Analysis and Optimisation of a parallel finite element code

Olivier Goury

August 27, 2010

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2010
Abstract

The Finite Element Method is a well-known numerical method for solving all types of partial differential equations issued from various scientific fields. The accuracy of the solution can be increased by using powerful machines.

ParaFEM is a parallel finite element library, written in Fortran 90 and using MPI. In this project, we will investigate its performance and try some optimisations. Results show that there are 2 sections of the code taking about 90% of the time. Three different optimisations were tested: implementing a mixed code using OpenMP and MPI; a clever way of distributing the elements on the processors; introducing BLAS routines instead of intrinsic Fortran functions within the code. Experience shows that the mixed code can be beneficial in some bits of the code but not so efficient in some other parts; the BLAS routines have no special effects on the performance and the clever way of partitioning teaches us that minimising the area of contact surfaces between subdomains is sometimes not enough to get good performance, the number of neighbours that each subdomains has can be a critical point.
## Contents

1 Introduction 1

2 Background theory 3
   2.1 ParaFEM and the Finite element method (FEM) 3
      2.1.1 The finite element method 3
      2.1.2 ParaFEM 6
   2.2 METIS 8
   2.3 Machines 10
      2.3.1 HECToR Cray XT4 10
      2.3.2 HECToR Cray XT6 12

3 Porting 14
   3.1 Compiling 14
      3.1.1 The original code 14
      3.1.2 The mixed OpenMP/MPI version 15
   3.2 Running 15
      3.2.1 The original code 15
      3.2.2 The mixed OpenMP/MPI version 16
   3.3 Compiler optimisations 17
      3.3.1 Flags 17
      3.3.2 Results 17

4 Profiling and Testing 19
   4.1 Preliminary performance analysis 20
      4.1.1 Timings within the code 20
      4.1.2 Profiling tools 21
   4.2 Tests 26
      4.2.1 Scaling 26
      4.2.2 XT4 versus XT6 28
      4.2.3 Specific tests on HECToR Cray XT6 31

5 Optimisations and Results 35
   5.1 Use of BLAS routines 35
      5.1.1 Implementation and Results 35
      5.1.2 flop rate evaluation 38
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Flags to be used for use of OpenMP</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>Runtime with different compiler options</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Output from scalasca (truncated for clarity) with 120 processes and 125000 elements in the mesh</td>
<td>23</td>
</tr>
<tr>
<td>4.2</td>
<td>Calls obtained with gprof</td>
<td>25</td>
</tr>
<tr>
<td>4.3</td>
<td>Execution time, Speedup, and Efficiency of the code</td>
<td>26</td>
</tr>
<tr>
<td>4.4</td>
<td>Comparison of Runtime and Efficiency on HECToR XT4 and XT6, when increasing the number of cores</td>
<td>29</td>
</tr>
<tr>
<td>4.5</td>
<td>Ratio of runtimes between HECToR XT4 and XT6</td>
<td>31</td>
</tr>
<tr>
<td>5.1</td>
<td>Specific timings on the &quot;solve equations&quot; section. A million element mesh, 120 processors, stopped after 10 iterations</td>
<td>37</td>
</tr>
<tr>
<td>5.2</td>
<td>Flop rates of the original code and the BLAS version on HECToR Cray XT4 and XT6</td>
<td>39</td>
</tr>
<tr>
<td>5.3</td>
<td>Specific timings on the &quot;solve equations&quot; section. A million element mesh, 120 processors, stopped after 10 iterations</td>
<td>49</td>
</tr>
<tr>
<td>5.4</td>
<td>Communication information depending on the mesh partition method</td>
<td>50</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Shape functions over different types of elements 4
2.2 Shape functions 5
2.3 Example of an output of ParaFEM 7
2.4 Multilevel partitioning 9
2.5 Recursive Bisection 9
2.6 a Cray SeaStar chip. Taken from [9] 11
2.7 Architecture of HECToR Cray XT4 12
2.8 Architecture of an HECToR Cray XT6 node 13

4.1 A 20-node hexahedron 19
4.2 Percentage of the execution times for the small mesh and the larger one 21
4.3 Scalasca result presentation (Here in the case of a mesh of 125000 elements) 22
4.4 Percentage of runtime spent in MPI routines time and Computations with 60 and 120 processors 24
4.5 Speedup for p121 on a mesh of 2560 elements, involving 36720 equations 27
4.6 Speedup of different sections of p121 28
4.7 Comparison of the performance of p121 on HECToR XT4 and XT6 30
4.8 Scaling of p121 using a fixed number of cores, but gathering them differently 32
4.9 Execution time of different sections of p121 using always 5 nodes 33
4.10 Speedup of different sections of p121 using a fixed number of nodes 33

5.1 The conjugate gradient algorithm in ParaFEM 36
5.2 One MPI task per die and 6 threads spawned 40
5.3 One MPI task per node and 24 threads spawned 41
5.4 Runtime of the two main sections of the code with different threads configurations 43
5.5 Subdomains 1 and 5 obtained with the ParaFEM naive mesh decomposition in 10 pieces 46
5.6 Subdomains 1 and 15 obtained with the ParaFEM naive mesh decomposition in 48 pieces 46
5.7 Subdomains obtained with the METIS mesh decomposition in 10 and 48 47
5.8 Process followed to get a mesh input file to ParaFEM that is obtained using METIS 48
Acknowledgements

I would like to sincerely thank my supervisor, Dr. Alan Simpson, for his valuable suggestions and comments. His support was essential and extremely appreciated, especially towards the end of the dissertation period.

I also wish to thank Dr. Lee Margetts, for providing suggestions, help and being very reactive to my questions.

I finally wish to thank Dr. Pierre Kerfriden and Dr. Stephane Bordas for coming up with the subject of this dissertation.
Chapter 1

Introduction

The Finite Element Method is a widely used numerical method for solving all types of scientific problems issued from different fields such as computational fluid dynamics, nuclear engineering or material sciences. The accuracy of this method is highly dependent on the quality of the mesh used, and the more refined the mesh is, the longer the computation time is. In 3-dimensional problems, with a reasonably refined mesh, the number of mesh elements can get very large. Using the power of supercomputers is hence a key to be able to run simulations with better accuracy.

In this project, we will investigate the performance of ParaFEM, which is a parallel finite element library. ParaFEM is written in Fortran 90/95 and uses message passing (MPI). The ParaFEM package includes quite a few finite element programs calling the parallel library. In this project we will restrict ourselves to analyse a specific program solving the three-dimensional strain of an elastic solid. Benchmarking and profiling will be done to get insights of the performance of the code. Some different kinds of optimisations will then be tested. Simulations will be run first on the United Kingdom National Supercomputing Service, also known as HECToR. There are 3 main optimisations that will be tested:

- the first one will be replacing the intrinsic fortran matrix multiplication function by their equivalent BLAS\(^1\) subroutines
- the other one will be to use the new possibilities of the HECToR Cray XT6 by implementing a mixed OpenMP-MPI version of the code.
- the last one will be to make a better partition of mesh elements given to each processor using the METIS mesh partitioner.

Chapter 2 first explains the basic concepts of the finite element method. It then describes ParaFEM and its strategies to solve finite element problems in parallel. A description of METIS \(^2\), which is the mesh partitioning tool used is given. The architecture of the HECToR Cray XT4 and XT6 is also explained. Chapter 3 talks about everything

\(^1\)Basic Linear Algebra Subroutines

\(^2\)Mesh partitioning tool
relevant to port the codes onto the HECToR machines and how the runs were made. Chapter 4 describes a series of test done on ParaFEM, presents its scaling on the XT4 and the XT6 machine. Profiling of the code is also presented. In Chapter 5, the different kind of optimisations attempted on ParaFEM are explained and tested. Chapter 6 will sum up the results and suggests some possible future work.
Chapter 2

Background theory

This chapter gives background information relevant to well understand the project.

2.1 ParaFEM and the Finite element method (FEM)

2.1.1 The finite element method

The finite element method is a numerical method for solving partial differential equations. The space discretisation is done over small elements (called the finite elements) of usually simple shape such as triangles or quadrilaterals in 2 dimensions or tetrahedrons in 3 dimensions. Shape functions are then defined on each element and the solution is approximated by being a linear combination of those functions (which typically are simple). Each element contains a certain number of nodes and the problem is treated by solving a system involving the value of the solution at each of those nodes.

Figure 2.1 gives some example of pieces shapes function in 2 dimensions (Pictures were obtained from [13]). One shape function is assigned to each node and is composed by each sub-shape function contiguous to the node.
Increasing the number of nodes per element means having more complicated shape functions.

**Simple example in 1 dimension**

A simple example is explained in the following to get some insights on how the method works.

Consider an elastic rod of length $L$ subjected to an axial loading $F$ and denote $u$ the displacement of points along this rod (this example is inspired from [1]).

Then the equation governing this problem is of the form:

$$\frac{d^2u}{dx^2} + F = 0 \quad (2.1)$$
We now decide to divide the rod into 3 pieces (3 elements) with 2 nodes per element. The shape functions are chosen as hat functions (there are 4 of them because there are 4 nodes as a whole). See Figure 2.2

![Figure 2.2: Shape functions](image)

A approximate solution \( \hat{u} \) is sought, being a linear combination of the shape functions \( \phi_i \):

\[
\hat{u} = \sum_{i=1}^{4} u_i \phi_i
\]

In Galerkin’s method, the most widely used in FEM (and actually used in ParaFEM), each side of the equation 2.1 is multiplied by a function part of the shape functions vectorial space \( v \) and integrated over the whole region:

\[
\int_0^L \frac{d^2 u}{dx^2} v + F v \ dx = 0, \quad \forall v \in \text{shape functions space}
\]

Using integration by parts (Green’s formula in more than one dimension), one can simplify the equation in the form:

\[
\int_0^L \frac{du}{dx} \frac{dv}{dx} \ dx = \int_0^L F v, \quad \forall v \in \text{shape functions space}
\]

In Galerkin’s method, \( u \) is then replace by \( \hat{u} \) and \( v \) by all the shape functions, one at a time, leading to a system of equations:

---

1Picture picked from [14] and modified.
\[
\int_0^L \sum_{i=1}^4 u_i \frac{d\varphi_i}{dx} \frac{d\varphi_j}{dx} \, dx = \int_0^L F \varphi_j, \forall j = 1, \ldots, 4
\]

which can be written in matrix formulation:

\[
K \cdot u = f
\]

where

\[
K_{ij} = \int_0^L \frac{d\varphi_i}{dx} \frac{d\varphi_j}{dx} \, dx
\]

\[
f_i = \int_0^L F \varphi_i
\]

Solving this system will hence give an approximate solution to the problem.

### 2.1.2 ParaFEM

ParaFEM is a parallel finite element library, written in Fortran 90/95 and using MPI. It contains a collection of subroutines encapsulating the calls to the MPI library as well as a number of programs solving different types of partial differential equations, such as eigenvalue problems, transient problems or static equilibrium of linear elastic solids problems. Extra information is given in [15] and the code is available in [16].

In Figure 2.3 is given an example of the deformation of an elastic solid stressed in one corner, obtained after resolution using ParaFEM\(^2\)

\(^2\)Picture given by Dr. Lee Margetts
Element by element strategy

From the finite element discretisation, a linear system is obtained. For storage reasons, priority is given to iterative methods over direct methods. The PCG (Preconditioned conjugate gradient) algorithm is used. A good introduction to the conjugate gradient algorithm can be found in [2]. The main operations involved in this technique are matrix/vector multiplication and dot products.

Initialisation is done through

\[ p_0 = r_0 = f - Ku \]  

(2.2)

where \( r_0 \) is the first residual and \( p_0 \) the first search direction.

\[ q_k = Kp_k \]

\[ \alpha_k = \frac{r_k^T r_k}{p_k^T q_k} \]

\[ u_{k+1} = u_k + \alpha_k p_k \]

\[ r_{k+1} = r_k - \alpha_k q_k \]

\[ \beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k} \]

\[ p_{k+1} = r_{k+1} + \beta_k p_k \]
ParaFEM uses an element-by-element technique (also called mesh-free method). The global stiffness matrix $K$ is never assembled. Only local stiffness matrices are computed. The product $K.p$ is then carried out element by element as a sum of the type:

$$q = \sum_{i=1}^{\text{#elements}} K_i p_i$$

(2.3)

where $K_i$ is the stiffness matrix of the $i^{th}$ element, and $p_i$ is the appropriate part of $p$ corresponding to the the nodes bordering element $i$. In a parallel execution of the CG algorithm, whilst the matrix/vector products for each element set of one processor can be done independently, communications between processors are still needed before and after the sum. The dot products also involve communications since each processor carries out a local dot product on a piece of vector, and those results need to be summed up.

**Mesh partitioning**

The way the partitioning of the elements is done in ParaFEM is very naive, since it is just dependent on how the elements are numbered, but not on what their position is within the mesh. If they are say $N$ elements and $p$ processors, processor 1 gets basically the elements numbered from 1 to $\frac{N}{p}$, processor 2 the elements from $\frac{N}{p}$ to $\frac{2N}{p}$ and so on.

**2.2 METIS**

METIS is a family of programs used to partition graphs and hypergraphs. It contains several packages, including a serial partitioning program and a parallel one. We will use only the serial one, since it is simply fast enough to perform the partitions we are interested in.

The strategy used to partition the graph is called multilevel recursive-bisection. It is multilevel because it coarsens the original graph as much as possible, partitions it and then refines it. See Figure 2.4 (picked from [4]).
The way the partition is done is following a recursive-bisection strategy. The graph is divided step by step. First the whole graph is divided into two subgraphs according to a specific strategy. Each subgraph is then divided into two smaller graphs. The procedures carries on until the number of partition wished is reached. Figure 2.5 obtained from [12], illustrates this partitioning process.
In the multilevel recursive bisection strategy, the coarsened graph obtained is partitioned using recursive bisection and then refined.

METIS contains also a mesh partitioner, which is the feature that is of interest for our problem. It converts the input mesh into a graph, partition it and then output the partitions. In this case, the graph partitioner minimises the number of cut edges which will have the consequence on the mesh partition to minimize the total communication volume, that is, the total connectivity of the different subdomains. It is meant to also try to keep the number of adjacent subdomains as low as possible.

2.3 Machines

2.3.1 HECToR Cray XT4

Cray XT4 is a massively parallel processing system (MPP). It uses AMD 2.3 GHz Opteron processors. With the start-up of the HECToR Cray XT6 system, the XT4 system has been reduced by almost half its original size[7]. Out of the 5664 quad-core compute nodes remain 3072. This still represents an amount of 12288 cores. 8 GB of memory is shared between each quad-core socket. This gives the whole system a memory of 24.6 TB.

Each core has 2 levels of cache and there is a 3rd level, shared between the 4 cores. A core can operate up to 4 floating point operations per clock cycle which gives each core a theoretical peak of 9.2 Gflops [8]. This means the whole machine has a theoretical peak performance of around 113 Tflops.

Processors are connected through a 3D-Torus communication network via Cray SeaStar2 chips, one SeaStar chip being linked to each quad-core processor[9].

See Figure 2.6 for a more detailed description of a SeaStar chip.
Point-to-point bandwidth is 2.17 GB/s and the latency between two nodes is about 6µs [7].

In Figure 2.7 is represented the architecture of an XT4 node.

\[\text{Figure 2.6: a Cray SeaStar chip. Taken from [9]}\]

\[\text{Figure 2.7 is represented the architecture of an XT4 node}^{3}\]

\[\text{Picture slightly modified from [6]}\]
2.3.2 HECToR Cray XT6

The main differences between the Cray XT6 system and the XT4 system lies in the architecture present at each compute node. Where there used to be quad-core processors in the XT4, there are now two 12-core processors in the XT6.

Those are AMD 2.1 GHz Opteron processors delivering a peak performance of 8.4 Gflops. They are 1856 compute nodes wich makes a total of \( 24 \times 1856 = 44544 \) cores. This gives the system a theoretical peak performance of around 374 Tflops. 32 GB of memory is available at each node. The total memory of the system is hence almost 60TB.

Each 12-core processor is composed of 2 dies containing 6 cores. There are 3 levels of caches within each die, with the first two level private to each core and the 3rd one shared between the 6 cores. Dies are connected with Hyper Transport links (called HT links). The bandwidth of those links is proportional to how "close" a die is to the others: 38.4 GB/s to the neighbouring die within the same 12-core processor, 25.6 GB/s to the corresponding die in the other 12-core processor of the node and 12.8 GB/s to the "diagonally opposite" die in the other 12-core processor of the node [5]. Figure 2.8
gives detailed information of an XT6 node.

Figure 2.8: Architecture of an HECToR Cray XT6 node

An essential point is that the Cray XT6 is genuinely multi-core: on each node, it has much more computing power than the Cray XT4 (around 6 times more) but the communication network remains similar. One can wonder if it is actually possible to make efficient use of these type of systems, which are likely to become a standard in High Performance Computing in the next few years.

4 Picture taken from [5]
Chapter 3

Porting

During this project, a mixed OpenMP/MPI version of the code was implemented. This chapter presents the different phases for compiling and running this code as well as the original code which is a pure-MPI code. A much more detailed explanation on how the mixed code was implemented is given in Chapter 5.

3.1 Compiling

For all the compiling steps described in the following, the Portland Group compiler was used. This compiler was pre-installed on both the Cray XT4 and XT6 machines. A call with this compiler is made through the ftn wrapper which calls the compiler associated to the currently loaded compiler module.

To compile the modules, the command used is essentially:

```
ftn -c <module>.f90
```

Some flags are also added for compiler optimisation. All the .o files are then archived in the library using the command ar.

The compilation of the programs is also made using the ftn command, using some extra flags to link the program to the ParaFEM library.

3.1.1 The original code

The source code of Parafem is divided into 2 pieces, one for the modules that contain all the library and one for the programs (which call this library). The compilation is done in two steps, the modules have to be compiled first, and then the programs. A makefile is provided for both steps.
3.1.2 The mixed OpenMP/MPI version

Compilation of a mixed OpenMP/MPI code is actually quite straightforward. All what is required is to add a specific flag to the original compiling command. We can note that this is done slightly differently depending on which compiler is used. Check Table 3.1.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Portland Group</td>
<td>-mp=nonuma</td>
</tr>
<tr>
<td>Pathscale</td>
<td>-mp</td>
</tr>
<tr>
<td>Cray compilers</td>
<td>OpenMP supported by default</td>
</tr>
<tr>
<td>GNU compilers</td>
<td>-fopenmp</td>
</tr>
</tbody>
</table>

Table 3.1: Flags to be used for use of OpenMP

Since the code is compiled with the PGI compiler, all what is needed is hence to add the flag -mp=nonuma to the compiling command.

3.2 Running

A certain number of input files are required to run ParaFEM. Those files includes data relevant to define the problem, such as the mesh, the boundary conditions, the loading applied on the solid, or other information such as the total number of elements or the limit number of iterations.

3.2.1 The original code

The code is run through the command aprun. To run on the Cray XT4 or XT6 machines, a batch script has to be done, containing essentially PBS(Portable Batch System) directives and the aprun command, which is then submitted using the qsub command.

PBS directives Various PBS directives are given, such as the number of MPI tasks requested mppwidth, the number of MPI tasks per XT node mppnppn, or the walltime walltime which is the time after which the job will be killed if its execution is not terminated. In the following is shown an example of PBS directives on the XT6 system:

```
#PBS -A d04-og1
#PBS -N ParaFEM
#PBS -l mppwidth=120
#PBS -l mppnppn=24
```
This means that 5 nodes will be allocated, with 24 MPI tasks per node \((5 \times 24 = 120)\). This is true for the Cray XT6. On the XT4, only up to 4 MPI processes per node can be allocated. The walltime will be 20 minutes, the job name will be ParaFEM (PBS -N ParaFEM). Budget will be taken from d04-og1 (from the PBS -A directive) and the output and error files will be joint (PBS -j oe).

**aprun** The aprun is used to run the code. It contains a number of options to specify how many MPI tasks will be created and how many of those tasks there will be per node. An example is given below:

```
aprun -n 120 -N 24 ./p121 p121-large
```

120 MPI processes will be created with 24 of them per node. Again, note that this can be run only on the XT6, the XT4 having only 4 cores per node. The executable called here is p121 and the input files called are named after p121-large.

The number of MPI processes created by aprun in total, or per node, can’t exceed what was requested within the PBS directives.

### 3.2.2 The mixed OpenMP/MPI version

The job is submitted in the same way than the original code. The submission script is similar, though some extra options have to be added to the aprun command, and some environment variables have to be set.

**Environment variables** First of all the number of threads has to be set through the `OMP_NUM_THREADS` environment variable. The variable `PSCOMP AFFINITY` has to be set to `FALSE` to ensure better performance [5].

**aprun** The –d flag is used to set the number of threads per MPI processes. The –S flag sets the number of MPI processes per NUMA region within a XT node.

An example of submission script on the Cray XT6 is shown below.

```
#PBS -A d04-og1
#PBS -N ParaFEM
#PBS -l mppwidth=120
#PBS -l mppnpnn=24
#PBS -l walltime=00:19:59
#PBS -j oe
```
The PBS directives will allocate 5 nodes and all the 24 cores available per node. 20 processes will be created with 4 processes per node. On each node, those 4 processes will be dispatched so exactly 1 process will be defined on each of the 4 NUMA subregions. Each of those processes will then spawn 6 threads.

Each NUMA region has a local memory, and is hence well suited to shared memory programming. This way of dispatching the threads is thought to be the most sensible one.

## 3.3 Compiler optimisations

Compiler optimisations can have a strong impact on the performance of a code.

### 3.3.1 Flags

The code is here compiled with the Portland Group compiler. This compiler has a number of flags for compiling optimisation. The flag `−O <level>` specifies the level of optimisation. This level can go from 0 to 4. The flags `−fast` and `fastsse` contain a general set of options, regarded as generally efficient, with `−fastsse` extending `−fast` to SSE hardware [17].

### 3.3.2 Results

The main program was compiled using different combinations of the flags presented in the previous subsection. Results are displayed in Figure 3.3.2.
\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
Compiler flags & Execution time (sec) \\
\hline
-00 & 14.02 \\
-01 & 13.73 \\
-02 & 13.49 \\
-03 & 13.52 \\
-04 & 13.49 \\
-fast & 12.85 \\
-fast -03 & 12.78 \\
-fast -04 & 13.03 \\
-fastsse & 12.79 \\
-fastsse -03 & 12.88 \\
-fastsse -04 & 12.78 \\
\hline
\end{tabular}
\caption{Runtime with different compiler options}
\end{table}

-fast is the option used by the original program. We can see that -fastsse and -fastsse -04 are the fastest. However we can also see that the combinations -fast, -fast -03 and -fastsse -03 are giving almost similar results. We can essentially say that using any of those 5 combinations won’t make a dramatical change on the speed of the code.
Chapter 4

Profiling and Testing

The problem tested is called p121, and is defined in [1] (program 1 from Chapter 12). It models the three-dimensional strain of an elastic solid. The solid represented is a cuboid, strained in one corner. The mesh is made of 20-node hexahedrons. See Figure 4.1.

Figure 4.1: A 20-node hexahedron

In the following we will try out this problem, varying sometimes the refinement of the

---

mesh, that is, the number of hexahedrons the mesh is made of (hexahedrons will be referred to as element from now on).

Tests will be conducted on the HECToR Cray XT4 and XT6 machines.

4.1 Preliminary performance analysis

4.1.1 Timings within the code

We first consider a mesh of rather small size, that is with 2560 elements, which represents 12465 nodes and 36720 equations. First of all, a run is made using one core only on the XT4 machine. The program can be divided into several steps. ParaFEM outputs some basic timing information: it presents the different sections, the time spent in each of them, and the relative percentage spent in each section. This output is printed below.

<table>
<thead>
<tr>
<th>Program section execution times</th>
<th>Seconds</th>
<th>%Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>0.135398</td>
<td>1.04</td>
</tr>
<tr>
<td>Compute coordinates and steering array</td>
<td>0.158152</td>
<td>1.22</td>
</tr>
<tr>
<td>Compute interprocessor communication tables</td>
<td>0.009429</td>
<td>0.07</td>
</tr>
<tr>
<td>Allocate neq_pp arrays</td>
<td>0.001718</td>
<td>0.01</td>
</tr>
<tr>
<td>Compute element stiffness matrices</td>
<td>0.879180</td>
<td>6.78</td>
</tr>
<tr>
<td>Build the preconditioner</td>
<td>0.004425</td>
<td>0.03</td>
</tr>
<tr>
<td>Get starting r</td>
<td>0.027243</td>
<td>0.21</td>
</tr>
<tr>
<td>Solve equations</td>
<td>11.686239</td>
<td>90.07</td>
</tr>
<tr>
<td>Recover stresses</td>
<td>0.011856</td>
<td>0.09</td>
</tr>
<tr>
<td>Output results</td>
<td>0.061662</td>
<td>0.48</td>
</tr>
<tr>
<td>Total execution time</td>
<td>12.975302</td>
<td>100.00</td>
</tr>
</tbody>
</table>

Result 1: Output from ParaFEM of p121 with 2560 elements

We can see that around 90\% of the execution time is spent within the section "Solve equations". Another section that takes a significant time (7\%) is the one where the computation of the stiffness matrices is done.

We are now making another run but with a more refined mesh: 27000 elements, 116281 nodes and 329340 equations. The problem becomes roughly 10 times bigger. Output is printed below.
We see that the section 2, namely the "Compute coordinates and steering array" part, is now taking around 47% of the total execution time which is similar to the time taken by section 8, "Solve equations". A comparison of those 2 results is given in Figure 4.2.

It becomes apparent that the section of code where the computation of the coordinates and the steering array take place is increasing in importance with the problem size. Its execution time is comparable to the one of solving the equations. For performance purposes, those two parts need to be especially taken care of.

### 4.1.2 Profiling tools

To obtain some more specific insights in the code performance, the code was profiled using different profiling tools: Scalasca and gprof. gprof was used because it gives sim-
ple information such as the number of function calls as well as the call graph. Scalasca is able to give more detailed information on the relative timings of each routine and the time spent doing communications. Using those two tools together should give us a good understanding of the code’s behaviour.

Scalasca

Scalasca is installed on HECToR Cray XT4. It supports measurement and analysis of MPI and OpenMP and gives a number of useful information for evaluating the code’s performance. Results can be interactively checked through a graphical report explorer. The graphical output explorer gives different characteristics of the performance, such as the timing in general, but also the time spent doing communications, or the size of the data exchanged. It also shows computational imbalance. All this information is given for each routine, with a "warmer" color for the "hot spots" (from blue to red). This makes it easy to find out where there is room for optimisation. Results are presented in a tree representing which routine called which one. Scalasca can also produce textual results giving essentially timing information.

test

We first consider p121 with a mesh of 125000 elements. We run it using 120 cores. The graphical output from Scalasca is presented in Figure 4.3 and the textual output in Table 4.1.

![Scalasca result presentation](image)

Figure 4.3: Scalasca result presentation (Here in the case of a mesh of 125000 elements)

In Figure 4.3, which is set to show timings, we can see that they are 2 routines taking
a relatively significant time. Those are find_g and the main program p121 (the time of a routine is printed without including the time spent within the subroutines it calls: p121 would take 100% of the time otherwise). This can be rechecked in Table 4.1. This output has been shortened, keeping only the important routines for clarity. Type MPI refers to routines from the MPI library, COM refers to the ones that are user-defined but call MPI directly or indirectly and USR the ones performing purely local computations. max_tbc corresponds to the size of the buffer used for the tracing of each routine.

<table>
<thead>
<tr>
<th>flt</th>
<th>type</th>
<th>max_tbc</th>
<th>time</th>
<th>% region</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY</td>
<td>2180668</td>
<td>3753.96</td>
<td>100.00</td>
<td>(summary) ALL</td>
</tr>
<tr>
<td>MPI</td>
<td>1025452</td>
<td>1211.34</td>
<td>32.27</td>
<td>(summary) MPI</td>
</tr>
<tr>
<td>COM</td>
<td>169776</td>
<td>1382.81</td>
<td>36.84</td>
<td>(summary) COM</td>
</tr>
<tr>
<td>USR</td>
<td>985392</td>
<td>1150.57</td>
<td>30.65</td>
<td>(summary) USR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>flt</th>
<th>type</th>
<th>max_tbc</th>
<th>time</th>
<th>% region</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td>283000</td>
<td>22.12</td>
<td>0.59</td>
<td>MPI_Isend</td>
</tr>
<tr>
<td>MPI</td>
<td>260360</td>
<td>27.59</td>
<td>0.73</td>
<td>MPI_Recv</td>
</tr>
<tr>
<td>MPI</td>
<td>141440</td>
<td>117.91</td>
<td>3.14</td>
<td>MPI_Barrier</td>
</tr>
<tr>
<td>MPI</td>
<td>141080</td>
<td>271.42</td>
<td>7.23</td>
<td>MPI_Allreduce</td>
</tr>
<tr>
<td>USR</td>
<td>84864</td>
<td>0.79</td>
<td>0.02</td>
<td>elap_time</td>
</tr>
<tr>
<td>COM</td>
<td>67824</td>
<td>0.49</td>
<td>0.01</td>
<td>my_barrier</td>
</tr>
<tr>
<td>COM</td>
<td>50760</td>
<td>14.31</td>
<td>0.38</td>
<td>dot_product_p</td>
</tr>
<tr>
<td>MPI</td>
<td>33960</td>
<td>1.32</td>
<td>0.04</td>
<td>MPI_Waitall</td>
</tr>
<tr>
<td>USR</td>
<td>25008</td>
<td>1147.24</td>
<td>30.56</td>
<td>find_g</td>
</tr>
<tr>
<td>MPI</td>
<td>19520</td>
<td>464.65</td>
<td>12.38</td>
<td>MPI_Bcast</td>
</tr>
<tr>
<td>COM</td>
<td>16968</td>
<td>118.92</td>
<td>3.17</td>
<td>gather</td>
</tr>
<tr>
<td>COM</td>
<td>16944</td>
<td>74.30</td>
<td>1.98</td>
<td>scatter</td>
</tr>
<tr>
<td>MPI</td>
<td>9720</td>
<td>7.39</td>
<td>0.20</td>
<td>MPI_Reduce</td>
</tr>
<tr>
<td>MPI</td>
<td>100</td>
<td>0.31</td>
<td>0.01</td>
<td>MPI_Send</td>
</tr>
<tr>
<td>MPI</td>
<td>96</td>
<td>0.00</td>
<td>0.00</td>
<td>MPI_Test</td>
</tr>
<tr>
<td>EPK</td>
<td>48</td>
<td>9.24</td>
<td>0.25</td>
<td>TRACING</td>
</tr>
<tr>
<td>COM</td>
<td>24</td>
<td>2.25</td>
<td>0.06</td>
<td>shutdown</td>
</tr>
<tr>
<td>MPI</td>
<td>24</td>
<td>280.73</td>
<td>7.48</td>
<td>MPI_Finalize</td>
</tr>
<tr>
<td>COM</td>
<td>24</td>
<td>1149.27</td>
<td>30.61</td>
<td>p121</td>
</tr>
<tr>
<td>COM</td>
<td>24</td>
<td>0.00</td>
<td>0.00</td>
<td>find_pe_procs</td>
</tr>
<tr>
<td>MPI</td>
<td>24</td>
<td>13.44</td>
<td>0.36</td>
<td>MPI_Init</td>
</tr>
<tr>
<td>MPI</td>
<td>24</td>
<td>0.00</td>
<td>0.00</td>
<td>MPI_Comm_rank</td>
</tr>
<tr>
<td>MPI</td>
<td>24</td>
<td>0.00</td>
<td>0.00</td>
<td>MPI_Comm_size</td>
</tr>
</tbody>
</table>

Table 4.1: Output from scalasca (truncated for clarity) with 120 processes and 125000 elements in the mesh

We can see here that basically a third of the time is spent in the find_g routine and another third in the general calculations of the p121 main program (not including subroutine calls). The find_g routine is used in the part where the steering array is computed. In section 4.1.1 we already noticed that this part represents a large chunk of
the total execution time. This routine is of type USR, which means it does not involve any communications and is just made of local computations. The fact that another large chunk is taken by the main program `p121` is not very informative, since it doesn’t precisely tells us where this time is spent. However, looking back to section 4.1.1 it is obvious to spot that the chunk of time relevant is coming from the part where equations are solved. A more detailed analysis of this section will be done.

In Figure 4.4 is shown a comparison between the percentage of the runtime spent doing computations versus doing communications.

![Figure 4.4: Percentage of runtime spent in MPI routines time and Computations with 60 and 120 processors](image)

We see that the communication time proportion increases with the number of processors (from 20% using 60 processors to 30% using 120 processors). MPI_Bcast and MPI_Allreduce are the MPI routines taking most of the time. MPI_Finalize also takes a noticeable amount of time.

**Gprof**

To get information about the number of calls made to each function, gprof was used. Gprof is not fully functional on HECToR since timings are not supported. However calls and call graphs do work. In Table 4.1.2 is presented a shortened output of the calls from gprof. This was generated using 2 processors, which explains why many of the functions used only once in serial have 2 calls. Each routine name is preceded by the module file name it belongs to.
The 4 most-called functions (beemat, shape_der, invert, determinant) are all involved within the computation of the element stiffness matrices. One process calls them essentially as many times as the number of elements the process has been assigned times the number of points used to perform the numerical integration of the shape function on each element (typically 8). Those routines are mathematical and quite optimised already. Furthermore, results from profiling in the previous section showed that despite the large number of calls we see here, those routines do not have a large impact on the execution time.

Then, the find_g function, that we spotted taking a significant time in the previous section, is called in total as many times as the number of elements. The routines gather and scatter are called when solving the equations, using the preconditioned gradient algorithm. The call to those two routines are made around the matrix vector multiplication made element by elements presented in the equation 2.3. They are called one time in each iteration and both of them call 2 times the my_barrier routine which involves an MPI_barrier.

All the other routines are essentially called once per MPI task.
4.2 Tests

We are now going to test the problem p121 from ParaFEM (presented at the beginning of this chapter) with various mesh sizes on the HECToR Cray XT4 and XT6 machines.

4.2.1 Scaling

We want to get a first idea on how well ParaFEM scales. We are first going to carry out our experiments on the XT4 machines. The XT6 will be tested in section ??.

We first consider a rather small mesh, containing 2560 elements, 12465 nodes, and involving the resolution of 36720 equations. This problem can be considered as having a small size. Results are gathered in Table 4.3 and the Speedup is displayed in Figure 4.5. The number of cores has been varied from 1 to 128. For visibility, a logarithmic scale has been used.

<table>
<thead>
<tr>
<th>Number of cores</th>
<th>Execution time (sec)</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12.98</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>7.30</td>
<td>1.8</td>
<td>0.89</td>
</tr>
<tr>
<td>4</td>
<td>5.15</td>
<td>2.5</td>
<td>0.63</td>
</tr>
<tr>
<td>8</td>
<td>2.97</td>
<td>4.4</td>
<td>0.55</td>
</tr>
<tr>
<td>16</td>
<td>1.68</td>
<td>7.7</td>
<td>0.48</td>
</tr>
<tr>
<td>32</td>
<td>1.25</td>
<td>10.4</td>
<td>0.32</td>
</tr>
<tr>
<td>64</td>
<td>0.97</td>
<td>12.4</td>
<td>0.20</td>
</tr>
<tr>
<td>128</td>
<td>1.01</td>
<td>11.8</td>
<td>0.10</td>
</tr>
</tbody>
</table>

Table 4.3: Execution time, Speedup, and Efficiency of the code
Figure 4.5: Speedup for p121 on a mesh of 2560 elements, involving 36720 equations

We can see that the scaling is not ideal (speedup of about 10 using 32 processors). It is especially going bad beyond 8 cores: the efficiency is then less than \( \frac{1}{2} \). One obvious reason for this is that the problem is too small to scale well beyond a small number of processors. Despite that, we will try to get an understanding of what takes time during the execution and do all the optimisations possible before considering a larger problem.

In Figure 4.6, is presented a more detailed speedup graph, presenting the speedup of different sections of the code.
We can see that some sections of the code do scale almost linearly: the ones where the steering array and the stiffness matrices are computed. Looking back to the previous section, we noticed that computing the stiffness matrices had no significant impact on the total runtime. However, computing the steering array could be taking an important time in the case the mesh is containing a large enough number of elements. This section is not involving any communications between processors. We can deduce that its excellent scaling comes from the fact there is a good load balance among the processors.

The setup section is essentially serial since it is just one processor which reads the input files and send the data to the other processors. Its speedup is then obviously non-relevant.

The section where the conjugate gradient algorithm is applied (Solving the equations) does not have a perfect scaling: speedup is about 14 with 32 processors. It involves a relatively important quantity of communications which can explain those performances. A more detailed analysis of this part of the code will be done in section 5.1.

4.2.2 XT4 versus XT6

Some comparisons between the performance of ParaFEM on those two systems will be done. Runs with a number of processors which is a multiple of 24 will be carried out, so that all processors within an XT6 node can be used.
We reconsider the medium-sized mesh of 125000 elements, observe the performance obtained on both machines. Results are displayed in Table 4.4 and Figure 4.7. Efficiency was calculated using the runtime of 24 cores as a reference (this is equivalent to consider that the code scales perfectly until 24 cores). The formula for efficiency used was: $E = \frac{T_{24}}{T_p}$ (where $T_p$ is the runtime with $p$ cores). A run with a single core would have been very expensive on the XT6 since, even if just one core is used, the whole node is charged.

<table>
<thead>
<tr>
<th>Nb of cores</th>
<th>On the XT4</th>
<th>On the XT6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Runtime</td>
<td>Efficiency</td>
</tr>
<tr>
<td>24</td>
<td>120.33</td>
<td>1</td>
</tr>
<tr>
<td>48</td>
<td>64.4</td>
<td>0.93</td>
</tr>
<tr>
<td>72</td>
<td>45.8</td>
<td>0.88</td>
</tr>
<tr>
<td>96</td>
<td>36.0</td>
<td>0.84</td>
</tr>
<tr>
<td>120</td>
<td>30.5</td>
<td>0.79</td>
</tr>
<tr>
<td>240</td>
<td>19.1</td>
<td>0.63</td>
</tr>
<tr>
<td>480</td>
<td>13.9</td>
<td>0.43</td>
</tr>
</tbody>
</table>

Table 4.4: Comparison of Runtime and Efficiency on HECToR XT4 and XT6, when increasing the number of cores
First of all, we can see the code scales reasonably well on both machines, the estimate of the efficiency stays over one half as long as the number of processors is smaller than 400. Timings on the XT6 are a bit slower than on the XT4. In Table 4.5 are printed the ratio of the execution times between XT4 and XT6 (How much slower XT6 is, compared to XT4).

Figure 4.7: Comparison of the performance of p121 on HECToR XT4 and XT6
<table>
<thead>
<tr>
<th>Number of cores</th>
<th>XT6 Runtime</th>
<th>XT4 Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>1.111339</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>1.0942861</td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>1.0841296</td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>1.0784693</td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>1.076283</td>
<td></td>
</tr>
<tr>
<td>240</td>
<td>1.0848931</td>
<td></td>
</tr>
<tr>
<td>480</td>
<td>1.1320504</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: Ratio of runtimes between HECToR XT4 and XT6

This difference comes from the fact XT4 uses 2.3 GHz cores whilst XT6 uses 2.1 GHz ones. The ratio of those 2 speeds is $\frac{2.3}{2.1} \simeq 1.095$. We can see that the ratio of the runtimes varies around this value: whether it is larger or smaller depends on the architecture and means that one system is more or less efficient than the other one. Looking at the efficiency graph (Figure 4.7(b)), we can see the XT6 system is slightly more efficient than the XT4 when the number of processors is less than 400.

### 4.2.3 Specific tests on HECToR Cray XT6

In this section, we will carry out some specific tests on HECToR XT6. We would like now to analyse the effects of using multi-cores processors, and more precisely how increasing the number of cores per node affects the performance.

To specifically test how the performance changes when using many-core processors, we will make several runs, using always the same number of cores. The differences between these runs will be that even though the number of cores stays identical, we will use less and less nodes, so that the runs will be less and less "packed" (Fewer cores per processors will be used).

In Figure 4.8 is displayed a graph showing the performance of p121 on a medium-sized-mesh (125000 elements). The experiments were carried out using always 120 cores, but dispatching those cores differently onto the nodes (6 cores per node on 20 nodes, 12 cores per node on 10 nodes and 24 cores per node on 5 nodes).
Figure 4.8: Scaling of p121 using a fixed number of cores, but gathering them differently

We can see that the execution time does not differ much, no matter how the cores are dispatched. This can let us think that the p121 program of ParaFEM do not have issues with congestion appearing onto a node.

In Figure 4.9 and 4.10 is shown the scaling of p121 using an increasing number of cores on 5 nodes. This goes up to 24 cores on each node, so 120 cores.
Figure 4.9: Execution time of different sections of p121 using always 5 nodes

Figure 4.10: Speedup of different sections of p121 using a fixed number of nodes
We can see that there is no significant drop in the speedup when the number of cores used on a node gets large. p121 handles well the multi-core processor architecture. One could have expected some congestion on each node since much more data is transferred from node to node when using 24 cores instead of just 4 for example. The quality of the interconnect seems to be good enough to carry out all those communications. In this case, it is the bandwidth that is large enough to handle the sizes of the messages. An interpretation is that the amount of data transferred at a time between each processor in the p121 program is small compared to the bandwidth of the XT6 interconnect. This allows the code to run as efficiently on a 24-core node as on a 4-core one.
Chapter 5

Optimisations and Results

5.1 Use of BLAS routines

To carry out vector and matrix operations, ParaFEM uses Fortran 90 intrinsic functions and array manipulation operators. A straightforward optimisation that can be tried out is replacing those functions where possible by some of BLAS routines. Those replacements will be restricted to the section "solve equations" of the code, which is one of the most computationally expensive and contains several vector operations.

5.1.1 Implementation and Results

Implementation

The "solve equation" section can essentially be divided into 4 parts which are presented in Figure 5.1.
The following explains what was replaced.

- The section 2. is made of one matrix vector multiplication using \texttt{MATMUL}:
  \[
  \text{utemp}\_pp(:,:,iel) = \text{MATMUL(}\text{storkm}\_pp(:,:,iel),\text{pmul}\_pp(:,:,iel))
  \]
  This has been replaced by a call to the routine \texttt{DGEMV} (not \texttt{SEGMV} since the code works in double precision).

- The section 4. involved several vector operations:
  \[
  \begin{align*}
  \text{up} & = \text{DOT\_PRODUCT\_P}(r\_pp,d\_pp) \\
  \text{alpha} & = \text{up}/\text{DOT\_PRODUCT\_P}(p\_pp,u\_pp) \\
  x\_\text{new}\_pp & = x\_pp + p\_pp*\text{alpha} \\
  r\_pp & = r\_pp - u\_pp*\text{alpha} \\
  d\_pp & = \text{diag\_precon}\_pp*r\_pp \\
  \text{beta} & = \text{DOT\_PRODUCT\_P}(r\_pp,d\_pp)/\text{up} \\
  p\_pp & = d\_pp + p\_pp*\text{beta}
  \end{align*}
  \]
  Those were replaced in a relative straightforward way using several of the other double precision BLAS subroutines, such as \texttt{DDOT} (for the dot product of two vectors),
DAXPY (multiplication of a vector by a scalar and vector addition), but also DCOPY (makes the copy of one vector into another) and DSCAL (simply multiply a vector by a scalar). Those were used to complete the BLAS operations so that they correspond to the vector operations actually done: there are no such function doing an operation of the type \( z = \alpha x + y \) or \( x = \alpha x + y \). This involves extra operations, but we hope for those to be balanced by the BLAS speed up. These extra operations could have been skipped by using pointers, but this was not implemented. DOT\_PRODUCT\_P, performing the dot product locally on each process (using the intrinsic Fortran function DOT\_PRODUCT) and communicating with the others to obtain the full dot product, was replaced by a similar DOT\_PRODUCT\_BLAS function calling the DDOT BLAS routine. The BLAS version of the code is printed below.

```fortran
up = DOT\_PRODUCT\_BLAS(neq\_pp,r\_pp,d\_pp)
alpha = up/DOT\_PRODUCT\_BLAS(neq\_pp,p\_pp,u\_pp)

CALL DCOPY(neq\_pp,x\_pp,1,xnew\_pp,1)  ! copy x\_pp to xnew\_pp
CALL DAXPY(neq\_pp, alpha, p\_pp,1,xnew\_pp,1)  ! xnew\_pp = xnew\_pp + p\_pp*alpha

CALL DAXPY(neq\_pp, -alpha, u\_pp, 1, r\_pp, 1)  ! r\_pp = r\_pp - u\_pp*alpha

d\_pp = diag\_precon\_pp*r\_pp  ! No such function in the BLAS library
beta = DOT\_PRODUCT\_BLAS(neq\_pp,r\_pp,d\_pp)/up
CALL DSCAL(neq\_pp, beta, p\_pp, 1)  !p\_pp = beta*p\_pp
CALL DAXPY(neq\_pp, 1.0\_iwp, d\_pp, 1, p\_pp, 1)  !p\_pp = 1.0*d\_pp + p\_pp
```

**Results**

We include some specific timing around the 4 sections presented in Figure 5.1 and make a run using a large mesh of a million elements and 120 MPI tasks on 120 cores on both the XT4 and the XT6. The problem has 12,059,800 equations. To make it less costly, we don’t run until completion, but stop after 10 conjugate gradient iterations. The timings are summed up over the 10 iterations. Results are shown in Table 5.1.

<table>
<thead>
<tr>
<th>Section</th>
<th>Time on the XT4</th>
<th>Time on the XT6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
<td>With BLAS</td>
</tr>
<tr>
<td>Gather</td>
<td>0.1327</td>
<td>0.1319</td>
</tr>
<tr>
<td>Matrix/Vector Multiplication</td>
<td>0.0758</td>
<td>0.0762</td>
</tr>
<tr>
<td>Scatter</td>
<td>0.1089</td>
<td>0.1093</td>
</tr>
<tr>
<td>Vector Operations</td>
<td>0.0870</td>
<td>0.0867</td>
</tr>
</tbody>
</table>

Table 5.1: Specific timings on the "solve equations" section. A million element mesh, 120 processors, stopped after 10 iterations.
We can see that the BLAS routines do not give a very significant improvement in the performance. The vector operations on the XT6 are slightly speeded up (by about 5.7%). There is no noticeable improvement on the XT4.

5.1.2 flop rate evaluation

To get some insights and try to understand the results, we will evaluate the flop rate achieved by both the original code and the BLAS version, on the section where the vector updates are carried out (the fourth section), and see which percentage of the peak flop rate this represents.

**Size of the vectors** The size of the vectors is dependent on the number of equations of the problem. It is basically the number of equations divided by the number of MPI processes (sometimes rounded up or down if this number is not an integer). It is called \( \text{neq}_{\text{pp}} \) in the code and its value is exactly \( 12059800 \) in this case which we will round up to 12 millions for clarity. We hence have \( \text{neq}_{\text{pp}} = \frac{12059800}{120} = 1.0 \times 10^5 \).

**flop calculation** There are 3 dot products which count each for \( 2 \times \text{neq}_{\text{pp}} \) flops. The operations of the type \( x = y + \alpha \ast z \) are also at the cost of \( 2 \times \text{neq}_{\text{pp}} \) flops.

Finally the operation \( d_{\text{pp}} = \text{diag}_{\text{precon}_{\text{pp}}} \ast r_{\text{pp}} \) is multiplying 2 vector term by term and this hence requires \( \text{neq}_{\text{pp}} \) flops.

As a whole, this makes a total of \( 13 \times \text{neq}_{\text{pp}} \) flops for this piece of code. The version with the BLAS routine is involving \( 15 \times \text{neq}_{\text{pp}} \) flops (the extra flops coming from the call to \text{DCOPY} and \text{DSCAL}).

For the original code, this was run 10 times and took 0.0870 seconds, so the flop rate on this section is:

\[
10 \times \text{Number of flops} / \text{Time} = 10 \times \frac{13 \times 10^5}{0.0870} = 0.15 \text{ Gflops/s}
\]

For the BLAS version, we obtain:

\[
10 \times \frac{15 \times 10^5}{0.0867} = 0.17 \text{ Gflops/s}
\]

The clock rate of an XT4 core is 2.3 GHz and it can perform up to 4 operations at a time, so the peak performance of one core is 9.2 Gflops/s. The flop rate of this piece of code is hence around 1.6% of the peak for the original code, and around 1.9% for the BLAS version, which is very poor, especially for the BLAS case.

Flop rates on both the XT4 and the XT6 are displayed in Table 5.2.

---

1 Floating Point OPeration
<table>
<thead>
<tr>
<th>Code</th>
<th>Rate on the XT4</th>
<th>Rate on the XT6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Code</td>
<td>1.6%</td>
<td>1.9%</td>
</tr>
<tr>
<td>BLAS version</td>
<td>1.9%</td>
<td>2.3%</td>
</tr>
</tbody>
</table>

Table 5.2: Flop rates of the original code and the BLAS version on HECToR Cray XT4 and XT6

An explanation of this very poor performance is that the piece of code involves some communications between processor at each call of the `DOT_PRODUCT_P` or the `DOT_PRODUCT_BLAS` function. The cores are hence not performing computations all the time.

To see what is the flop rate on pure calculation, we time 2 specific lines of the code involving pure computations:

```
CALL DAXPY(neq_pp, alpha, p_pp,1,xnew_pp,1)
CALL DAXPY(neq_pp, -alpha, u_pp, 1, r_pp, 1)
```

The time obtained is 0.0231 seconds on the XT4, giving a poor flop rate of 1.9%. This flop rate is actually identical to the one obtained when timing all the vector operations. We would expect BLAS routines in double precision to perform to a rate close to 10% of peak.

A small test case involving two calls to DAXPY in serial was done to check what flop rate is obtained on a very simple example. Results were quite unstable, depending on the vector size or the previous run executed. This poor BLAS performance can be possibly explained by complicated cache alignment issues.

### 5.2 Implementation of a Hybrid version of ParaFEM: OpenMP and MPI

Multi-core processors are likely to become the new standard in the future of High End Computing. To try to take advantage of the architecture of the HECToR XT6 machine, a hybrid version of the code was implemented and tested.

#### 5.2.1 Preliminary analysis

Implementation of the hybrid code requires including OpenMP directives where possible in the code. To compare the performance of the pure MPI version and the hybrid version, the codes will be benchmarked on the same number of cores. Then the number of MPI tasks in the hybrid code will be smaller than in the pure MPI code but each of those will spawn a certain number of threads so that we get the equality:
Number of threads $\times$ Number of MPI processes in the hybrid code = Number of MPI processes in the pure MPI code

A key point to understand is that all the bits of code which won’t be parallelised using OpenMP will go slower since they are fewer MPI processes assigned to them.

We have to decide how many MPI processes and how many threads will be used per node. An advised choice is to have 4 MPI tasks, one on each die of a node and then have 6 threads on each die. We do get $4 \times 6 = 24$ cores used per node. This sounds like a good choice since memory access is very quick within each die. Figure 5.2 represents such a partition. One core of each die has been put apart for clarity.

![Figure 5.2: One MPI task per die and 6 threads spawned](image)

Another choice is for example having just one MPI task per node that is spawning 24 threads (shown in Figure 5.3).
Figure 5.3: One MPI task per node and 24 threads spawned

Work is divided between the MPI tasks essentially through the fact that the elements of the mesh (the finite elements) are divided between them. In the case of the pure MPI code, a chunk of those elements was actually assigned to each core. In the case of the hybrid code, those elements will be assigned to each node (in the case we use 1 MPI task per node) or to each die (in the case we use 4 MPI tasks per node with 1 per die).

5.2.2 Implementation

From section 4.1.1, we know that they are 2 parts in the code that need to be especially taken care of since they are responsible for about 90% of the execution time: the “compute the steering array” part and the “solve equations” part. Hence, efforts will be made to put OpenMP directives in those parts.

- In the section where the steering array is computed, they are several loops over the number of elements assigned to each MPI processes. A rather obvious way of including OpenMP is to use the parallel loop directive around those type of loops. The way this was down is shown below.

```c
!$OMP PARALLEL
```
!$OMP DO
    DO iel=1,nels_pp
    ...
    END DO
!$OMP END DO
!$OMP END PARALLEL

where nels_pp represents the number of elements given to each MPI task in this case.

• In the section where the equations are solved, there are also a number of vector operations involved in the conjugate gradient algorithm. Those vector operations are applied in parallel already: a piece of vector, whose length is equal to the number of equations to be solved, is given to each MPI process. When the number of elements (and hence the number of equations) is high compared to the number of MPI processes, those vectors can be of a relatively large size. OpenMP directives can be easily included in those parts using the WORKSHARE directive. An example is given below:

!$OMP PARALLEL
!$OMP WORKSHARE
    xnew_pp = x_pp + alpha*p_pp
    ...
!$OMP END WORKSHARE
!$OMP END PARALLEL

where xnew_pp is a piece of the vector xnew given to each MPI processes (the same goes for x_pp and p_pp). The WORKSHARE directive will assign a chunk of those arrays to each thread and from the MPI process point of view, these operations will be performed in parallel.

• A number of dot products are also used. These are performed locally on each processor and summed up through a function named DOT_PRODUCT_P calling the intrinsic Fortran function DOT_PRODUCT. This can be easily parallelised using the WORKSHARE directive too [11].

5.2.3 Results

When adding OpenMP directives into the code, a very important point is to check that the results are identical to the ones obtained with the original code. These checks led to find out that a parallelised loop around the MATMUL intrinsic functions was surprisingly giving different results. That result yielded to think that the MATMUL command was not thread-safe in that case. This operation was then replaced by a call to the BLAS routine DGEMV which did not reproduce those errors. All results showed in the following are hence obtained with OpenMP directives and original code except for this BLAS call allowing OpenMP parallelisation in the loop.
A medium mesh

The hybrid code is first tested on a mesh of medium size. It contains 125000 elements. We try different thread configurations, using always the same number of cores: 120 processes, 40 MPI processes with 3 threads per process, 20 MPI processes with 6 threads per process, 5 MPI processes with 24 threads per process.

Results are shown in Figure 5.4.

![Figure 5.4: Runtime of the two main sections of the code with different threads configurations](image)

We can see that the hybrid codes are giving relatively similar results on the section where the steering array is computed. The timings are similar to the result obtained using a pure MPI version. There is not any performance improvements. Now, if we look at the section where the equations are solved, we can see that the version using 24 threads per MPI task is performing very poorly. The ones using 6 or 3 threads have performances almost identical to the pure MPI version. In general, we can say that the less threads the better (with 12 threads, we obtain a result not as bad as with 24, but still not as good as with 6 or 3).

We can wonder why there is such a difference between the performance of those two sections of the code when the number of threads is larger than 6. Where the steering array is computed, two loops are parallelised with OpenMP. Where the equations are solved however, the parallel regions are inside an outer loop counting the iterations. The conjugate gradient algorithm converged after 720 iterations in this case, which means that the threads had to be created 720 times since they are spawned at each iterations. We can imagine that this generates overhead.
A larger mesh

The Hybrid code is now tested with a larger mesh of one million elements. It contains around four million nodes and leads to the resolution of twelve million equations. For saving purposes, the code will not be run till completion but stopped after 100 iterations. Below are printed the results obtained with the pure MPI version with 120 processes and the hybrid version with 20 processes and 6 threads per process.

<table>
<thead>
<tr>
<th>Program section execution times</th>
<th>Seconds</th>
<th>%Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>25.434916</td>
<td>5.37</td>
</tr>
<tr>
<td>Compute coordinates and steering array</td>
<td>411.254496</td>
<td>86.86</td>
</tr>
<tr>
<td>Compute interprocessor communication tables</td>
<td>0.403149</td>
<td>0.09</td>
</tr>
<tr>
<td>Allocate neq_pp arrays</td>
<td>0.011345</td>
<td>0.00</td>
</tr>
<tr>
<td>Compute element stiffness matrices</td>
<td>3.090907</td>
<td>0.65</td>
</tr>
<tr>
<td>Build the preconditioner</td>
<td>0.034750</td>
<td>0.01</td>
</tr>
<tr>
<td>Get starting r</td>
<td>0.557387</td>
<td>0.12</td>
</tr>
<tr>
<td>Solve equations</td>
<td>9.252077</td>
<td>1.95</td>
</tr>
<tr>
<td>Recover stresses</td>
<td>0.055498</td>
<td>0.01</td>
</tr>
<tr>
<td>Output results</td>
<td>23.377980</td>
<td>4.94</td>
</tr>
<tr>
<td>Total execution time</td>
<td>473.472505</td>
<td>100.00</td>
</tr>
</tbody>
</table>

Result 3: Timing output from ParaFEM on the large mesh with the Pure MPI version

<table>
<thead>
<tr>
<th>Program section execution times</th>
<th>Seconds</th>
<th>%Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>29.998600</td>
<td>6.24</td>
</tr>
<tr>
<td>Compute coordinates and steering array</td>
<td>396.053972</td>
<td>82.36</td>
</tr>
<tr>
<td>Compute interprocessor communication tables</td>
<td>0.381552</td>
<td>0.08</td>
</tr>
<tr>
<td>Allocate neq_pp arrays</td>
<td>0.028491</td>
<td>0.01</td>
</tr>
<tr>
<td>Compute element stiffness matrices</td>
<td>16.738788</td>
<td>3.48</td>
</tr>
<tr>
<td>Build the preconditioner</td>
<td>0.714807</td>
<td>0.15</td>
</tr>
<tr>
<td>Get starting r</td>
<td>3.131005</td>
<td>0.65</td>
</tr>
<tr>
<td>Solve equations</td>
<td>11.169378</td>
<td>2.32</td>
</tr>
<tr>
<td>Recover stresses</td>
<td>0.245769</td>
<td>0.05</td>
</tr>
<tr>
<td>Output results</td>
<td>22.400960</td>
<td>4.66</td>
</tr>
<tr>
<td>Total execution time</td>
<td>480.863322</td>
<td>100.00</td>
</tr>
</tbody>
</table>

Result 4: Timing output from ParaFEM on the large mesh with the Hybrid version using 6 threads per MPI process

Please note that the "Solve equations section", which has been shortened to 100 iterations for saving time, is taking only around 2% of the total runtime here but is much more important when the code is run to completion.

We first see that the computation of the steering array is a bit faster in the hybrid code than in the pure MPI code. It gives an improvement of about 4%. However, we can
also notice that solving the equations is about 17% slower with the hybrid code. The other sections should not be taken into account since no OpenMP directives have been included in them and they are hence slower on the hybrid version because run with fewer MPI tasks.

The fact that we do get a little speedup on the "steering array" section of the code and a slowing down on the "solve equation" section could be explained by the fact threads have to be spawned at each iteration in the latter, unlike the former where threads are spawned just once. It is then a more favourable and straightforward spot for OpenMP parallelisation.

5.3 Reordering of elements using METIS

Load balance between the processors is ensured by assigning roughly the same number of elements per processor. However, this does not ensure the amount of communications between the processors will be optimal. The way the chunks of elements are assigned can potentially have a dramatic influence on the performance. Communications occurs when 2 elements are contiguous but assigned to 2 different processors. One therefore wants to minimize the neighbouring surfaces between subdomains.

5.3.1 Mesh partition

Preliminary observations

The way the partition of the elements of the mesh is done in ParaFEM is quite naive and is based on the numbering of the elements. On a regular mesh such as the one we used in p121, which is simply a cube, the elements are ordered from one corner, line by line and plane by plane. Figure 5.5 shows 2 subdomains obtained with this decomposition, in the case the cube has to be divided in 10 subdomains. For visibility, a cube is represented by a plane cutting it at its half.
This partition makes sense and sounds reasonable. It is though a particular and favourable case, since each subdomain fits perfectly onto a single plane. In the general case, the partition is likely to produce discontinuous subdomains. In Figure 5.6 are represented two subdomains obtained with the naive ParaFEM method when the domain is partitioned into 48 pieces.

We can see here that some subdomains created are disconnected. This is a phenomenon we would like to avoid, since this involves an increase in communication cost. Another undesirable effect is that the number of neighbours is increased too, since each of the two pieces of the subdomain is in contact with about twice as many neighbours as if the subdomain were connected.
The METIS decomposition

An idea to improve performance is to use a more subtle decomposition, which would minimize the amount of communications. For this, we will use the METIS mesh decomposition, presented in section 2.2. This method uses an advanced recursive bisection technique. It is supposed to create partition minimizing the overall contact surfaces between subdomains, and do its best to limit the number of neighbours of each particular subdomain created.

Figure 5.7 shows what is obtained when using the METIS mesh decomposition on the same problem, with both the 10-subdomain partition and the 48 one.

Looking at those pictures, we can say that METIS generates connected domains, which are of a rather compact shape. We can have some hopes that using these mesh decompositions will improve the performance.
5.3.2 Implementation

The mesh is given to ParaFEM through a file containing the coordinates of the nodes in the first part and the elements in the second part. The order of the elements in this file decides the partition used by ParaFEM: the first process will get the first chunk of the elements written, the second one, the second chunk and so on.

The METIS partitioner takes as input a file containing the elements of the mesh. Those have to be written in a specific format which is no the one used in ParaFEM. A function reading the ParaFEM mesh file and converting it to the METIS format was hence implemented. This file is then given as an input to the METIS partitioner, which provides a file containing the partition. Another function was implemented to read this output file and generate a new ParaFEM mesh file taking the partition into account. In this new file, the elements are written partition by partition, so that when read into ParaFEM, the partitions are taken into account.

A diagram is displayed in Figure 5.8.

![Figure 5.8: Process followed to get a mesh input file to ParaFEM that is obtained using METIS](image-url)
5.3.3 Results and further analysis

Results

We first try this on a cubic mesh containing 125000 elements (50 x 50 x 50). We will try out two different configurations, one with 24 cores on 24 partitions and one with 480 cores on 480 partitions. The first one almost corresponds to a similar decomposition as shown in Figures 5.7(a) and 5.7(b): for the naive decomposition, each plane will be a subdomain (it would have to be 25 subdomains for this to be exact, but this is quite close). The second decomposition will more correspond to a configuration of the type shown in Figures 5.7(c) and 5.7(d), less favourable to the naive decomposition. Results are displayed in Table 5.3. Only the timings for the two most important part of the code are shown.

<table>
<thead>
<tr>
<th>Execution time</th>
<th>Naive decomposition</th>
<th>METIS decomposition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>24 partitions</td>
<td>480 partitions</td>
</tr>
<tr>
<td>Compute steering array</td>
<td>58.5</td>
<td>3.0</td>
</tr>
<tr>
<td>Solve equations</td>
<td>150.8</td>
<td>8.4</td>
</tr>
</tbody>
</table>

Table 5.3: Specific timings on the "solve equations" section. A million element mesh, 120 processors, stopped after 10 iterations

They are several things to learn from this table.

- The time spent to compute the steering array does not seem to depend on whether the naive decomposition or the METIS decomposition has been used: the partition of the elements has no effect on this section of code. This result was actually predictable. Indeed, this section of code does not involve communications between processors. It hence does not matter how the elements are assigned to each processors since they won’t have to communicate and will just perform their calculation locally. They are no element more expensive than others. However, the "solve equation" section do involve quite a few communications and it is not surprising to notice a timing difference when changing the partition.

- For the 24 partition test, we can see that the METIS result is worse than the naive one (by around 5%). This is not surprising since this partition was of a type favourable to the naive ParaFEM partition.

- For the 480 partition test, the METIS decomposition is giving much worse results than the naive one. The execution time of the "solve equation" section is more than three times longer. This is a rather surprising result since we were expecting the METIS partition to be of much better quality than the naive one on this type of configuration.
Further analysis

To get more understanding of those results, we will check what is the quantity of data exchanged between processors, and how many neighbours each processor has. This is done in a quite straightforward way putting printouts of appropriate quantities within the code. Table 5.4 gives those results.

<table>
<thead>
<tr>
<th>Execution time</th>
<th>Naive decomposition</th>
<th>METIS decomposition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>24 partitions</td>
<td>480 partitions</td>
</tr>
<tr>
<td>Number of bytes exchanged in total</td>
<td>4.5MB</td>
<td>23.3MB</td>
</tr>
<tr>
<td>Median number of neighbours per process</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 5.4: Communication information depending on the mesh partition method

Now, we can see that in the 24-partition configuration, there is less data transferred when using the naive decomposition. In the 480-partition configuration, this amount is smaller when using the METIS decomposition. However, the number of neighbours for each process is much larger. This can explain the poor performance.

To be completely sure about this phenomenon, we use the scalasca profiling tool on the run using the METIS decomposition and 480 processors. Time spent in some of the key routines to understand the performance is printed below.

<table>
<thead>
<tr>
<th>flt type</th>
<th>max_tbc</th>
<th>time</th>
<th>% region</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY</td>
<td>8558536</td>
<td>17135.95</td>
<td>100.00 (summary) ALL</td>
</tr>
<tr>
<td>MPI</td>
<td>8074648</td>
<td>14258.17</td>
<td>83.21 (summary) MPI</td>
</tr>
<tr>
<td>COM</td>
<td>169800</td>
<td>1506.57</td>
<td>8.79 (summary) COM</td>
</tr>
<tr>
<td>USR</td>
<td>317496</td>
<td>1243.96</td>
<td>7.26 (summary) USR</td>
</tr>
<tr>
<td>MPI</td>
<td>3183400</td>
<td>6546.33</td>
<td>38.20 MPI_Isend</td>
</tr>
<tr>
<td>MPI</td>
<td>2929372</td>
<td>22.35</td>
<td>0.13 MPI_Recv</td>
</tr>
<tr>
<td>MPI</td>
<td>1527120</td>
<td>1697.50</td>
<td>9.91 MPI_Probe</td>
</tr>
<tr>
<td>MPI</td>
<td>141440</td>
<td>1856.09</td>
<td>10.83 MPI_Barrier</td>
</tr>
<tr>
<td>MPI</td>
<td>141080</td>
<td>379.22</td>
<td>2.21 MPI_Allreduce</td>
</tr>
<tr>
<td>USR</td>
<td>84864</td>
<td>3.05</td>
<td>0.02 elap_time</td>
</tr>
<tr>
<td>MPI</td>
<td>77160</td>
<td>2111.01</td>
<td>12.32 MPI_Bcast</td>
</tr>
<tr>
<td>COM</td>
<td>50760</td>
<td>12.60</td>
<td>0.07 dot_product_p</td>
</tr>
<tr>
<td>MPI</td>
<td>38520</td>
<td>24.62</td>
<td>0.14 MPI_Reduce</td>
</tr>
<tr>
<td>MPI</td>
<td>33960</td>
<td>58.51</td>
<td>0.34 MPI_Waitall</td>
</tr>
<tr>
<td>COM</td>
<td>16968</td>
<td>32.66</td>
<td>0.19 gather</td>
</tr>
<tr>
<td>COM</td>
<td>16944</td>
<td>60.38</td>
<td>0.35 scatter</td>
</tr>
<tr>
<td>USR</td>
<td>6456</td>
<td>1237.39</td>
<td>7.22 find_g</td>
</tr>
<tr>
<td>MPI</td>
<td>24</td>
<td>1482.17</td>
<td>8.65 MPI_Finalize</td>
</tr>
</tbody>
</table>
We can see than 83% of the runtime is spent in the MPI routines which is really high. When looking at the routine times, it is obvious that the MPI routine MPI_Isend is responsible for a huge chunk of the runtime. This routine is called in the gather and scatter routines performing data exchanges. It is now quite clear that processor are spending a lot of time sending many small messages instead of fewer larger messages.

This results shows that when considering a mesh decomposition, minimising the neighbouring surfaces (to minimize communications) is sometimes not the only relevant criterion: the number of neighbours that each processor have can affect significantly the performance.
Chapter 6

Conclusions and Future work

6.1 Conclusions

The aim of this project was to test and analyse the performance of the ParaFEM package, focusing on one of the programs (named p121), and try out some different optimisations. Tests were conducted on both HECToR Cray XT4 and XT6 machines. Performance has been analysed by the use of pure timings of different sections of the code and using some profiling tools. The optimisations carried out were replacing the Fortran array operations by calls to the BLAS routines, implementing a hybrid OpenMP-MPI version of the code and using a mesh-partitioner expected to divide the work between processes in an efficient way.

By using the timing output from the code, it was seen that for a reasonably large problem, there are two main sections of the code taking most of the execution time: the section where the steering array is computed and the section where the conjugate gradient algorithm is used to solve the equations. Using the profiling tools, we noticed that where the steering array is computed, there are no communications involved. However where the equations are solved, quite a few communications are carried out. The scaling of the code is reasonably good, and gets better when the problem considered is large. The "steering array computation" section is scaling almost perfectly due to the fact it involves no communications between processors. The "solve equation" section has a reasonable scaling (limited by the communications it carries out), as long as the number of elements per processor is high enough. This section is dominating and responsible for the overall scaling of ParaFEM.

A first optimisation that was tempted was to replace Fortran intrinsic function calls or array manipulation by calls to the BLAS library routines. This was done in the part of code applying the conjugate gradient algorithm (one of the two most demanding part of the code). Results showed that no significant difference in performance occurred. The flop rate achieved was relatively small (around 1% or 2%). When timing specific BLAS call lines, flop rate obtained was still poorer than expected. This result is possibly due to complicated cache alignment issues.
A second optimisation developed was the implementation of a Hybrid OpenMP-MPI version of the code, adding some OpenMP directives where appropriate. Several combinations of threads and MPI tasks were tried out, and the one using 4 MPI tasks per node and 6 threads spawned by each task seemed to be the most reliable and efficient one (the one using 8 MPI tasks and 3 threads per task gave similar performance). Using this configuration on a large problem showed that there can be a potential speedup on sections of the code where no or few communications are involved and the OpenMp parallelisation is simple. Where many communications are involved (such as in the "solve equation" section), OpenMP parallelisation can only be introduced piece-by-piece between those communications and possible OpenMP overhead can happen by the fact fork-and-join of threads happen constantly.

The last optimisation tested was to replace the naive mesh decomposition used in ParaFEM by a smarter one (METIS), trying to minimize the overall communications between processors by minimising the surface of neighbouring subdomains. The naive mesh decomposition can be of good quality on some favourable cases but is likely to give disconnected subdomains in the general case. Tests showed that for a general test case, the METIS partition involves much less communications than the naive partition. However, the performance of the latter was still much better than the former. In a more detailed analysis, we notice that despite the fact the overall contact surface between subdomains is smaller when using METIS, each subdomain has a number of neighbouring subdomains which is much higher than when using the naive method. Each process then has to send smaller messages but many of them. The conclusion was that in mesh partition, it is not only the area of the contact surfaces between subdomains that matters: the number of neighbours that each subdomain has can be critical.

6.2 Future work

There are a number of ways in which future work could be done.

Some further analysis could be carried out to understand why the flop rate of the BLAS routines is poor when called in ParaFEM.

The hybrid code could be optimised. There may be some way to parallelise the "solve equation" section in a more subtle way than how it is currently done and then involve less OpenMP overhead. The code in this section could be rewritten in a way enabling an efficient OpenMP implementation.

Regarding the mesh decomposition, some extra tests could be carried out. All the tests done here were using regular cubic meshes, where the naive mesh partition was never really terrible. It would be interesting to see how the naive partition looks like on a more complicated domain. The METIS mesh partition might be beneficial in those cases.
Bibliography


[5] HECToR Phase2b Good practice guide. Available at: http://www.hector.ac.uk/cse/documentation/Phase2b/

[6] Preparing for HECToR Phase2 guide. Available at: http://www.hector.ac.uk/cse/documentation/Phase2/  

[7] HECToR webpage. Available at: http://www.hector.ac.uk/service/hardware/


[13] Introduction to Finite Element Methods course from the university of colorado. Available at:http://www.colorado.edu/engineering/cas/courses.d/IFEM.d/

[15] Parafem Homepage. Available at: http://www.rcs.manchester.ac.uk/research/Parafem

[16] ParaFEM project page. Available at: http://code.google.com/p/parafem/

[17] Partnership for advance computing in europe website. Available at: 
    http://www.prace-project.eu/hpc-training/training_pdf/cpw09_compilers.pdf