbr_del_col+1
        else
                updof_idx = updof_idx+1
                updof(updof_idx) = i
        end if
        nbr_del_col_vec(i) = nbr_del_col
end do
```

The assemblation of the system matrix in the COO format is done by the *assemCOO* subroutine shown in appendix B. The subroutine loops through the element matrix. For

every non zero element encountered, the nnz variable is increased. The global coordinates for element nnz are calculated with $edof(i), edof(j)$ where i, j are the coordinates in the local element matrix. If the matrix element corresponds to a prescribed variable, the vector element of the force vector is calculated. If not, nnz is increased and $Ke(i, j)$ is stored in $K(nnz)$. Finally, the global coordinates are stored in $ik(nnz)$ and $jk(nnz)$ respectively. The code, is very simple, shown in listing 3.4.

Listing 3.4: Assembly into the coordinate format

```
n = size(Ke,1)
row = 0

do i=1, n
        i_glob = edof(i)
        if(Pre(i_glob) .eq. 0) then
                row=row+1
                do j=1, n
                        j_glob = edof(j)
                        if(Ke(i,j) .ne. 0) then
                                if(Pre(j_glob) .ne. 0)
                                   then
                                        F(j_glob) = F(
                                           j_glob) -
                    & Ke(i,j)*u(j_glob)
                                else if(j_glob .ge.
                                   i_glob) then
                                        nnz=nnz+1
                                        K_sparse(nnz) =
                                            Ke(i,j)
                                        ik(nnz) =
                                           i_glob-
                & nbr_del_col_vec(i_glob)
                                        jk(nnz) =
                                           j_glob-
                & nbr_del_col_vec(j_glob)
                                end if
                        end if
                end do
        end if
end do
```

Now, when the system matrix is in the COO format, has to be converted to CSR in order to use PARDISO. For this we use subroutines from *SPARSKIT* - a tool package for working with sparse matrices written by Yousef Saad. The subroutine *coicsr*, performs an in place conversion from COO to CSR i.e. the vectors K, ia and ja are overwritten and contains the system matrix on the CSR format on output. In the assembly process, entries with the same coordinates in the global matrix were put after each. To remove these duplicate entries we use the subroutine *clncsr*, also from SPARSKIT.

3.3.6 Results with Coordinate Format Solution

To test the performance of this application the same simple rectangular model as in the previous sections was used. As before, Triangle was used to generate the triangular mesh and now, when the banded matrix format was not longer used, there was no need to utilize the renumbering algorithm for node renumbering anymore, so the Triangle mesh was directly used. The architecture used for testing was a four processor (AMD Opteron 2.4 GHz) Linux machine with 16 GB RAM. With this architecture the application managed problems with up to eleven million degrees of freedom before swapping on the hard drive. Different steps in the program was time measured and with this approach considerable performance improvement could be seen. In figure 3.4 the performance of the assembly routine *assemCOO* is shown and here one can see that this assembling is much faster than with the previous solutions. For eleven millions degrees of freedom the system matrix is assembled in less than a minute. Also, *assemCOO* takes care of the force vector and sub matrix extraction which results in a large improvement as the very time consuming subroutines *loadBANDmul* and *submatrixCOO* can be eliminated.

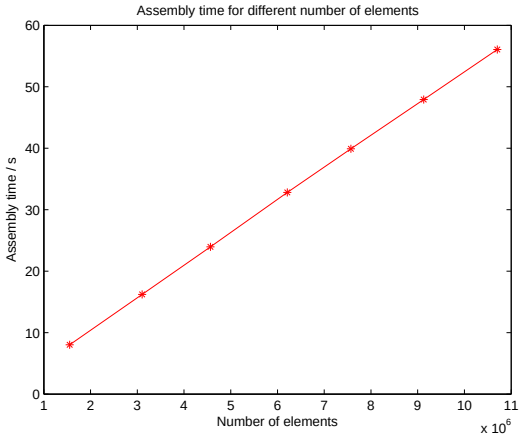

Figure 3.4: Assembly times for different number of degrees of freedom.

After the assembly process the matrix format has to be converted from COO to CSR. The conversion times with the current solution are shown in figure 3.5. For eleven million degrees of freedom the conversion was performed in less than three minutes so the overhead produced when not assembling directly in to the CSR format is relatively small.
The factorization time for different problem sizes are illustrated in figure 3.6. The largest problem was factorized in less than four minutes. The wallclock time of the *pardiso_solve* subroutine was measured and the results are shown in figure 3.7. The largest problem was solved in about seven minutes. To illustrate how well the solver scales on more than one processor, the performance in GFLOPS during the factorization phase for different number of processors are plotted. The result is shown in 3.8. One can see that the solver scales very well, it is almost linear for up to four processors.

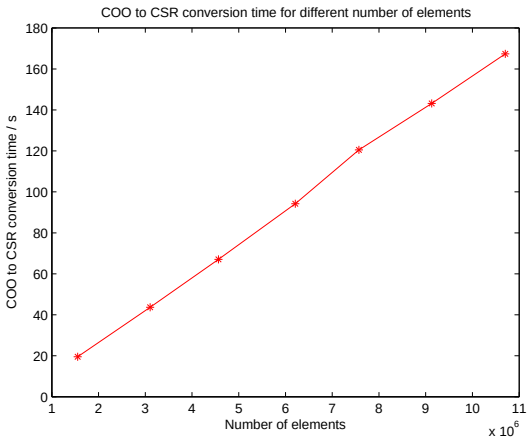

Figure 3.5: Conversion times for different number of degrees of freedom.

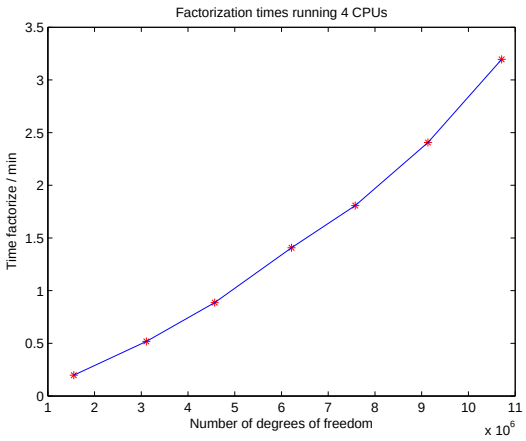

Figure 3.6: Factorization times for different number of degrees of freedom.

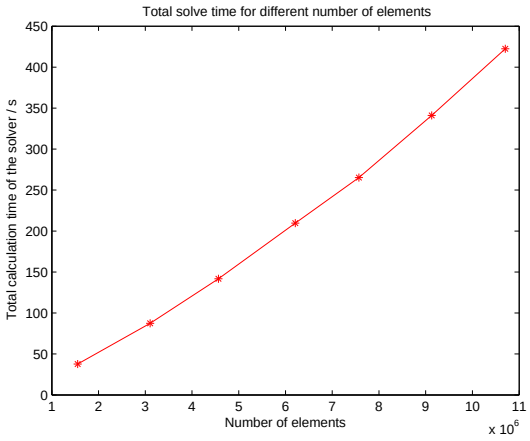

Figure 3.7: Total solve times for different number of degrees of freedom.

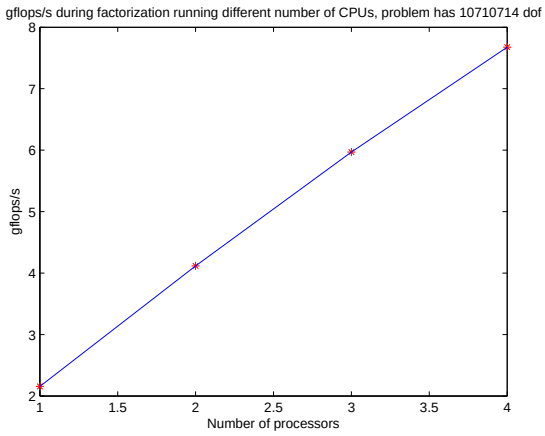

Figure 3.8: Performance in GFLOPS for different number of processors.

Chapter 4

Integrating the Parallel Solver into FEM-Design

To test the implemented solver it was integrated in the commercial finite element code FEM-Design. This chapter describes the implementation process and performance studies comparing the solver with the existing solver.

4.1 The Implementation

The FEM-Design solver is implemented in Fortran. The first approach tried was to integrate the solver directly into the FEM-Design code. The problem was that the FEM-Design solver is written in Fortran77 and the new solver is written in Fortran90. To make the FEM-Design code compile with the Intel Fortran90 compiler would take to much time not available in this project. For the temporary solution, the system matrix, assembled in the coordinate format, and the force vector was written from FEM-Design to file. A special application was written to handle the following steps:

FEM-data import The system matrix, the displacement vector and the force vector is read from the file produced by FEM-Design, and saved into arrays.

Matrix conversion The system matrix is converted from the COO format to the CSR format in order to use the solver.

Solve The new solver is called.

Result export The displacement vector is written to file.

The application was compiled as an executable and called from FEM-Design directly after the FEM-data was exported to file. The result was thereafter read from the file and was illustrated graphically by FEM-Design.

4.2 Results

The first step was to verify that the solver generated correct results. For this, some calculations was done on a small model with both the solver and with FEM-Design. The mesh of this model is shown in figure 4.1. The results were identical to the eighth decimal. The next step was to measure and compare the calculation times for larger problems.

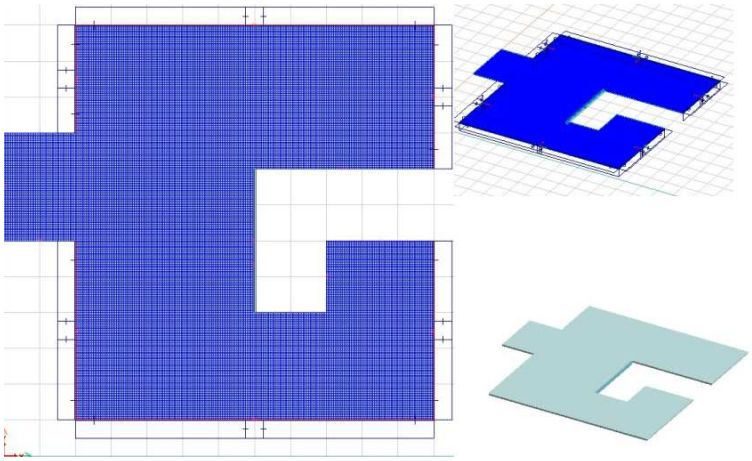

Figure 4.1: An illustration of the model used for result verification.

For this purpose a more complex model, shown in figure 4.2 was used. The structure was divided into three different meshes in order to test on different problem sizes. The testing architecture was Intel 2.4 GHz Dual Core machines with 2 GB RAM. In figure

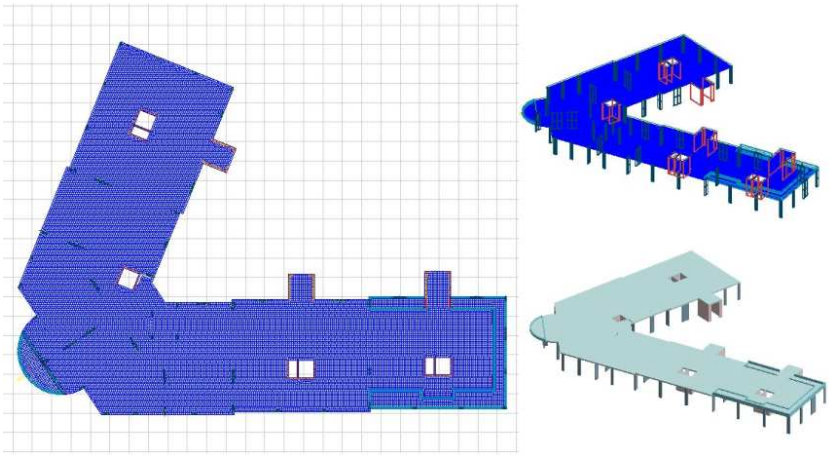

Figure 4.2: To the left the mesh is shown and to the right a three-dimensional illustration with and without mesh.

4.3 the solve times for FEM-Design are shown and in figure 4.4 the solve times for the parallel solver. The differences are noticeable, the largest example with 630000 equations was solved in about six hours with FEM-Design while it was solved in about one minute with the parallel solver i.e. a difference of almost a factor 360. One reason for this huge difference is that FEM-Design does a lot of reading and writing on the hard drive between

different execution steps. The new solver, on the other hand, keeps all information in memory throughout the entire calculation, which is much more time efficient. Also, the skyline matrix format used in FEM-Design is very memory consuming. A more efficient solution would be to assembly directly into a less memory consuming format e.g. the COO format.

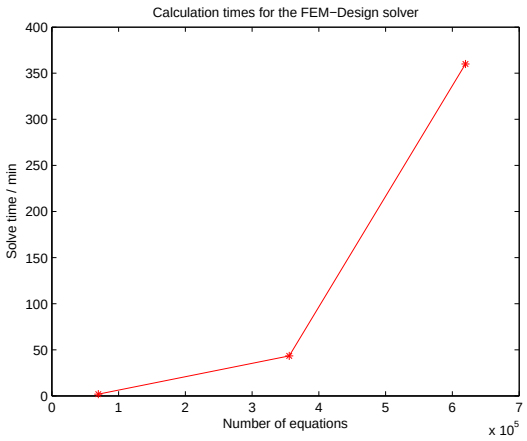

Figure 4.3: Calculation times for the FEM-Design solver.

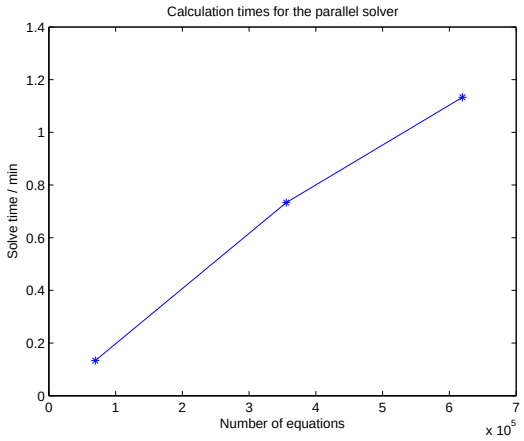

Figure 4.4: Calculation times for the parallel solver.

These problems where also calculated on a four processor (AMD Opteron 2.4 GHz) Linux machine with 16 GB RAM, to evaluate how well the parallel solver scaled on these kinds of problems. Figure 4.5 shows that it scales very well also for these problems, almost linear for up to four processors.

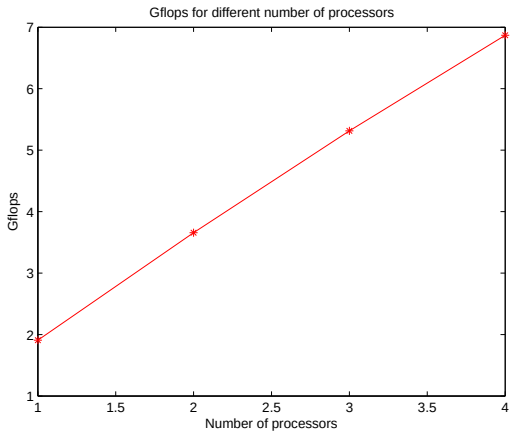

Figure 4.5: Scaling for the parallel solver, the performance is measured in GFLOPS.

Chapter 5

Distributed Programming Tools

When problems are too large for local computing resources, it is benificial to distribute calculation to non-local resources. A company could have an in-house dedicated server or small cluster for handling these computations. This frees the user of the program to perform other tasks whilst waiting for the computations to finish. Of course, this could be done for example by transferring the input files to a remote computer and invoking the solver via a terminal. This alternative is not so user friendly, especially when considering commercial software. It should be more convenient for the user to utilize distributed computations.

Figure 5.1 illustrates the concept of Remote Procedure Calls in distributed computing. Instead of one object invoking a method in another object locally, the same method could be invoked remotely on another machine. For example if FE modeling is being done on some low-performance machine, the time consuming solution of the system could be distributed to a better suited architecture via a Remote Procedure Call.

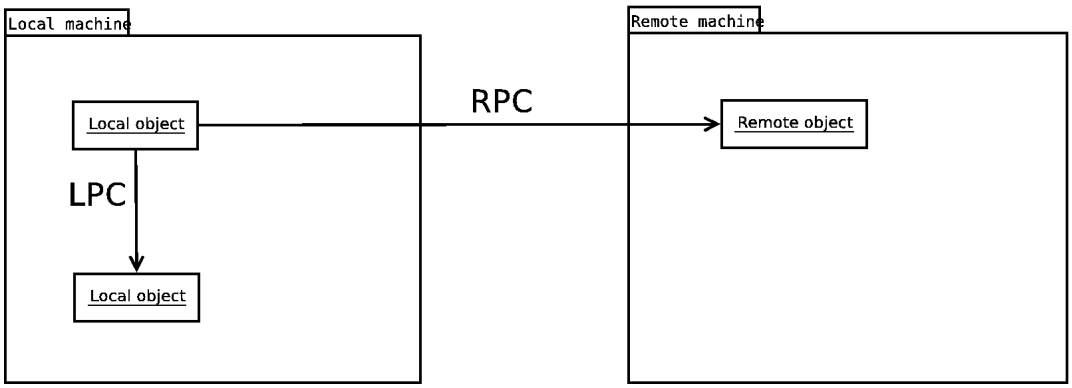

Figure 5.1: Remote Procedure Calls in distributed computing.

There exists many different programming tools, e.g. SOAP[9], Web Services[10], CORBA[8], Microsoft .NET[11], and Ice[12]. The preceeding tools can briefly be described as:

SOAP A protocol for exchanging XML-based messages over networks, normally using HTTP/HTTPS.

Web Services Basically, the term Web Service refers to clients and servers that communicate using XML messages that follow the SOAP standard, also there usually is a machine readable description of the operations supported by the server in the *Web Services Description Language* (WSDL).

CORBA Stands for Common Object Request Broker Architecture it is a vendor-independent architecture and infrastructure that computer applications use to work together over networks. Developed by the Object Management Group, OMG. Some of the people that developed CORBA are the ones that are now maintaining ICE. You can pretty much say that CORBA is the predecessor of Ice, although still maintained by OMG.

.NET Tools and librarys developed by Microsoft for connecting different applications developed for Microsoft platforms.

A disadvantage with .NET is that it only runs on Microsoft platform. This makes it unsuitable for use with Linux based resources. SOAP has very serious performance penalities in this type of applications, both in terms of network bandwidth and CPU overhead. Since our application will send large amounts of data, text based XML-based data exchange isn't efficient. The same applies to Web Services as it is based on the same technology. Another issue with Web Services is lack of standardization. A more in-depth review of these considerations can be found in [12].

Ice on the other hand, can be used in heterogenous enviroments, is efficient in network bandwith (sending data in binary form), memory use and CPU overhead. It is also easy to learn, has built in security and provides all features required for applications needing high-performance network transfers. Since this project focuses on transferring large amount of data over heterogenous systems, Ice was chosen for the distributed implementation. The next section will describe the Ice architecture in more detail.

5.1 Ice Overview

Ice is an object-oriented, middleware [1] providing tools, APIs and library support for building object oriented client-server applications. In contrast to e.g. Microsoft .NET, Ice can be used in heterogenous systems. This means that the client and server can run on different machine architectures or operating systems, be implemented in different programming languages, and also support a wide variety of networking technologies. Other advantages with using Ice include:

Synchronous/Asynchronous messaging For example, when running long calculations, the client doesn't have to wait for the results to be sent back from the server. Ice is inherently an asynchronous architecture but forces synchronous behaviour by default, it is a relatively easy task to change this.

Support for multiple interfaces There exists facilities to provide multiple implementations of client/server interfaces while retaining a single object identity across these interfaces. This makes it easy to write different kind of clients without making many changes, also, version handling is simplified.

[1] Middleware is computer software that connects software components or applications

Implementation independence Clients are not aware of how servers implement their objects.

Threading support Ice is fully threaded and APIs are thread-safe[2].

Transport independence Ice has support for both the TCP/IP and the UDP transport protocols. Also, there is support for SSL which means that information that has to be sent over "unsafe" networks can be encrypted. The Ice tool Glacier2 is used to handle firewalls.

Location and server transparency The user doesn't have to worry about locating objects and managing the underlying transport mechanisms. To the programmer, the client and server appear "connection-less".

Source code availibility The source code for Ice is fully available. However, use in commercial proprietary codes require a licensing agreement with ZeroC.

In this section, a brief overview of the Ice architecture is given. This is basically a sort of selection of the contents of sections 2 and 32 in [12]. Of course, far from everything can be covered in this report. However it is hoped that the description is clear enough that the reader can understand the later description of the actual implementation.

5.2 The Slice Definition Language

Ice provides a Remote Procedure Call, or RPC, protocol for invoking methods from client to server (or the other way around). This means that the client can cause a method-/procedure to be executed in a different address space (e.g. on a different computer on a local network) without the programmer explicitly handling all the complex details of this remote interaction. Of course, since Ice is supposed to be object oriented and language/platform independ, all these remote procedures has to be described in some object oriented, language independent way. This is where Slice comes in.

The Slice definition provides a contract by which the client and server must abide. Ice also provides special compilers that translate Slice definitions into language specific type definitions and APIs for a particular implementation language. These different translation algorithms are called *language mappings*. The defined types and APIs are then used by the devoloper to program specific functionality in either client or server. At the moment, Slice provides language mappings for C++, Java, C#, Visual Basic .NET, Python, Ruby, and PHP. Slice is a purely declarative language, because to be language independent, only interfaces and types can be described (not implementations). List 5.1 is an example of a small Slice definition file

Listing 5.1: Slice definition example

```
module Demo {
    interface Printer {
        void printString(string s);
    };
```

[2]A piece of code is thread-safe if it functions correctly during simultaneous execution by multiple threads

```
};
```

In this C++ example, the language mapping for a module is not surprisingly a C++ namespace and the interface within the module will be mapped to a C++ interface within the corresponding namespace. To generate these mappings, one has to invoke the **slice2cpp** compiler in the following way:

$ slice2cpp Printer.ice

This will generate two C++ source files:

Printer.h Header file with type definitions corresponding the Slice definitions for the *Printer* interface. In our case, this file will also include a definition of a pure virtual function corresponding to the printString method. This means that all implementations of the *Printer* interface must implement this method. Of course, if both client and server is written in C++, this header must be included in them both.

Printer.cpp Contains the source code for our Printer interface. This file also provides some basic tools for clients and servers e.g. code that marshals[3] on the client side and correspondingly unmarshals the data on the server side. For example, the string *s* passed from client to server has to be converted to a binary format on the client side, sent to the server side, and converted back in to the right format.

Figure 5.2 illustrates the situation when both client and server is written in C++. The client devoloper writes her client and the server developer writes her server, the only information these developers need for writing their end of the application is the Slice definition file. The final executables both use the C++ Ice run-time library and communicate via Remote Procedure Calls. In the final step the actual implementation of the client/server

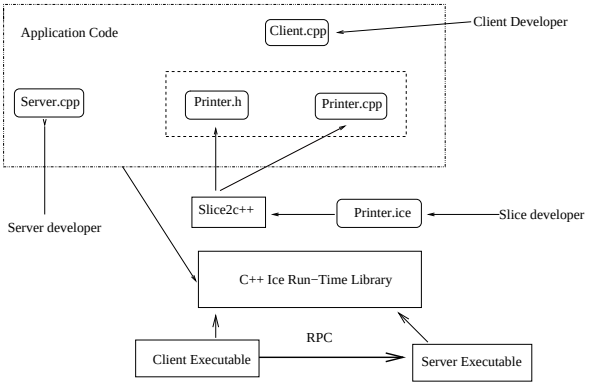

Figure 5.2: Development process if client and server share the same development environment. This process is also described in [12].

must be implemented. As Ice automatically handles all network communication, this is a relatively easy task. However, one needs a basic understanding of the principles of client/server communication in Ice, which is described in the next section

[3]the process of transforming the memory representation of an object to a data format suitable for storage or transmission. In Ice, the data is transmitted in a binary format

5.3 Principles of Ice Communication

Figure 5.3 illustrates the basic idea behind client/server communication in Ice. In this particular case, the are three servants instantiated on the server side- These are given the arbitrary names ”A”, ”B” and ”C”. The client holds a reference, or proxy, to the servant ”A” on the server side. Basically this means that ”A” is an incarnation of an implementation of a specific Slice defined interface and that the client can invoke remote procedure calls using this proxy that also contains information on how to access the server. In this case there also exists other servants but these are distinguished by their identities which have to be specified by the programmer (one can also us UUID to simplify the process if many servants are needed). Now, when the client invokes an operation using its ”A” proxy the following will happen on the server side; The object adapter which is listening to its specified network endpoints will receive the call from the client, look up the correct servant in its associated Active Servant Map (ASM) using the passed object identity and then dispatching the request to this servant. All data unmarshaling/marshaling is handled ”under the hood” so the user need not worry about this.

Figure 5.3 is missing one key element in Ice, namely the entity called communicator. This is the main entry point to the Ice run-time and is used both on the client and server side. The communicator handles the client/server side thread pools, configuration properties, object factories, Loggers, statistics, default router (e.g. used by Glacier2 for firewall handling), default locator (resolving object identity to proxy), plug-in manager and last but not least object adapters. For a full description of distributed programming with ICE, see [12].

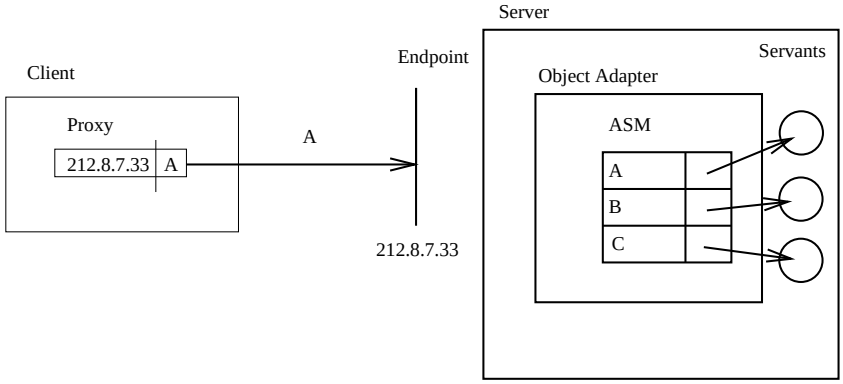

Figure 5.3: Binding a request to the correct servant in Ice using direct binding. A more detailed description can be found in [12].

Chapter 6

Distributed Interface for the Parallel Solver

This chapter describes the implementation of the distributed interface for the previously developed parallel solver.. The implementation is based on Ice [12] which is described earlier. The implemented system has been tested both locally on a Microsoft Windows system, remotely with the server running on a LUNARC [1] GNU/Linux node and the client running on a Windows computer. In the current implementation both client and server are implemented in C++. The server code however, links to a finite element solver implemented in Fortran. The approach for doing this is also described in this chapter

6.1 The Slice Definition

The Slice definition for the client/server application is given in list A.1 in appendix A. It is named *remoteComputation.ice*. As discussed earlier, when running the Slice compiler on this file, a header file *remoteComputation.h* and cpp file *remoteComputation.cpp* will be generated. The latter containing for example some code for marshalling/unmarshalling of data that the programmer need not be concerned about.

The first interface defined in the slice definition is a module called remoteComputation. This module will be mapped in to an equivalent namespace in C++. Everything else defined in this module will correspondingly be mapped to functions, classes and datastructures in the generated C++ namespace. Thereafter some Slice sequences are defined, these will be correspondingly mapped to C++ type definitions of std::vector. For example, *sequence<int>* will be mapped to *std::vector< ::Ice::Int>* in C++. Obviously, these type definitions describe our input/output vectors and matrices. These definitions look like the following:

```
sequence<int> intVec;
sequence<intVec> intMat;
sequence<double> doubleVec;
```

[1] Center for Scientific and Technical Computing in Lund

```
sequence<doubleVec> doubleMat;
```

Some exceptions are also defined in the Slice definition. *RangeError* is an exception thrown when there is some error in the solution. It is not implied that all solution errors are caused by range issues, a better name would have been *SolutionError*. *RequestCanceledException* is used in situations where client requests are canceled on the server side, for example if the server is forced to shut down. For example the RangeError exception is defined in the following way:

```
exception RangeError
{
};
```

Next, the interface for remote procedure calls is defined, this is done in the following way:

```
interface remoteExecution
 {

  ["ami", "amd"] void execute(doubleMat Coords,
    intMat Edof,
    doubleVec ElProp,
    int nnd,
    int nel,
    int nnl,
    int npv,
    intVec pvind,
    doubleVec pv,
    intVec loadind,
    doubleVec preloads,
    out doubleMat ElemForces,
    out doubleVec F,
    out doubleVec u) throws RangeError;
    };
```

An interface in Slice corresponds to an interface in C++, i.e a class with all member functions purely virtual. In this case, the pure virtual function is the *execute* method. This method has to be implemented by any servant implementing the remoteComputation interface. The preceding metadata ["ami", "amd"] has to do with asynchronous method invokation/dispatch and will be discussed later. The input/output parameters to the function will be mapped in a way that follows C++ coding standard. For example the first parameter will be mapped into the corresponding C++ parameter *const ::remoteComputation::doubleMat&* and the last into *::remoteComputation::doubleVec&*. The parameters are described as follows:

Coords The coordinate matrix.

Edof The topology matrix.

ElProp Element properties.

nnd The number of nodes in the mesh.

nel The number of elements.

nnl The number of prescribed nodal loads.

npv The number of prescribed variables.

pvind Indices of the prescribed variables.

pv Numerical values of prescribed variables. The actual indices of the variables are deferred from their corresponding entries in the *pvind* vector.

loadind Indices of prescribed loads.

preloads Numerical values of prescribed loads. This is the same construct as with the prescribed variables. The reason for this way of sending the data is minimizing overhead.

Elemforces The output element forces. This parameter is preceeded with the Slice keyword *out* which makes it possible to receive it as an output from the remote procedure call.

F The output force vector.

u The output displacement vector.

In case of a calculation error, the execute method can throw a RangeError. This is basically all that has to specified to be able to produce a simple client/server application for distributed computations. In the next section the client implementation is described.

6.2 Client Implementation

The implemented client is a simple console based application written in C++, where all communication with the server is handled by the Ice runtime. The client reads input data from text files and invokes the *execute* procedure (described above) remotely. Since the purpose of the client is not to provide a full finite element application but to illustrate how one could take advantage of distributed computations this is sufficient. A diagram describing the different classes in the client application can be seen in figure 6.1. The client depends on the Ice libraries. All information pertaining client/server communication, defined in the Slice definition is in the remoteComputation header so this also has to be included. Furthermore all reading from input files is done by the ElemData class. When invoking remote procedure calls on large problems, or when many clients are connected to the server, response times from the server could be very long. Now, Ice is inherently an asynchronous library but it forces synchronous behaviour by default. This means that when using the default Ice behaviour in these case the client will wait for a response from the server before continuing with program execution. This is not acceptable for any realistic finite element application. To overcome this problem, Ice provides tools for making asynchronous method invokations (AMI) where client execution continues after the remote procedure call.

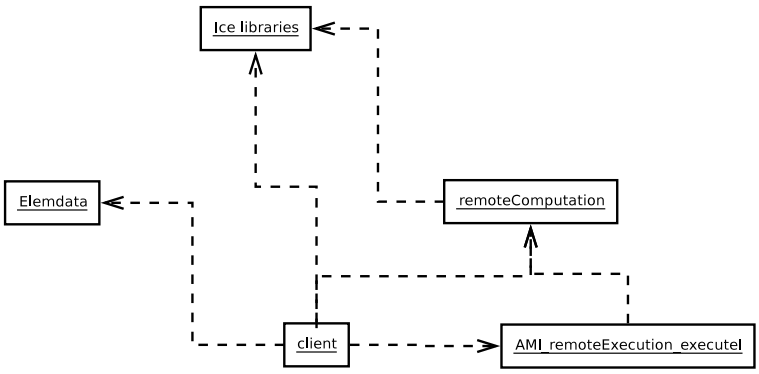

Figure 6.1: Dependency diagram for the client application.

This is the reason for the "ami" metadata proceeding the definition of the *execute* method in the Slice definition. This directive causes the compiler to generate code for asynchronous method invokation in the *remoteComputation* class. The user only has to create a callback object for the procedure call and specify what to do when receiving the results from the server (also error handling has to be specified). All this is done in the *AMI_remoteExecution_executeI* class which is an implementation of the *AMI_remoteExecution_execute* interface. This interface is in turn generated (in the *remoteComputation* header) by the Slice compiler when specifying the "ami" metadata directive for the *execute* function. In the next section the actual code written will be reviewed in detail.

6.2.1 Client Code

The code for the *ElemData* class is trivial and not relevant to distributed computations so therefore it is not discussed here. First the code for the main client program is presented, then we show how to handle the callback from the asynchronous method invokation in the *AMI_remoteExecution_executeI* class. The complete client code (except for *ElemData*) is included in appendix A.

Listing A.2 shows the code for the main client program. The client inherits from *Ice::application*, which is a helper class that Ice provides in order to eliminate writing the same boiler plate code over and over. Among other things, it initiates a communicator and handles interrupts from the user.

The client has two public and methods one private method. The first public method is the *run* method. This is needed by the main method defined in *Ice::application* and defines what is to happen when running the client. Basically the *run* method replaces the actual *main* method, but is contained inside another method handling some basic initializations and interrupt listening. The *main* method of the client is shown below.

```
int
main(int argc, char * argv[])
{
    AsyncClient app;
```

```
    return app.main(argc, argv);
}
```

As the reader can see the *main* method inherited from *Ice::Application* is called. This method in its turn calls the *run*method implemented by the derived application.

interruptCallback is an optional method that has to be defined if the statement

```
callbackOnInterrupt();
```

is specified in the *run* method. The purpose of this method is to override the default interrupt behaviour of *Ice::Application*.

The *run* method has to basic responsibilities:

1. Create a proxy to a *remoteExecution* servant.
2. Initiate the user menu and when prompted make the remote procedure call.

The servant creation is illustrated below:

```
Ice::ObjectPrx base = communicator()->stringToProxy(
        "SimplePrinter:default -p 10000");
remoteExecutionPrx re = remoteExecutionPrx::checkedCast(base);
if (!re)
 throw "Invalid proxy";

AMI_remoteExecution_executePtr cb =
    new AMI_remoteExecution_executeI;
```

After these lines, the variable *re* holds a ”remote object reference” to an instance of a class implementing the interface defined in the Slice definition. In this case the remote object is on the local computer (localhost) but could easily be changed. On the last line in the preceding code snippet a callback object is created for the asynchronous method invokation. Next, the input data structures are defined and read by calling methods in the *ElemData* class:

```
int nnd, nel, nnl, npv;
doubleMat Coords, ElemForces;
intMat Edof;
vector<int> pv_ind, load_ind;
vector<double> pv, pre_loads, F, u;
vector<double> ElProp;

::readProblemSize(nnd, nel, nnl, npv);
::readBCandElProp(nnl, npv, load_ind,
pre_loads, pv_ind, pv, ElProp);
::readNode(Coords, nnd);
::readEle(Edof, nel);
```

The remote procedure call is then executed in the following way:

```
re->execute_async(cb,Coords, Edof, ElProp,
nnd, nel, nnl, npv, pv_ind, pv,
load_ind, pre_loads);
```

The execution will immediately continue after this call and the callback object created will handle the response when it arrives. Last in the *run* method, error handling is done. As shown by the code examples implementing a Ice client is not very complicated.

To complete the asynchronous method implementation the server response callbacks must also be implemented. The full code for the *AMI_remoteExecution_executeI* can be found in listing A.3. There are two methods in this class; *ice_response* indicates that the operation is completed successfully and *ice_exception* indicates that a local or user exception was raised. The definition of the the *ice_response* method is given below.

```
virtual void ice_response(
    const ::remoteComputation::doubleMat& ElemForces,
    const ::remoteComputation::doubleVec& F,
    const ::remoteComputation::doubleVec& u)
{
    cout << "Finished Calculation" << endl;
    /*DO SOMETHING WITH THE RESULTS*/
}
```

The *ice_response* method receives the output parameters defined in the Slice definition as input parameters. In this method the results can be modified or put into an other data structure.

In the *ice_exception* method, the error is received as an input parameter, is printed.

```
virtual void ice_exception(
    const ::Ice::Exception& ex)
{
    try {
        ex.ice_throw();
    }
    catch(const Ice::LocalException& e) {
        cerr << "calculation failed " << e << endl;
    }
}
```

This completes the code for the asynchronous method invokation. Compared to other RPC libraries, Ice significantly reduces the amount of complexity and code needed to implement asynchronous code.

6.3 Server Implementation

The server is also implemented in C++ and does not involve more work than writing a client, however, there are some issues to consider. If all of the servers threads (one thread in our case) are busy dispatching long-running operations then no threads are available to process new requests. This may lead to long response times for client requests. Asynchronous Method Dispatch (AMD), the server side equivalent of AMI, addresses this scalability issue. When using AMD the server can suspend the processing of incoming requests in order to release the dispatch thread as soon as possible. In our case, this means queueing the request as a job in a workqueue and forking a new thread responsible for performing the tasks in the queue.

An aspect that can make implementing a server difficult, is linking C++ with Fortran. The solver is implemented in Fortran and Ice and Slice does not have a Fortran binding. When building the server in Visual Studio, linking was accomplished using a dynamic link library (DLL) built with the Intel Visual Fortran compiler. When building on GNU/Linux using the Intel C++ compiler, linking was done directly to object files built with the Intel Fortran compiler. There are also issues in the way that matrices are stored in C++ and Fortran. Fortran uses column-major order while C++ the row-major order. To solve this problem, and also to make the transition from *std::vector* to the arrays needed by Fortran, a modified version of the *FMatrix* [13] template class written by Carsten A. Arnholm is used.

Figure 6.2 shows the different classes in the server application. As shown in this figure, all

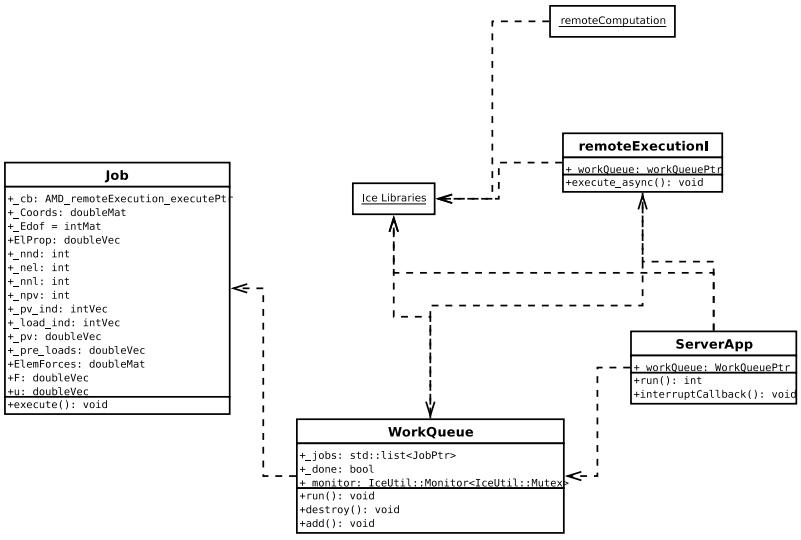

Figure 6.2: Dependency diagram for the server application. The diagram also shows public methods without input parameters for some classes.

the information pertaining a specific calculation is held within an object of the class *Job*. It also holds a callback object (the *_cb* variable) which is similar to the AMI case. The execute method in this class starts jobs and handles the callbacks. The *Job* class has a private method *solve_system* which converts the input matrices using the *FMatrix* class,

calls the Fortran routines for solving the system and finally converts the Fortran results to appropriate STL vectors.

The *WorkQueue* holds a list of pointers to Jobs. These are smart pointers defined by Ice. A smart pointer is a pointer that automatically frees the associated memory when the pointer goes out of scope or is deleted, eliminating many of the memory management errors in C++. *WorkQueue* inherits from *IceUtil::thread* and therefore implements a method, *run*, which is the method that is called by the Ice runtime when starting a new Workqueue thread. It also has a *destroy* method handling the destruction of the queue and an *add* method for adding jobs to the queue.

The server itself is defined in a very similar way as the client, inheriting from *Ice::application* and implementing a *run* and *interruptCallback* method. On the server side, the *remoteExecution* interface from the Slice definition, must be implemented. This is the responsibility of the *remoteExecutionI* class. The *execute_async* method only adds a job to the workqueue.

6.3.1 Server Code

In this section the most important parts of the server code is reviewed. All code with the exception of the *FMatrix* template class can be found in appendix A. Listing A.4 shows the server specific code, it is very similar to the client code, inheriting from *Ice::application*. To handle remote method calls a workqueue is first instantiated. An object adapter and a servant for the *remoteExecution* interface is created and the servant is added to the object adapters active servant map (ASM).

```
_workQueue = new WorkQueue();

Ice::ObjectAdapterPtr adapter
  = communicator()->createObjectAdapterWithEndpoints(
  "SimplePrinterAdapter", "default -p 10000");
  Ice::ObjectPtr object = new remoteExecutionI(_workQueue);
  adapter->add(object,
  communicator()->stringToIdentity("SimplePrinter"));
```

The workqueue thread is spawned which implies executing the code in its *run* method in a new thread.

```
_workQueue->start();
```

The adapter is activated, and the server will actively start listening for requests over the specific network interfaces.

```
adapter->activate();;
```

The server then waits for a shutdown request.

```
communicator()->waitForShutdown();
```

As the reader can se, writing this server is not very complicated. In the servant implementation (the implementation of the *remoteExecution* interface) the only thing that is done is calling the workqueues add method, adding a job to the workqueue.

The code for the workqueue is illustrated in listing A.5 and A.6. The code in the *run* method is illustrated below:

```
void
WorkQueue::run()
{
    IceUtil::Monitor<IceUtil::Mutex>::Lock lock(_monitor);

    while(!_done)
    {
        if(_jobs.size() == 0)
        {
            _monitor.wait();
        }

        if(_jobs.size() != 0)
        {
            JobPtr currentJob = _jobs.front();
            if(!_done)
            {
                currentJob->execute();
                _jobs.pop_front();
            }
        }
    }

  list<JobPtr>::const_iterator p;
  for(p = _jobs.begin(); p != _jobs.end(); ++p)
  {
        cout << "Canceled request" << endl;
    (*p)->notifyJobCancelled();
  }
}
```

The workqueue thread waits until there are jobs to process, pops the one in the front and calls its *execute* method. When the server is shutdown (implies *_done* is set to true) outstanding client requests are notified of cancellation.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

After reviewing the different approaches to parallelization, it was found that Intel MKL and the PARDISO solver was the most suitable alternative. The implemented parallel solver turned out to be very fast and scaled well on multiple processors. One problem with using the PARDISO solver is the required CSR sparse matrix format. After some experimentation with banded formats, a good solution was found in assembling in to the COO format and then converting to CSR.

The solver was compared with FEM-Designs original skyline solver. The study showed that the new PARDISO based solver was more efficient than the original solver. Besides the parallel computations, the differences also derive from the fact that FEM-Designs original solver uses a lot of disk IO even if there is memory available in the machine.

A client/server applications for distributed computations was implemented in C++ using Ice. It turned out to be very convenient and efficient to use Ice for these kinds of applications. A special template class was written in order to link to the Fortan solver. The application uses asynchronous method invokation/dispatch. On the client side, the user shouldn't have to wait for computations to finish before performing other tasks and the server has to be able to handle multiple client requests

7.2 Future Work

A future task is looking att some smart CSR assembly methods in order to eliminate the conversion overhead. This work focused mostly on Intel MKL for parallel computations. It is suggested that further testing of the PETSc libraries be done in the future. This is especially important when problems grow extremely large. Since PETSc uses a distributed memory model via MPI, it is more suitable for large clusters like the ones at LUNARC.

The parallel solver should be fully integrated in FEM-Design using the Intel Visual Fortran 10.0 Compiler. This required more work as the existing code had to be adopted for the new compiler which was not possible in the timeframe of this work. It would also be

interesting to do more work with the distributed computations. For example making a full integration with FEM-Design and testing other language mappings. Also, a more object oriented interface could be developed, allowing computations on many different problems.

Appendix A

Client/Server code

In this appendix the code for the client/server application is shown, some code is excluded because it is trivial and not relevant to the matter at hand.

A.1 Slice definition

Listing A.1: Slice definition

```
// ************************************************************
//
// Slice definition file for distributed computations
//
// Written by: Filip Johansson and Fredrik Hansson
//
//
// ************************************************************


module remoteComputation
{

sequence<int> intVec;
sequence<intVec> intMat;
sequence<double> doubleVec;
sequence<doubleVec> doubleMat;

exception RangeError
{
};

exception RequestCanceledException
{
};
```

```
interface remoteExecution
 {

  ["ami", "amd"] void execute(doubleMat Coords,
    intMat Edof,
    doubleVec ElProp,
    int nnd,
    int nel,
    int nnl,
    int npv,
    intVec pvind,
    doubleVec pv,
    intVec loadind,
    doubleVec preloads,
    out doubleMat ElemForces,
    out doubleVec F,
    out doubleVec u) throws RangeError;
    };

};
```

A.2 Client code

Listing A.2: Main client application

```
// ************************************************************
//
// Main client program. Uses asynchronous method invokation
//  for the distributed
// computations.
//
// Written by: Filip Johansson and Fredrik Hansson
//
// ************************************************************

#include <Ice/Ice.h>
#include <remoteComputation.h>
#include <AMI_remoteExecution_executeI.h>
#include <ElementData.h>

using namespace std;
using namespace remoteComputation;

class AsyncClient : public Ice::Application
{
    public:
        virtual int run(int,char*[]);
        virtual void interruptCallback(int);
    private:
        void menu();
};

int
```

```
main(int argc, char * argv[])
{
    AsyncClient app;
    return app.main(argc, argv);
}

int AsyncClient::run(int argc, char * argv[])
{
    callbackOnInterrupt();

    int status = 0;
    try {

        Ice::ObjectPrx base = communicator()->stringToProxy(
                              "SimplePrinter:default -p 10000");
        remoteExecutionPrx re = remoteExecutionPrx::checkedCast(base);
        if (!re)
            throw "Invalid proxy";

        AMI_remoteExecution_executePtr cb =
        new AMI_remoteExecution_executeI;


        int nnd, nel, nnl, npv;
        doubleMat Coords, ElemForces;
        intMat Edof;
        vector<int> pv_ind, load_ind;
        vector<double> pv, pre_loads, F, u;
        vector<double> ElProp;

        ::readProblemSize(nnd, nel, nnl, npv);
        ::readBCandElProp(nnl, npv, load_ind,
        pre_loads, pv_ind, pv, ElProp);
        ::readNode(Coords, nnd);
        ::readEle(Edof, nel);

        char c;
        menu();
        do
        {
            cout << "==>";
            cin >> c;
            if (c == 'c')
                re->execute_async(cb,Coords,
                Edof, ElProp, nnd,
                nel, nnl, npv, pv_ind, pv,
                load_ind, pre_loads);
            else if (c == 'x')
                interruptCallback(0);
             else
            {
                cout <<
                                "unknown command '"
                                << c << "'"
                                << endl;
                menu();
            }
        }
```

```
        while(cin.good());


    } catch (const Ice::Exception & ex) {
        cerr << ex << endl;
        status = 1;
    } catch (const char * msg) {
        cerr << msg << endl;
        status = 1;
    }
    return status;
}

void
AsyncClient::interruptCallback(int)
{
    try
    {
        communicator()->destroy();
    }
    catch(const IceUtil::Exception& ex)
    {
        cerr << appName() << ": " << ex << endl;
    }
    catch(...)
    {
        cerr << appName() << ": unknown exception" << endl;
    }
    exit(EXIT_SUCCESS);
}

void
AsyncClient::menu()
{
    cout <<
        "usage:\n"
        "c: calculate\n"
        "x: exit\n"
        ;
}
```

Listing A.3: Class for handling callback from asynchronous method invokation

```
#include "remoteComputation.h"

using namespace std;

class AMI_remoteExecution_executeI :
    public remoteComputation::AMI_remoteExecution_execute
{
    virtual void ice_response(
        const ::remoteComputation::doubleMat& ElemForces,
        const ::remoteComputation::doubleVec& F,
        const ::remoteComputation::doubleVec& u)
    {
        cout << "Finished Calculation" << endl;
```

```
        /*DO SOMETHING WITH THE RESULTS*/
    }

    virtual void ice_exception(
        const ::Ice::Exception& ex)
    {
        try {
            ex.ice_throw();
        }
        catch(const Ice::LocalException& e) {
            cerr << "calculation failed " << e << endl;
        }
    }
};
```

A.3 Server Code

Listing A.4: Main server application

```
// **********************************************************************
//
// Main server program for distributed computing application
//
// Written by: Filip Johansson and Fredrik Hansson
//
//
// **********************************************************************

#include <Ice/Ice.h>
#include <remoteComputation.h>
#include <WorkQueue.h>
#include <Job.h>
#include <Ice/Application.h>
#include <FMatrix.h>
#include <list>

typedef int      INTEGER;
typedef double    REAL;


using namespace std;
using namespace remoteComputation;


class ServerApp : virtual public Ice::Application {
public:
    virtual int run(int, char*[]) {

        callbackOnInterrupt();

        _workQueue = new WorkQueue(); //Inititate the work queue

        Ice::ObjectAdapterPtr adapter
```

```
            = communicator()->createObjectAdapterWithEndpoints(
                "SimplePrinterAdapter", "default -p 10000");
        Ice::ObjectPtr object = new remoteExecutionI(_workQueue);
        adapter->add(object,
                     communicator()->stringToIdentity("SimplePrinter"));

        _workQueue->start();
//spawn workqueue thread

        adapter->activate();
//adapter starts listening to network interface

        communicator()->waitForShutdown();

        if (interrupted()) {
            cerr << appName() <<
": received signal, shutting down" << endl;
        }
        adapter = 0;
        _workQueue->getThreadControl().join();
        return 0;
    }

    virtual void interruptCallback(int)
    {
    cout <<
"Received interrupt signal, terminating!" << endl;
    _workQueue->destroy();
    communicator()->shutdown();
    }
private:
    WorkQueuePtr _workQueue;
};

int main(int argc, char* argv[])
{
ServerApp app;
return app.main(argc, argv);
}
```

Listing A.5: The workqueues header.

```
// **********************************************************************
//
// Header file for class handling the work queue in the AMD model
//
// written by: Filip Johansson
//
// **********************************************************************

#ifndef WORK_QUEUE_H
#define WORK_QUEUE_H

#include <remoteComputation.h>
#include <IceUtil/Thread.h>
#include <IceUtil/Monitor.h>
```

```
#include <IceUtil/Mutex.h>
#include <Job.h>

#include <list>

class WorkQueue : public IceUtil::Thread
{
public:

    WorkQueue();

    virtual void run();

    void add(const remoteComputation::AMD_remoteExecution_executePtr & cb
        ,
        const doubleMat& Coords,
        const intMat& Edof,
        const doubleVec& ElProp,
        int nnd,
        int nel,
        int nnl,
        int npv,
        const intVec& pv_ind,
        const doubleVec& pv,
        const intVec& load_ind,
        const doubleVec& pre_loads);

    void destroy();

private:


    /*struct CallbackEntry
    {
        const remoteComputation::AMD_remoteExecution_executePtr cb;
        doubleMat Coords;
        intMat Edof;
        doubleVec ElProp;
        int nnd;
        int nel;
        int nnl;
        int npv;
        intVec pv_ind;
        doubleVec pv;
        intVec load_ind;
        doubleVec pre_loads;
        doubleMat ElemForces;
        doubleVec F;
        doubleVec u;
    };*/

    IceUtil::Monitor<IceUtil::Mutex> _monitor;
    std::list<JobPtr> _jobs;
    bool _done;
};

typedef IceUtil::Handle<WorkQueue> WorkQueuePtr;
```

```
#endif
```

Listing A.6: The workqueues implementation.

```
// ************************************************************************
//
//
//
// Written by: Filip Johansson
//
// ************************************************************************

#include <Ice/Ice.h>
#include <WorkQueue.h>
#include <FMatrix.h>

using namespace std;

WorkQueue::WorkQueue() :
    _done(false)
{
}

void
WorkQueue::run()
{
    IceUtil::Monitor<IceUtil::Mutex>::Lock lock(_monitor);

    while(!_done)
    {
        if(_jobs.size() == 0)
        {
            _monitor.wait();
        }

        if(_jobs.size() != 0)
        {
            //
            // Get next work item.
            //
            JobPtr currentJob = _jobs.front();

            if(!_done)
            {
                //
                // Execute calculation/send callback to client
                //
                currentJob->execute();

                //Job executed, remove from queue
                _jobs.pop_front();


            }
        }
    }
```

```
    //
    // Throw exception for any outstanding requests.
    //
    list<JobPtr>::const_iterator p;
    for(p = _jobs.begin(); p != _jobs.end(); ++p)
    {
        cout << "Canceled request" << endl;
        (*p)->notifyJobCancelled();
    }
}

void
WorkQueue::add(const remoteComputation::AMD_remoteExecution_executePtr &
    cb,
        const doubleMat& Coords,
        const intMat& Edof,
        const doubleVec& ElProp,
        int nnd,
        int nel,
        int nnl,
        int npv,
        const intVec& pv_ind,
        const doubleVec& pv,
        const intVec& load_ind,
        const doubleVec& pre_loads)
{
    IceUtil::Monitor<IceUtil::Mutex>::Lock lock(_monitor);

    if(!_done)
    {
        //
        // Add work item.
        //
        JobPtr newJob = new Job(cb,Coords,Edof,
        ElProp,nnd,nel,nnl,npv,
        pv_ind,pv,load_ind,pre_loads);

        if(_jobs.size() == 0)
        {
            _monitor.notify();
        }
        _jobs.push_back(newJob);

    }
    else
    {
        //
        // Destroyed, throw exception.
        //
        cout << "Destroyed!!!" << endl;
        cb->ice_exception(RequestCanceledException());
    }

}

void
WorkQueue::destroy()
```

```
{
    IceUtil::Monitor<IceUtil::Mutex>::Lock lock(_monitor);

    //
    // Set done flag and notify.
    //
    _done = true;
    _monitor.notify();
}
```

Listing A.7: Servant implementation

```
// **********************************************************************
//
// Implementation of remoteExecution servant.
//
// Written by: Filip Johansson and Fredrik Hansson
//
//
// **********************************************************************

#include <Ice/Ice.h>
#include <remoteComputation.h>
#include <WorkQueue.h>
#include <Job.h>
#include <Ice/Application.h>
#include <FMatrix.h>
#include <list>

typedef int      INTEGER;
typedef double    REAL;


using namespace std;
using namespace remoteComputation;


class remoteExecutionI : virtual public remoteExecution {
public:

    remoteExecutionI(const WorkQueuePtr&);


    virtual void execute_async(
    const remoteComputation::AMD_remoteExecution_executePtr & cb,
        const doubleMat& Coords,
        const intMat& Edof,
        const doubleVec& ElProp,
        int nnd,
        int nel,
        int nnl,
        int npv,
        const intVec& pv_ind,
        const doubleVec& pv,
        const intVec& load_ind,
        const doubleVec& pre_loads,
```

```
        const Ice::Current&);

private:
    WorkQueuePtr _workQueue;
};


remoteExecutionI::remoteExecutionI(const WorkQueuePtr& workQueue) :
    _workQueue(workQueue)
    {
    }

void remoteExecutionI::
execute_async(const remoteComputation::AMD_remoteExecution_executePtr &
    cb,
        const doubleMat& Coords,
        const intMat& Edof,
        const doubleVec& ElProp,
        int nnd,
        int nel,
        int nnl,
        int npv,
        const intVec& pv_ind,
        const doubleVec& pv,
        const intVec& load_ind,
        const doubleVec& pre_loads,
        const Ice::Current&)
{

            _workQueue->add(cb, Coords, Edof, ElProp, nnd,
                nel, nnl, npv, pv_ind, pv, load_ind, pre_loads);
}
```

Listing A.8: Header file for the *Job* class

```
#ifndef __Job_h__
#define __Job_h__

#include <remoteComputation.h>
#include <IceUtil/IceUtil.h>
using namespace remoteComputation;

class Job : public IceUtil::Shared {
public:
    Job(const AMD_remoteExecution_executePtr & cb,
        const doubleMat& Coords,
        const intMat& Edof,
        const doubleVec& ElProp,
        int nnd,
        int nel,
        int nnl,
        int npv,
        const intVec& pv_ind,
        const doubleVec& pv,
```

```
        const intVec& load_ind,
        const doubleVec& pre_loads);
    void execute();
    void notifyJobCancelled() {
        _cb->ice_exception(RequestCanceledException());
    }
private:
    bool solve_system();

    AMD_remoteExecution_executePtr _cb;
    doubleMat _Coords;
    intMat _Edof;
    doubleVec _ElProp;
    int _nnd, _nel, _nnl, _npv;
    intVec _pv_ind, _load_ind;
    doubleVec _pv, _pre_loads;
    doubleMat ElemForces;
    doubleVec F, u;
};
typedef IceUtil::Handle<Job> JobPtr;
#endif
```

Listing A.9: Implementation of the *Job* class

```
Job::Job(const AMD_remoteExecution_executePtr & cb,
        const doubleMat& Coords,
        const intMat& Edof,
        const doubleVec& ElProp,
        int nnd,
        int nel,
        int nnl,
        int npv,
        const intVec& pv_ind,
        const doubleVec& pv,
        const intVec& load_ind,
        const doubleVec& pre_loads) : _cb(cb),
        _Coords(Coords), _Edof(Edof), _ElProp(ElProp), _nnd(nnd),
        _nel(nel), _nnl(nnl), _npv(npv), _pv_ind(pv_ind),
        _pv(pv), _load_ind(load_ind), _pre_loads(pre_loads)
{
}
void Job::execute()
{

    if (!solve_system()) {
        _cb->ice_exception(RangeError());
        return;
        }
        _cb->ice_response(ElemForces, F, u);
}



extern "C" __declspec(dllimport) short
    __cdecl execute_fortran(
    REAL* Coords, INTEGER* Edof, REAL* ElProp, REAL* F,
```

```
    REAL* u, INTEGER* Pre,INTEGER* nnd, INTEGER* nel,
    INTEGER* npv, REAL* ElemForces);

bool Job::solve_system()
{

    FMATRIX<REAL> Coords_fortran(_Coords,_nnd,2);
    FMATRIX<INTEGER> Edof_fortran(_Edof,_nel,6);
    FMATRIX<REAL> ElProp_fortran(_ElProp,3,1);
    FMATRIX<REAL> ElemForces_fortran(_nel,3);

    FMATRIX<REAL> F_fortran(_nnd*2,1);
    F_fortran.initiate_values(_load_ind, _pre_loads);
    FMATRIX<REAL> u_fortran(_nnd*2,1);
    u_fortran.initiate_values(_pv_ind, _pv);
    FMATRIX<INTEGER> Pre_fortran(_nnd*2,1);
    Pre_fortran.initiate_values(_pv_ind);


    execute_fortran(Coords_fortran, Edof_fortran, ElProp_fortran,
    F_fortran, u_fortran, Pre_fortran, &_nnd,
    &_nel, &_npv, ElemForces_fortran);

    F_fortran.toSTL(F);
    u_fortran.toSTL(u);
    return true;

}
```

Appendix B

Parallel application code

In this appendix the code for the parallel application is shown. The application is divided into five modules.

B.1 Main program

Listing B.1: main.f90

```
program main

    use publicVars
    use inout
    use stress
    use fem
    use solve

    implicit none
    integer,parameter :: infile = 15, outfile = 16

    integer :: nnd, nel, nnl, npv, nl
    real(8), dimension(:,:), allocatable :: Coords, ElemForces
    real(8), dimension(:), allocatable :: F, u, ElProp
        ! Pre is the vector describing which node
        ! displacements that are prescribed
    integer, dimension(:,:), allocatable :: Edof
    integer, dimension(:), allocatable :: Pre
    character(60) geometryfile, bcelpropfile, outputfile
    integer :: Ktest(3,3), utest(3), ftest(3), K_sub(11), ia_sub(6),
       ja_sub(11), Pre_sub(5), nnz_sub
    integer :: i, testvec(2), updof(3)

    geometryfile = "geometry.txt"
    bcelpropfile = "bcelprop.txt"
    outputfile = "utdata.dat"

    call readproblemsize(nnd, nel, nnl, npv)

    allocate(Coords(nnd,2))
```

```
    allocate(Edof(nel,6))
    allocate(ElProp(3))
    allocate(F(nnd*2))
    allocate(u(nnd*2))
    allocate(Pre(nnd*2))
    allocate(ElemForces(nel,3))

    call readbcandelprop(F, u, Pre, ElProp, nnl, npv, nnd, bcelpropfile)

    call readgeometry(Coords, Edof, nnd, nel)

    call execute(Coords, Edof, ElProp, F, u, Pre,nnd,&
                 nel, npv, ElemForces)


    deallocate(Coords)
    deallocate(Edof)
    deallocate(ElProp)
    deallocate(F)
    deallocate(u)
    deallocate(Pre)
    deallocate(ElemForces)

end program main
```

B.2 File reading/writing

Listing B.2: inout.f90

```
module inout
use publicVars
contains
! Subroutine that parses the input file and produces the system matrices.

subroutine readproblemsize( nnd, nel, nnl, npv)

    integer :: nnd, nel, nnl, npv
    integer, parameter :: infile = 15

    open(unit=infile, file = 'meshing/triangle.1.node' ,access='
        sequential',&
         action='read',status='old')

    read(infile,*) nnd

    close(infile)

    open(unit=infile, file = 'meshing/triangle.1.ele' ,access='sequential
        ',&
         action='read',status='old')

    read(infile,*) nel
```

```
    close(infile)

    open(unit=infile, file = 'bcelprop.txt' ,access='sequential',&
        action='read',status='old')

    read(infile,*) nnl, npv

    close(infile)


end subroutine readproblemsize

subroutine readgeometry(Coords, Edof, nnd, nel)

    integer :: i, j, dummy, nodeNbrs(3), Edof(nel, 6)
    integer, parameter :: infile = 15
    real(8) :: Coords(nnd, 2)
    open(unit=infile, file = 'meshing/triangle.1.node' ,access='
        sequential',&
        action='read',status='old')

    read(infile,*)
    write(*,*) 'nnd:'
    write(*,*) nnd
    write(*,*) 'nel:'
    write(*,*) nel
    do j=1,nnd
        read(infile,*) dummy, (Coords(j,i),i=1,2)
    end do

    close(infile)

    open(unit=infile, file = 'meshing/triangle.1.ele' ,access='sequential
        ',&
        action='read',status='old')

    read(infile,*)

    do j=1, nel
        !write(*,*) j
        read(infile,*) dummy, (nodeNbrs(i),i=1,3)
        do i=1,3
            Edof(j,2*i-1) = 2*nodeNbrs(i)-1
            Edof(j,2*i) = 2*nodeNbrs(i)
        end do
    end do

    close(infile)

end subroutine readgeometry

subroutine readbcandelprop(F, u, Pre, ElProp, nnl, npv, nnd, inputfile)

    implicit none
    integer, parameter :: infile = 15
    integer :: nnl, npv, nnd, i, j, node, dof, Pre(nnd*2)
    real(8) :: F(nnd*2), u(nnd*2), ElProp(3)
    character *(*) inputfile
```

```
    open(unit=infile, file = inputfile ,access='sequential',&
        action='read',status='old')

    read(infile, *) nnl, npv

    F=0
    if (nnl > 0) then
        do j=1,nnl
            read(infile,*) node, dof, F(2*node+dof-2)
        end do
    end if

    Pre = 0;
    u = 0;
    do j=1,npv
        read(infile,*) node, dof, u(2*node+dof-2)
        Pre(2*node+dof-2) = 1
    end do

    read(infile,*) (ElProp(i), i=1,3)

end subroutine readbcandelprop

subroutine writeresult(F, u, Pre, nnd, nel, outputfile, ElemForces)

    integer, parameter :: outfile = 16
    real(8), dimension(nnd*2) :: F, u
    integer :: i, nnd, nel, Pre(nnd*2)
    character *(*) outputfile
    real(8), dimension(nel,3) :: ElemForces

    open(unit=outfile, file=outputfile, access='sequential', &
             action='write', status='unknown')


    write(outfile,*) 'Forces: '
    write(outfile,*) '-------------------------------------------'
    write(outfile,'(T2,A,T15,A,T32,A)') 'Node', 'Fx [N]', 'Fy [N]'
    write(outfile,*) '-------------------------------------------'
    do i=1,size(F,1)/2
        !if (Pre(i) == 1) then
            write(outfile,'(T2,I0,T10,2F15.8)') i,F(2*i-1),F(2*i)
        !end if
    end do
    write(outfile,*) '-------------------------------------------'
    write(outfile,*) ''
    write(outfile,*) ''
    write(outfile,*) 'Displacements: '
    write(outfile,*) '-------------------------------------------'
    write(outfile,'(T2,A,T10,A, T32, A)') 'Node', 'ux [m]', 'uy [m]'
    write(outfile,*) '-------------------------------------------'
    do i=1,size(u,1)/2
        !if (Pre(i) == 0) then
            write(outfile,'(T2,I0,T10,F10.6, T32, F10.6)') i,u(2*i-1), u
                (2*i)
        !end if
    end do
```

```
    write(outfile,*) '
       ------------------------------------------------------------------------------'

    write(outfile,*) ''
    write(outfile,*) ''
    write(outfile,*) 'Stresses: '
    write(outfile,*) '
       ------------------------------------------------------------------------------'

    write(outfile,'(T2,A,T17,A,T37,A, T57, A)') 'Element', &
             'sigmax [N/m2]', 'sigmay [N/m2]', 'sigmaxy [N/m2]'
    write(outfile,*) '
       ------------------------------------------------------------------------------'

    do i=1, size(ElemForces,1)
        write(outfile,'(T2,I0,T17,F17.7, T37, F17.7, T57, F17.7 )') i,
            ElemForces(i,1), &
                     ElemForces(i,2), ElemForces(i,3)
    end do
    write(outfile,*) '
       ------------------------------------------------------------------------------'

    close(outfile)
end subroutine writeresult

end module inout
```

B.3 FEM calculations

Listing B.3: fem.f90

```
module fem
! Module containg fe related methods like assembling and
! element methods
! also contains a function returning a systems bandwidth
use publicVars
contains

subroutine get_nbr_del_col_vec(nbr_del_col_vec, Pre, updof)

    integer :: i, nbr_del_col=0, updof_idx=0
    integer, dimension(:) :: nbr_del_col_vec, Pre, updof

    nbr_del_col_vec = 0

    do i=1,size(Pre)
        if (Pre(i) .ne. 0) then
            nbr_del_col = nbr_del_col+1
        else
            updof_idx = updof_idx+1
            updof(updof_idx) = i
        end if
        nbr_del_col_vec(i) = nbr_del_col
```

```
    end do

end subroutine get_nbr_del_col_vec

subroutine assemCOO(edof, Pre, u, F, K_sparse, Ke, nnz, ik, jk,
    nbr_del_col_vec )

    integer :: i, j, nnz, n, row, i_glob, j_glob
    integer, dimension(:) :: ik, jk, edof, Pre, nbr_del_col_vec
    real(8), dimension(:) :: K_sparse, u, F
    real(8), dimension(:,:) :: Ke

    n = size(Ke,1)
    row = 0

    do i=1, n
        i_glob = edof(i)
        if(Pre(i_glob) .eq. 0) then
            row=row+1
            do j=1, n
                j_glob = edof(j)
                if(Ke(i,j) .ne. 0) then
                    if(Pre(j_glob) .ne. 0) then
                        F(j_glob) = F(j_glob)-Ke(i,j)*u(j_glob)
                    else if(j_glob .ge. i_glob) then
                        nnz=nnz+1
                        K_sparse(nnz) = Ke(i,j)
                        ik(nnz) = i_glob-nbr_del_col_vec(i_glob)
                        jk(nnz) = j_glob-nbr_del_col_vec(j_glob)
                    end if
                end if
            end do
        end if
    end do

end subroutine assemCOO

subroutine COOtoCSR(K_sparse, ik, jk, nnz, neq)

    ! internal work arrays for sparskit2 methods
    integer,allocatable :: indu(:)
    integer,allocatable :: iwk(:)

    real(8), dimension(:) :: K_sparse
    integer, dimension(:) :: ik, jk
    integer :: nnz, neq

    allocate(indu(neq))
    allocate(iwk(neq+1))

!   write(*,*) 'innan coiscr'
    !----- Convert to compressed sparse row format -----
    call coicsr(neq,nnz,1,K_sparse,jk,ik,iwk)

!   write(*,*) 'innan clncsr'
    !----- Add duplicate elements and sort vectors -----
    call clncsr(3,1,neq,K_sparse,jk,ik,indu,iwk)
```

```
    nnz=ik(neq+1)-1

    deallocate(indu,iwk) ! deallocate internal work arrays

end subroutine COOtoCSR

      subroutine coicsr (n,nnz,job,a,ja,ia,iwk)
      integer ia(nnz),ja(nnz),iwk(n+1)
      real*8 a(*)
!-----------------------------------------------------------------------
! IN-PLACE coo-csr conversion routine.
!-----------------------------------------------------------------------
! this subroutine converts a matrix stored in coordinate format into
! the csr format. The conversion is done in place in that the arrays
! a,ja,ia of the result are overwritten onto the original arrays.
!-----------------------------------------------------------------------
! on entry:
!----------
! n = integer. row dimension of A.
! nnz   = integer. number of nonzero elements in A.
! job   = integer. Job indicator. when job=1, the real values in a are
!         filled. Otherwise a is not touched and the structure of the
!         array only (i.e. ja, ia)  is obtained.
! a = real array of size nnz (number of nonzero elements in A)
!         containing the nonzero elements
! ja    = integer array of length nnz containing the column positions
!     of the corresponding elements in a.
! ia    = integer array of length nnz containing the row positions
!     of the corresponding elements in a.
! iwk   = integer work array of length n+1
! on return:
!-----------
! a
! ja
! ia    = contains the compressed sparse row data structure for the
!         resulting matrix.
! Note:
!-------
!         the entries of the output matrix are not sorted (the column
!         indices in each are not in increasing order) use coocsr
!         if you want them sorted.
!----------------------------------------------------------------------c
!  Coded by Y. Saad, Sep. 26 1989                                      c
!----------------------------------------------------------------------c
      real*8 t,tnext
      logical values
!-----------------------------------------------------------------------
      values = (job .eq. 1)
! find pointer array for resulting matrix.
      do 35 i=1,n+1
         iwk(i) = 0
 35   continue
      do 4 k=1,nnz
         i = ia(k)
         iwk(i+1) = iwk(i+1)+1
```

```
 4     continue
!-----------------------------------------------------------------------
       iwk(1) = 1
       do 44 i=2,n
          iwk(i) = iwk(i-1) + iwk(i)
 44    continue
!
!      loop for a cycle in chasing process.
!
       init = 1
       k = 0
 5     if (values) t = a(init)
       i = ia(init)
       j = ja(init)
       ia(init) = -1
!-----------------------------------------------------------------------
 6     k = k+1
!      current row number is i.  determine  where to go.
       ipos = iwk(i)
!      save the chased element.
       if (values) tnext = a(ipos)
       inext = ia(ipos)
       jnext = ja(ipos)
!      then occupy its location.
       if (values) a(ipos)  = t
       ja(ipos) = j
!      update pointer information for next element to come in row i.
       iwk(i) = ipos+1
!      determine  next element to be chased,
       if (ia(ipos) .lt. 0) goto 65
       t = tnext
       i = inext
       j = jnext
       ia(ipos) = -1
       if (k .lt. nnz) goto 6
       goto 70
 65    init = init+1
       if (init .gt. nnz) goto 70
       if (ia(init) .lt. 0) goto 65
!      restart chasing --
       goto 5
 70    do 80 i=1,n
          ia(i+1) = iwk(i)
 80    continue
       ia(1) = 1
       return
!-------------------- end of coicsr ----------------------------------
!-----------------------------------------------------------------------
end subroutine coicsr

 subroutine clncsr(job,value2,nrow,a,ja,ia,indu,iwk)
!      .. Scalar Arguments ..
       integer job, nrow, value2
!      ..
!      .. Array Arguments ..
       integer ia(nrow+1),indu(nrow),iwk(nrow+1),ja(*)
       real*8  a(*)
!      ..
```

```
!
!     This routine performs two tasks to clean up a CSR matrix
!     -- remove duplicate/zero entries,
!     -- perform a partial ordering, new order lower triangular part,
!        main diagonal, upper triangular part.
!
!     On entry:
!
!     job   = options
!         0 -- nothing is done
!         1 -- eliminate duplicate entries, zero entries.
!         2 -- eliminate duplicate entries and perform partial ordering.
!         3 -- eliminate duplicate entries, sort the entries in the
!              increasing order of clumn indices.
!
!     value2  -- 0 the matrix is pattern only (a is not touched)
!                1 matrix has values too.
!     nrow    -- row dimension of the matrix
!     a,ja,ia -- input matrix in CSR format
!
!     On return:
!     a,ja,ia -- cleaned matrix
!     indu    -- pointers to the beginning of the upper triangular
!                portion if job > 1
!
!     Work space:
!     iwk     -- integer work space of size nrow+1
!
!     .. Local Scalars ..
      integer i,j,k,ko,ipos,kfirst,klast
      real*8  tmp
!     ..
!
      if (job.le.0) return
!
!     .. eliminate duplicate entries --
!     array INDU is used as marker for existing indices, it is also the
!     location of the entry.
!     IWK is used to stored the old IA array.
!     matrix is copied to squeeze out the space taken by the duplicated
!     entries.
!
      do 90 i = 1, nrow
         indu(i) = 0
         iwk(i) = ia(i)
 90   continue
      iwk(nrow+1) = ia(nrow+1)
      k = 1
      do 120 i = 1, nrow
         ia(i) = k
         ipos = iwk(i)
         klast = iwk(i+1)
 100     if (ipos.lt.klast) then
            j = ja(ipos)
            if (indu(j).eq.0) then
!     .. new entry ..
               if (value2.ne.0) then
                  if (a(ipos) .ne. 0.0D0) then
```

```
                  indu(j) = k
                  ja(k) = ja(ipos)
                  a(k) = a(ipos)
                  k = k + 1
               endif
            else
               indu(j) = k
               ja(k) = ja(ipos)
               k = k + 1
            endif
         else if (value2.ne.0) then
!     .. duplicate entry ..
            a(indu(j)) = a(indu(j)) + a(ipos)
         endif
         ipos = ipos + 1
         go to 100
      endif
!     .. remove marks before working on the next row ..
      do 110 ipos = ia(i), k - 1
         indu(ja(ipos)) = 0
 110  continue
 120  continue
      ia(nrow+1) = k
      if (job.le.1) return
!
!     .. partial ordering ..
!     split the matrix into strict upper/lower triangular
!     parts, INDU points to the the beginning of the upper part.
!
      do 140 i = 1, nrow
         klast = ia(i+1) - 1
         kfirst = ia(i)
 130     if (klast.gt.kfirst) then
            if (ja(klast).lt.i .and. ja(kfirst).ge.i) then
!     .. swap klast with kfirst ..
               j = ja(klast)
               ja(klast) = ja(kfirst)
               ja(kfirst) = j
               if (value2.ne.0) then
                  tmp = a(klast)
                  a(klast) = a(kfirst)
                  a(kfirst) = tmp
               endif
            endif
            if (ja(klast).ge.i) then
               klast = klast - 1
            end if
            if (ja(kfirst).lt.i) then
               kfirst = kfirst + 1
            end if
            go to 130
         endif
!
         if (ja(klast).lt.i) then
            indu(i) = klast + 1
         else
            indu(i) = klast
         endif
```

```
 140  continue
      if (job.le.2) return
!
!     .. order the entries according to column indices
!     burble-sort is used
!
      do 190 i = 1, nrow
         do 160 ipos = ia(i), indu(i)-1
            do 150 j = indu(i)-1, ipos+1, -1
               k = j - 1
               if (ja(k).gt.ja(j)) then
                  ko = ja(k)
                  ja(k) = ja(j)
                  ja(j) = ko
                  if (value2.ne.0) then
                     tmp = a(k)
                     a(k) = a(j)
                     a(j) = tmp
                  endif
               endif
 150        continue
 160     continue
         do 180 ipos = indu(i), ia(i+1)-1
            do 170 j = ia(i+1)-1, ipos+1, -1
               k = j - 1
               if (ja(k).gt.ja(j)) then
                  ko = ja(k)
                  ja(k) = ja(j)
                  ja(j) = ko
                  if (value2.ne.0) then
                     tmp = a(k)
                     a(k) = a(j)
                     a(j) = tmp
                  endif
               endif
 170        continue
 180     continue
 190  continue
      return
!---- end of clncsr -------------------------------------------------
!--------------------------------------------------------------------
end subroutine clncsr

subroutine submatrixCSR(neq, K, ia, ja, nnz_sub, Pre, updof)
    ! Performs an in-place extraction of the submatrix to be used for the
        pardiso
    ! solver, also calculates the vector updof containing the
        unprescribed
    ! degrees of freedom

    integer, dimension(:) :: Pre
    real(8), dimension(:) :: K
    integer, dimension(:) :: ia, ja, updof
    integer, dimension(:), allocatable :: nbr_del_col_vec
    integer :: i, neq, nnz, nnz_sub, row = 0, tmp_rowstart, nbr_del_col,
        col

    nnz_sub = 0 ! initialize number of non-zeros in submatrix
```

```
    allocate(nbr_del_col_vec(neq))
    nbr_del_col_vec = 0
    nbr_del_col = 0

    do i=1, neq
        if (Pre(i) .ne. 0) then
            nbr_del_col = nbr_del_col + 1
        end if
        nbr_del_col_vec(i) = nbr_del_col
    end do

    do i=1, neq !loop over all rows of K
        if ( Pre(i) .eq. 0 ) then !this dof is unprescribed
            row = row+1 !the submatrix has a new row
            updof(row) = i
            tmp_rowstart = nnz_sub+1 !index where row in submatrix starts
            do j=1, ia(i+1)-ia(i) !loop over columns in this row
                col = ja( ia(i) + j - 1 )
                if ( Pre( col ) .eq. 0 ) then !found a nonzero to be
                    added
                    nnz_sub = nnz_sub +1
                    K(nnz_sub) = K( ia(i) + j-1 ) !overwrite in to K
                    ja(nnz_sub) = col - nbr_del_col_vec( col )!overwrite
                        the column into the old vector
                end if
            end do
            ia(row) = tmp_rowstart
        end if
    end do

    ia(row+1) = nnz_sub+1 !Add special last entry

    deallocate(nbr_del_col_vec)


end subroutine submatrixCSR


 subroutine loadBANDmul(bw, neq, K, u, f)

    integer :: bw, neq, j, i
    real(8), dimension(:,:) :: K
    real(8), dimension(:) :: u, f

    do j=1,neq !loop through the displacement vector
        if (u(j) .ne. 0) then !if variable prescribed (not to zero)
            perform multiplication
            do i=1, neq
                if (i .gt. j .and. i-j .le. bw) then !below diagonal,
                    make symmetric shift
                    f(i) = f(i) - K(j,i-j+1)*u(j)
                else if ( i .le. j .and. j-i .le. bw) then  !above
                    diagonal
                    f(i) = f(i) -K(i,j-i+1)*u(j)
                end if
            end do
        end if
    end do
```

```
end subroutine loadBANDmul

subroutine elementMatrix(Ke, Edof, Coords, ElProp, elnum)

    !f2py intent(in, out) Ke

    integer :: elnum

    real(8) :: E, v, t, detC, x1, x2, x3, y1, y2, y3

    real(8), dimension(6,6) :: C, invC, Ke

    real(8), dimension(3,3) :: D

    real(8), dimension(3,6) :: Q

    real(8), dimension(:,:) :: Coords

    real(8) :: ElProp(:)

    integer, dimension(:,:) :: Edof
    E = ElProp(1)
    v = ElProp(2)
    t = ElProp(3)

    x1 = Coords(Edof(elnum,2)/2,1)
    x2 = Coords(Edof(elnum,4)/2,1)
    x3 = Coords(Edof(elnum,6)/2,1)
    y1 = Coords(Edof(elnum,2)/2,2)
    y2 = Coords(Edof(elnum,4)/2,2)
    y3 = Coords(Edof(elnum,6)/2,2)

    C(1,:) = (/  1.0_ap, x1, y1, 0.0_ap, 0.0_ap, 0.0_ap /)
    C(2,:) = (/  0.0_ap, 0.0_ap, 0.0_ap, 1.0_ap, x1, y1 /)
    C(3,:) = (/  1.0_ap, x2, y2, 0.0_ap, 0.0_ap, 0.0_ap /)
    C(4,:) = (/  0.0_ap, 0.0_ap, 0.0_ap, 1.0_ap, x2, y2 /)
    C(5,:) = (/  1.0_ap, x3, y3, 0.0_ap, 0.0_ap, 0.0_ap /)
    C(6,:) = (/  0.0_ap, 0.0_ap, 0.0_ap, 1.0_ap, x3, y3 /)

    D(1,:) = E/(1-v*v)*(/ 1.0_ap, v, 0.0_ap /)
    D(2,:) = E/(1-v*v)*(/ v, 1.0_ap, 0.0_ap /)
    D(3,:) = E/(1-v*v)*(/ 0.0_ap, 0.0_ap, (1-v)/2 /)

    Q(1,:) = (/ 0.0_ap, 1.0_ap, 0.0_ap, 0.0_ap, 0.0_ap, 0.0_ap /)
    Q(2,:) = (/ 0.0_ap, 0.0_ap, 0.0_ap, 0.0_ap, 0.0_ap, 1.0_ap /)
    Q(3,:) = (/ 0.0_ap, 0.0_ap, 1.0_ap, 0.0_ap, 1.0_ap, 0.0_ap /)

    detC = x1*(y2-y3) + x2*(y3-y1) + x3*(y1-y2)

    call matinv(6, C, invC)

    Ke = matmul(matmul(matmul(matmul(transpose(invC),&
            transpose(Q)),D),Q),invC)*0.5*detC*t

    return

end subroutine elementMatrix
```

```
integer function bandWidth(Edof)

    integer, dimension(:,:) :: Edof

    integer :: elemWidth(size(Edof,1))


    do i=1, size(Edof,1)
        elemWidth(i) = maxval(Edof(i,:)) - minval(Edof(i,:)) + 1
    end do

    bandWidth = maxval(elemWidth)

end function bandWidth

subroutine countnnz(K, bw, nnd, nnz)
    integer :: bw, nnz, nnd
    real(8), dimension(:,:) :: K

    nnz = 0
    do i=1, nnd*2
        do j=1,bw
            if (K(i,j) .ne. 0) then
                nnz = nnz + 1
            end if
        end do
    end do

end subroutine countnnz

subroutine BANDtoCSR(K, Kvec, ia, ja, bw, nnd)

    integer, dimension(:) :: ia, ja
    real(8), dimension(:,:) :: K
    real(8), dimension(:) :: Kvec
    integer :: col_idx, bw, nnz=0

    ia = 0

    do i=1, nnd*2
        col_idx = i
        do j=1, bw
            if(K(i,j) .ne. 0) then
                nnz = nnz +1 !increment non-zero counter
                Kvec(nnz) = K(i,j) !insert the non-zero element
                ja(nnz) = col_idx !the inserted non-zeros columnindex
                if (ia(i) .eq. 0) then
                    ia(i) = nnz
                end if
            end if
            col_idx = col_idx +1
        end do
    end do

    ia(nnd*2+1) = nnz+1 !Add special last entry

end subroutine BANDtoCSR
```

```
subroutine assem(Edof, Ke, elnum, K)

    !f2py intent(in, out) K

    integer, dimension(:,:) :: Edof
    real(8), dimension(:,:) :: Ke, K
    integer :: elnum, n
    integer, dimension(size(Edof,2)) :: eldof

    eldof = Edof(elnum,:)

    n = size(Ke,1)
    do j=1,n
        do i=j,n
            if (eldof(i) >= eldof(j)) then
                if ( abs(Ke(j,i)) .gt. 1e-12 ) then
                K(eldof(j),eldof(i)-eldof(j)+1) = &
                K(eldof(j),eldof(i)-eldof(j)+1) + Ke(j,i)
                end if
            else
                if ( abs(Ke(i,j)) .gt. 1e-12 ) then
                K(eldof(i),eldof(j)-eldof(i)+1) = &
                K(eldof(i),eldof(j)-eldof(i)+1) + Ke(j,i)
                end if
            end if
        end do
    end do


end subroutine assem

subroutine matinv(n,a,ai)

    !
    ! This subroutine inverts a matrix "a" and returns the inverse in "ai
       "
    ! n - Input by user, an integer specifying the size of the matrix to
        be inverted.
    ! a - Input by user, an n by n real array containing the matrix to
       be inverted.
    ! ai - Returned by subroutine, an n by n real array containing the
       inverted matrix.
    ! d - Work array, an n by 2n real array used by the subroutine.
    ! io - Work array, a 1-dimensional integer array of length n used by
       the subroutine.
    !
    !     From http://www.mae.usu.edu/faculty/wphillips/MAE6510.html
    !     Modified by Jonas Lindemann
    !

    implicit real (A-I,M-Z)

    integer :: n, i, j, m
    real(8) :: a(n,n)
    real(8) :: ai(n,n)

    integer, allocatable :: io(:)
```

```
real(8), allocatable :: d(:,:)

!!$omp parallel private(i,j,m,io,d)


allocate(io(n))
allocate(d(n,2*n))

!      Fill in the "io" and "d" matrix.
!      ********************************
do i=1,n
   io(i)=i
end do
do i=1,n
   do j=1,n
      d(i,j)=a(i,j)
      if(i.eq.j)then
         d(i,n+j)=1.0_8
      else
         d(i,n+j)=0.0_8
      endif
   end do
end do

!      Scaling
!      *******

do i=1,n
   m=1
   do k=2,n
      if(abs(d(i,k)).gt.abs(d(i,m)))m=k
   end do
   tmp=d(i,m)
   do k=1,2*n
      d(i,k)=d(i,k)/tmp
   end do
end do

!      Lower Elimination
!      *****************

do i=1,n-1

!         Pivoting
!         ********
   m=i
   do j=i+1,n
      if(abs(d(io(j),i)).gt.abs(d(io(m),i)))m=j
   end do
   itmp=io(m)
   io(m)=io(i)
   io(i)=itmp

!       Scale the Pivot element to unity
!       ********************************

   r=d(io(i),i)
   do k=1,2*n
```

```
            d(io(i),k)=d(io(i),k)/r
         end do

      !        ******************************

         do j=i+1,n
            r=d(io(j),i)
            do k=1,2*n
               d(io(j),k)=d(io(j),k)-r*d(io(i),k)
            end do
         end do
      end do

      !      Upper Elimination
      !      *****************

      r=d(io(n),n)
      do k=1,2*n
           d(io(n),k)=d(io(n),k)/r
      end do
      do i=n-1,1,-1
         do j=i+1,n
            r=d(io(i),j)
            do k=1,2*n
               d(io(i),k)=d(io(i),k)-r*d(io(j),k)
            end do
         end do
      end do

      !    Fill Out "ai" matrix
      !    ********************

      do i=1,n
         do j=1,n
            ai(i,j)=d(io(i),n+j)
         end do
      end do

      deallocate(io)
      deallocate(d)

      !!$omp end parallel

      return

end subroutine matinv
end module fem
```

B.4 Execution

Listing B.4: stress.f90

```
module stress
```

```
use publicVars
use fem
use solve
use inout

contains

subroutine execute(Coords, Edof, ElProp, F, u, Pre,nnd,&
        nel, npv, ElemForces)

        !f2py intent(in, out) F
        !f2py intent(in, out) u
        !f2py intent(in,out) ElemForces

        implicit none
        integer :: nnd, nel, bw, ierr, i, j, nnz, npv, neq
        real(8), dimension(:,:) :: Coords
        integer, dimension(:,:) :: Edof
        real(8), dimension(:,:) :: ElemForces
        real(8), dimension(:) :: F, u
      real(8), dimension(:), allocatable :: u_pardiso, F_pardiso
        integer, dimension(:) :: Pre
        real(8) :: start, end
      real(8) omp_get_wtime
      external omp_get_wtime

      integer, dimension(:), allocatable :: updof !vector containing the
          unprescribed variables
      integer :: nnz_sub
      real(8), dimension(:,:), allocatable :: K
      !Sparse system matrix storage
        real(8), dimension(:), allocatable :: Kvec
      integer, dimension(:), allocatable :: ia, ja
      !Element matrix
        real(8), dimension(6,6) :: Ke
      ! Element stresses and properties
        real(8) :: sigmae(3), ElProp(3)
      integer, dimension(:), allocatable :: nbr_del_col_vec, ik, jk
      real(8), dimension(:), allocatable :: K_sparse
      integer omp_get_thread_num
      external omp_get_thread_num

      neq = nnd*2

        allocate(nbr_del_col_vec(size(Pre)))
        allocate(K_sparse(nel*21))
      allocate(ik(nel*21))
      allocate(jk(nel*21))
      allocate(updof((nnd*2-npv)))

        call get_nbr_del_col_vec(nbr_del_col_vec, Pre, updof)
        nnz = 0

      start = omp_get_wtime()
        do i=1,nel
                call elementmatrix(Ke, Edof, Coords, ElProp, i)
              call assemCOO(Edof(i,:), Pre, u, F, K_sparse, Ke, nnz, ik,
                  jk, nbr_del_col_vec )
```

```
      end do
      end = omp_get_wtime()
      write(*,'(A,F)') 'assem time: ', end-start

      deallocate(nbr_del_col_vec)

      start = omp_get_wtime()
      call COOtoCSR(K_sparse, ik, jk, nnz, neq)
      end = omp_get_wtime()
        write(*,'(A,F)') 'cootocsr time: ', end-start

      allocate(F_pardiso(size(updof)))
        allocate(u_pardiso(nnd*2-npv))
      F_pardiso = F(updof)

        u_pardiso =0
        u=0
      start = omp_get_wtime()
      call pardiso_solve(neq-npv, K_sparse, ik, jk, u_pardiso, F_pardiso)
      end = omp_get_wtime()
      write(*,'(A,F)') 'pardiso time: ', end-start

      u(updof) = u_pardiso

      deallocate(K_sparse)
      deallocate(ik)
      deallocate(jk)
      deallocate(updof)
      deallocate(u_pardiso)
      deallocate(F_pardiso)

      !$omp parallel do default(private) shared(Edof,Coords,ElProp,u,
          ElemForces,nel)
      do i=1, nel
            call calcelementforces(Edof, Coords, ElProp, i, sigmae, u)
            ElemForces(i,:) = sigmae
      end do
      !$omp end parallel do

      do i=1, nel
                  do j=1,3
            write(*,*) ElemForces(i,j)
            end do
      end do


end subroutine execute
end module stress
```

B.5 System solving

Listing B.5: solve.f90

```
subroutine pardiso_solve(n, a, ia, ja, x, b)
    !.. Internal solver memory pointer for 64-bit architectures
    !.. INTEGER*8 pt(64)
    !.. Internal solver memory pointer for 32-bit architectures
    !.. INTEGER*4 pt(64)
    !.. This is OK in both cases
    INTEGER*8 pt(64)
    !integer pt(64)
    !.. All other variables
    integer :: maxfct, mnum, mtype, phase, n, nrhs, error, msglvl
    integer :: iparm(64)
    integer , dimension(:) :: ia, ja
    real(8), dimension(:) :: a, b, x
    integer :: i, idum
    real(8) :: waltime1, waltime2, ddum, mem
    integer omp_get_max_threads
    external omp_get_max_threads

    !.. Fill all arrays containing matrix data.
    nrhs = 1
    maxfct = 1
    mnum = 1

    !..
    !.. Set up PARDISO control parameter
    !..
    do i = 1, 64
    iparm(i) = 0
    end do
    iparm(1) = 1 ! no solver default
    iparm(2) = 2 ! fill-in reordering from METIS

    iparm(3) = omp_get_max_threads() !numbers of processors, value of
        OMP_NUM_THREADS
    iparm(4) = 0 ! no iterative-direct algorithm
    iparm(5) = 0 ! no user fill-in reducing permutation
    iparm(6) = 0 ! =0 solution on the first n compoments of x
    iparm(7) = 16 ! default logical fortran unit number for output
    iparm(8) = 9 ! numbers of iterative refinement steps
    iparm(9) = 0 ! not in use
    iparm(10) = 13 ! perturbe the pivot elements with 1E-13
    iparm(11) = 1 ! use nonsymmetric permutation and scaling MPS
    iparm(12) = 0 ! not in use
    iparm(13) = 0 ! not in use
    iparm(14) = 0 ! Output: number of perturbed pivots
    iparm(15) = 0 ! not in use
    iparm(16) = 0 ! not in use
    iparm(17) = 0 ! not in use
    iparm(18) = -1 ! Output: number of nonzeros in the factor LU
    iparm(19) = -1 ! Output: Mflops for LU factorization
    iparm(20) = 0 ! Output: Numbers of CG Iterations
    error = 0 ! initialize error flag
    msglvl = 1 ! print statistical information
    mtype = 2 ! symmetric positive definite matrix

    !.. Initiliaze the internal solver memory pointer. This is only
    ! necessary for the FIRST call of the PARDISO solver.
```

```
    do i = 1, 64
    pt(i) = 0
    end do

    !.. Reordering and Symbolic Factorization, This step also allocates
    ! all memory that is necessary for the factorization
    phase = 11 ! only reordering and symbolic factorization
    write(*,'(A,I,A)') 'Solving system on ', iparm(3), 'processors'
    CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja, idum,
        nrhs, iparm, msglvl, ddum, ddum, error)

    WRITE(*,*) 'Reordering completed ... '
    IF (error .NE. 0) THEN

    !WRITE(*,*) 'The following ERROR was detected: ', error
    STOP
    END IF
    !WRITE(*,*) 'Number of nonzeros in factors = ',iparm(18)
    !WRITE(*,*) 'Number of factorization MFLOPS = ',iparm(19)

    !.. Factorization.
    phase = 22 ! only factorization
    CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja, idum,
        nrhs, iparm, msglvl, ddum, ddum, error)

    !WRITE(*,*) 'Factorization completed ... '
    IF (error .NE. 0) THEN
    !WRITE(*,*) 'The following ERROR was detected: ', error
    STOP
    ENDIF

    !.. Back substitution and iterative refinement
    iparm(8) = 2 ! max numbers of iterative refinement steps
    phase = 33 ! only factorization
    CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja, idum,
        nrhs, iparm, msglvl, b, x, error)

    do i=1,10
        write(*,*) x(i)
    end do

    !WRITE(*,*) 'Solve completed ... '
    write(*,'(A,I)') 'iparm(7) = ', iparm(7)
    write(*,'(A,I)') 'iparm(8) = ', iparm(8)
    write(*,'(A,I)') 'iparm(14) = ', iparm(14)
    !.. Termination and release of memory
    phase = -1 ! release internal memory
    CALL pardiso (pt, maxfct, mnum, mtype, phase, n, ddum, idum, idum,
        idum, nrhs, iparm, msglvl, ddum, ddum, error)
end subroutine pardiso_solve


end module solve
```

Bibliography

[1] O. Schenk and K. Gärtner, *Two-level dynamic scheduling in PARDISO: Improved scalability on shared memory multiprocessing systems*, 2002

[2] O. Schenk and K. Gärtner, *Solving Unsymmetric Sparse Systems of Linear Equations with PARDISO*, Journal of Future Generation Computer Systems, 20(3):475–487, 2004.

[3] O. Schenk and K. Gärtner, *On fast factorization pivoting methods for symmetric indefinite systems*, Elec. Trans. Numer. Anal., 23:158–179, 2006.

[4] Intel. *Intel MKL 10.0 manual*

[5] OpenMP specification, *http://www.openmp.org/mp-documents/spec25.pdf*

[6] Miguel Hermanns, *Parallel Programming in Fortran 95 using OpenMP*, School of Aeronautical Engineering, Universidad Politécnica de Madrid, 2002

[7] Jonathan Richard Shewchuk, *Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator* in "Applied Computational Geometry: Towards Geometric Engineering" (Ming C. Lin and Dinesh Manocha, editors), volume 1148 of Lecture Notes in Computer Science, pages 203-222, Springer-Verlag, Berlin, May 1996. (From the First ACM Workshop on Applied Computational Geometry.)

[8] Henning, M., and S. Vinoski. 1999. *Advanced CORBA Programming with C++*. Reading, MA: Addison-Wesley.

[9] World Wide Web Consortium. 2002. *SOAP Version 1.2 Specification.* http://www.w3.org/2000/xp/Group/# soap12. Boston, MA: World Wide Web Consortium.

[10] World Wide Web Consortium. 2002. *Web Services Activity.* http://www.w3.org/2002/ws. Boston, MA: World Wide Web Consortium.

[11] Microsoft. 2002. *.NET Infrastructure & Services.* http://www.microsoft.com/technet/treeview /default.asp?url=/technet/itsolutio ns/net/Default.asp. Bellevue, WA: Microsoft.

[12] Henning, M. and Spruiell, M. et al. 2007. *Distributed Programming with Ice* ZeroC, Inc. http://www.zeroc.com.

[13] Carsten A. Arnholm. 1997 *Mixed language programming using C++ and FORTRAN 77* http://arnholm.org/software/index.htm

[14] Division of Structural Mechanics, LTH, *Software Development for Technical Applications*, http://www.byggmek.lth.se/utbildning/kurser/valfria/vsm032%2C_programutveckling_foer_tekniska_tillaempningar%2C_6_hp/

[15] Satish Balay and Kris Buschelman and William D. Gropp and Dinesh Kaushik and Matthew G. Knepley and Lois Curfman McInnes and Barry F. Smith and Hong Zhang. 2001 *http://www-unix.mcs.anl.gov/petsc/petsc-2/documentation/referencing.html*

[16] John Burkardt *TRIANGULATION_RCM Reverse Cuthill-McKee Node Reordering*, School of Computational Science, Florida State University. *http://people.scs.fsu.edu/ burkardt/f_src/triangulation_rcm/triangulation_rcm.html*