Parallel FFT Libraries

Evangelos Brachos

September 1, 2011

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2011
Abstract

The focus of this project is the area of the fast Fourier transform (FFT) libraries (FFTW 3.3, P3DFFT 2.4 and 2DECOMP&FFT 1.1.319). During this project we benchmark the libraries on Hector phase 2b supercomputer. The main goal of this project is to examine the performance of the libraries which are built on the top of the FFTW’s latest version and compare their results with the FFTW itself. Moreover, the ability of the P3DFFT and the 2DECOMP&FFT to implement the 2D decomposition, let us experiment with the scaling of these libraries in thousands of cores. The findings from this experiment provide evidence of the FFTW library’s excellent scaling by using the 1D decomposition and the ability of the other two libraries to minimise the time required for the Fourier transform computation, by using the 2D decomposition. The main conclusions drawn from this study were the ability of the P3DFFT and the 2DECOMP&FFT libraries to scale up to thousands of cores and the affects of the Gemini Interconnect on their performance.
# Contents

1 Introduction 1

2 Background Theory 4

2.1 Fourier Analysis 4

2.2 Fourier Transform 4

2.3 Fourier Series 5

2.3.1 Discrete Fourier Transform (DFT) 5

2.3.2 Fast Fourier Transform (FFT) 6

2.4 Parallel FFT in Three Dimensions 6

2.4.1 Parallelisation of the 3D FFT 7

2.4.2 1D Decomposition 7

2.4.3 2D Decomposition 8

2.5 FFT Libraries 9

2.5.1 Parallel FFT Libraries 10

2.5.2 FFTW 3.3 10

2.5.3 P3DFFT 12

2.5.4 2DECOMP&FFT 14

2.5.5 Features Comparison 15

3 Design and Implementation 16

3.1 Design 16

3.2 Implementation 17

3.2.1 Programming Languages 18

3.2.2 Compilation 18

3.2.3 FFT Libraries Installation 18

3.2.4 Test Data 20

3.2.5 Test Programs 20

3.2.6 Scripts 22

4 Experimental Design 24

4.1 HPC Systems Architecture 24

4.1.1 Hardware 24

4.1.2 Interconnection Network 26

4.1.3 Compiler 26

4.2 Experiments Setup 27
4.2.1 Testing Different Decompositions ........................................... 27
4.2.2 Reporting ............................................................................... 28

5 Results and Performance Analysis ............................................. 30
  5.1 Slabs and Pencils on Hector ...................................................... 30
    5.1.1 1D Decomposition (Slab) .................................................... 30
    5.1.2 1D Decomposition Performance Analysis ............................ 35
    5.1.3 2D Decomposition (Rods or Pencils) ................................. 38
    5.1.4 Processors Grid Selection .................................................. 39
    5.1.5 2D Decomposition Performance ......................................... 41
    5.1.6 2D Decomposition Performance Analysis ............................ 44
  5.2 Slabs versus Pencils on Hector ............................................... 45

6 Conclusions & Discussions .......................................................... 50
  6.1 Conclusions ........................................................................... 50
  6.2 Discussions ............................................................................. 51
  6.3 Future Work ........................................................................... 52

A Results ........................................................................................ 54
  A.1 1D and 2D decomposition results .......................................... 54
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Stages During the 2D Decomposition</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>FFTW, P3DFFT and 2DECOMP&amp;FFT features.</td>
<td>15</td>
</tr>
<tr>
<td>4.1</td>
<td>Hector Hardware Features</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>Gemini Interconnect on Hector Phase 2b</td>
<td>26</td>
</tr>
<tr>
<td>4.3</td>
<td>1D decomposition test parameters</td>
<td>27</td>
</tr>
<tr>
<td>4.4</td>
<td>1D decomposition test parameters</td>
<td>28</td>
</tr>
<tr>
<td>5.1</td>
<td>SLAB: 256$^3$, 1 core per node, time and speedup</td>
<td>31</td>
</tr>
<tr>
<td>5.2</td>
<td>SLAB: 256$^3$, 16 cores per node, time and speedup</td>
<td>32</td>
</tr>
<tr>
<td>5.3</td>
<td>SLAB: 128$^3$, 1 core per node, time and speedup</td>
<td>33</td>
</tr>
<tr>
<td>5.4</td>
<td>SLAB: 128$^3$, 16 cores per node, time and speedup</td>
<td>34</td>
</tr>
<tr>
<td>5.5</td>
<td>SLAB: 256$^3$, FFTW OpenMP version versus MPI in one node</td>
<td>35</td>
</tr>
<tr>
<td>5.6</td>
<td>256$^3$: 1D &amp; 2D decomposition results</td>
<td>46</td>
</tr>
<tr>
<td>5.7</td>
<td>128$^3$: 1D &amp; 2D decomposition results</td>
<td>48</td>
</tr>
<tr>
<td>5.8</td>
<td>128$^3$: 2D decomposition results for 512 Hector’s nodes</td>
<td>48</td>
</tr>
<tr>
<td>A.1</td>
<td>PENCIL: 256$^3$, 16 cores per node, time and speedup</td>
<td>56</td>
</tr>
<tr>
<td>A.2</td>
<td>PENCIL: 256$^3$, 1 core per node, time and speedup</td>
<td>57</td>
</tr>
<tr>
<td>A.3</td>
<td>PENCIL: 128$^3$, 1 core per node, time and speedup</td>
<td>57</td>
</tr>
<tr>
<td>A.4</td>
<td>PENCIL: 128$^3$, 16 cores per node, time and speedup</td>
<td>58</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>2D decomposition of 3D FFT</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>2D decomposition of the P3DFFT</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>FFT benchmarking methodology</td>
<td>17</td>
</tr>
<tr>
<td>5.1</td>
<td>SLAB: Speedup using $256^3$ 3D FFT with 1 core per node</td>
<td>31</td>
</tr>
<tr>
<td>5.2</td>
<td>SLAB: Speedup using $256^3$ 3D FFT with 16 cores per node</td>
<td>32</td>
</tr>
<tr>
<td>5.3</td>
<td>SLAB: Speedup using $128^3$ 3D FFT 1 core per node</td>
<td>34</td>
</tr>
<tr>
<td>5.4</td>
<td>Libraries performance using $128^3$ 3D FFT 16 cores per node</td>
<td>35</td>
</tr>
<tr>
<td>5.5</td>
<td>SLAB: Speedup using $128^3$ 3D FFT 16 cores per node</td>
<td>36</td>
</tr>
<tr>
<td>5.6</td>
<td>Libraries performance using $256^3$ 3D array, 1024 total cores</td>
<td>40</td>
</tr>
<tr>
<td>5.7</td>
<td>PENCIL: Speedup using $256^3$ 3D FFT 1 and 16 cores per node</td>
<td>42</td>
</tr>
<tr>
<td>5.8</td>
<td>PENCIL: Speedup using $128^3$ 3D FFT 1 and 16 cores per node</td>
<td>43</td>
</tr>
<tr>
<td>5.9</td>
<td>Speedup using $256^3$ 3D FFT 1-16 cores per node</td>
<td>47</td>
</tr>
<tr>
<td>5.10</td>
<td>Speedup using $128^3$ 3D FFT 1-16 cores per node</td>
<td>49</td>
</tr>
<tr>
<td>A.1</td>
<td>SLAB: Libraries performance using $256^3$ 3D FFT with 1 core per node</td>
<td>54</td>
</tr>
<tr>
<td>A.2</td>
<td>SLAB: Libraries performance using $256^3$ 3D FFT, 16 cores per node</td>
<td>55</td>
</tr>
<tr>
<td>A.3</td>
<td>SLAB: Libraries performance using $128^3$ 3D FFT, 1 core per node</td>
<td>55</td>
</tr>
<tr>
<td>A.4</td>
<td>OpenMP performance versus the MPI implementations up to 16 cores - $256^3$ 3D FFT</td>
<td>56</td>
</tr>
<tr>
<td>A.5</td>
<td>PENCIL: Libraries performance using $128^3$ 3D FFT, 1 core per node</td>
<td>58</td>
</tr>
<tr>
<td>A.6</td>
<td>PENCIL: Libraries performance using $128^3$ 3D FFT, 16 cores per node</td>
<td>59</td>
</tr>
<tr>
<td>A.7</td>
<td>PENCIL: Libraries performance using $256^3$ 3D FFT, 1 core per node</td>
<td>59</td>
</tr>
<tr>
<td>A.8</td>
<td>PENCIL: Libraries performance using $256^3$ 3D FFT, 16 cores per node</td>
<td>60</td>
</tr>
<tr>
<td>A.9</td>
<td>Libraries performance using $256^3$ 3D FFT, 512 total cores</td>
<td>60</td>
</tr>
</tbody>
</table>
Acknowledgements

I would like to thank my supervisor Dr. Stephen Booth for the help and guidance he has given me. I would also like to thank Dr. Ning Li for all his assistance and the valuable resources he has sent to me. Finally, I would like to thank my parents for their encouragement.
Chapter 1

Introduction

Fast Fourier transform (FFT) libraries are used by many researchers in many scientific areas and industries. The Fast Fourier Transform has been one of the most popular numerical methods which is applied in almost every field of science such as oil exploration, signal processing, Monte Carlo simulations, molecular dynamics and many others.

The computation of the fast Fourier transform is a very demanding operation and requires a lot of time to complete depending on the size of the problem and on many other issues which will be presented on the following pages of this report. The fast Fourier transform computation can be executed much faster using parallel processing, especially nowadays, as many supercomputing facilities are available to scientists across the world. Furthermore, the multicore desktop computers offer the capability of parallel processing.

Every word in the result of an FFT depends on every input word. A parallel implementation of the FFT therefore requires significant number of communications.

The scientific computation of the fast Fourier transform algorithm could be implemented by using one of the many available FFT libraries which, in turn, make use of different implementations of the FFT.

In order to improve the performance of the fast Fourier transform computation it is very important to carefully choose the right FFT library. Sometimes the correct choice of a library seems to improve the program performance and reduce the computation time. However, that choice depends on the hardware architecture, on the nature of the
problem (type of data), on the implementation (sequential or parallel) and on the budget (some libraries are open source while some other are commercial).

Most of the FFT libraries offer parallel computation support using different programming models such POSIX [1], MPI [2], OpenMP [3], CUDA [4] and others (either CPU or GPU programming models). This capability decreases the computational time of the Fourier transform.

The performance of the FFT parallel computation poses a large number of issues regarding the scalability and some other aspects which sometimes become a bottleneck, either physical (interconnection network) or of a design nature (demanding communication of the fast Fourier transform). All of these issues, prevent the scientific programs which use FFT to scale up to a large number of processors.

Thus, it is very important to know the performance of different FFT libraries in the parallel FFT computation. Such knowledge could facilitate the scientists to know the scaling of their programs which compute the fast Fourier transform of a 3D mesh with certain dimensions by using one of the available FFT libraries. This will help reduce the financial costs required for deployment of resources in order to compute the 3D FFT in parallel.

On this project an extended benchmarking procedure implemented comparing the test programs performance which compute the 3D FFT by using the FFTW 3.3 [5] library, the P3DFFT [6] 2.4 and 2DECOMP&FFT [7] 1.1.319 which are built on the top of the FFTW 3.3. The performance analysis was conducted on Hector [8] phase 2b.

Outline

In the following pages an analytical description of the FFT libraries, used for this project, is made along with the implemented test programs and benchmarking procedure.

In particular the report is divided into six chapters:

Chapter 1 In this chapter we introduce briefly the fast Fourier transform usage and the existence of the FFT libraries.

Chapter 2 In this chapter we presented all the mathematical background of the problem and the FFT libraries features.
Chapter 3 This chapter is referring to the design and analysis methodology which implemented by such projects and the steps we made to implement this project.

Chapter 4 This chapter is referring to the HPC systems architectures by analysing the supercomputer’s software and hardware. There is also a discussion about the tests we made along with the combinations of the arguments we used.

Chapter 5 We made an extended presentation of the obtained results. After the results presentation of each decomposition there is an analysis for the particular decomposition results. At the end of this chapter there is a comparison between the two decompositions approaches.

Chapter 6 In this chapter we interpreted the results and we made conclusions about the different libraries and the different decompositions. Then follows a section where we presented some suggestions in order to give some directions about further improvement of this project in the future.

Appendix At the end of the report we presented some of the most important obtained results by using tables and graphs.
Chapter 2

Background Theory

2.1 Fourier Analysis

Joseph Fourier proved that any continuous function could be produced as an infinite sum of simpler trigonometric functions.

2.2 Fourier Transform

Fourier transform is part of Fourier analysis and it is a general form for a continuous and aperiodic time signal $f(k)$.

The Fourier transform is a mathematical method that decomposes a signal into the series of frequencies that compose the time series. Forward Fourier Transform:

$$F(k) = \int_{-\infty}^{\infty} f(x)e^{-2\pi ikx} dx, \ \forall k \in \mathbb{R}$$

Inverse Fourier Transform:

$$f(x) = \int_{-\infty}^{\infty} F(k)e^{2\pi ikx} dk, \ \forall k \in \mathbb{R}$$

The basis of Fourier transform is the Euler’s formula $e^{i\varphi} = \cos \varphi + i\sin \varphi$. A complex exponential is a complex number where the real and the imaginary parts are sinusoids.

For two dimensions the Fourier transform becomes:

$$F(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(k_x, k_y)e^{-2\pi i(k_xx + k_yy)} dk_x dk_y$$
For two dimensions the inverse Fourier transform becomes:

\[ f(k_x, k_y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} F(x, y)e^{2\pi i (k_x x + k_y y)} \, dx \, dy \]

The n-dimensional Fourier transform can be defined for \( k, x \) in \( \mathbb{R}^n \) by

\[
F(x) = \left\{ \begin{array}{l}
\int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} f(k)e^{-2\pi i k x} \, d^n k
\end{array} \right.
\]

\[
f(k) = \left\{ \begin{array}{l}
\int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} F(x)e^{2\pi i k x} \, d^n x
\end{array} \right.
\]

### 2.3 Fourier Series

Fourier series can be generalized to complex numbers and further generalized to derive the Fourier Transform. In Fourier analysis a signal could be represented as a series of sines and cosines.

\[
f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(nx) + \sum_{n=1}^{\infty} b_n \sin(nx), \quad \forall \, n \in \mathbb{N}^*
\]

where:

\[
a_0 = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \, dx, \quad \forall \, n \in \mathbb{N}^*
\]

\[
a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} \cos(nx) \, dx, \quad \forall \, n \in \mathbb{N}^*
\]

\[
b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} \sin(nx) \, dx, \quad \forall \, n \in \mathbb{N}^*
\]

#### 2.3.1 Discrete Fourier Transform (DFT)

In order to find the frequency spectrum of a sampled function the discrete Fourier transform is used instead of Fourier transform.

Forward DFT:

\[
F_n = \sum_{k=0}^{N-1} f_k e^{-\frac{2\pi i kn}{N}}, \quad k = 0, \ldots, N - 1
\]

\( f_0 \ldots f_N \in \mathbb{C} \) are transformed into \( F_0 \ldots F_N \in \mathbb{C} \).
Inverse DFT:

\[ f_k = \frac{1}{N} \sum_{n=0}^{N-1} F_n e^{2\pi i kn/N}, \quad \forall N \in \{1, \ldots, N - 1\} \]

\(F_0, \ldots, F_N \in C\) are transformed into \(f_0, \ldots, f_N \in C\).

### 2.3.2 Fast Fourier Transform (FFT)

Fast Fourier transforms are of primary importance in computationally intensive applications. They are used for large-scale data analysis in order to solve partial differential equations. The FFT is the most frequently used algorithm today in the analysis and in operations on digital data.

There are many fast Fourier transform algorithms which can in fact compute the DFT in \(N \log(N)\) operations rather than the \(N^2\) operations required from the DFT. The fast Fourier transform algorithm was introduced by J.W. Cooley’s and J.W. Tukey’s [9]. In their paper they described a model in order to compute the Fourier coefficients of time series using fewer operations compared to the usual procedure. Since then, most of the FFT algorithms have been based on the Cooley-Tukey algorithm.

In 1942, Danielson and Lanczos [10] introduced an algorithm derivation. Their approach the initial DFT points are split into even and odd points and the length of each one of the two DFTs, is equal to \(N/2\).

### 2.4 Parallel FFT in Three Dimensions

Using the definition of the Fourier transform, the formula (2.1) is applied on a 3D mesh, with \(L \times M \times N\) dimensions consists of complex numbers.

\[
\sum_{x=0}^{L-1} \sum_{y=0}^{M-1} \sum_{z=0}^{N-1} B_{x,y,z} e^{-2\pi i \frac{wx}{L}} e^{-2\pi i \frac{wy}{M}} e^{-2\pi i \frac{wz}{N}}
\]

Stage 1: 1D Fourier transform along the \(z\)-dimension.

The computation of the formula (2.1) can be implement in 3 stages, which can be used to implement a parallel FFT computation in 3 dimensions. On each one of these 3 steps, a 1 dimensional fourier transform is implemented in each dimension as follows:

**Stage 1:** 1D Fourier transform along the \(z\)-dimension.
Stage 2: 1D Fourier transform along the $y$-dimension.

Stage 3: 1D Fourier transform along the $x$-dimension.

Obviously, the parallel implementation of 1D FFT affects the performance of the distributed FFT. Usually, in the case of the 2D FFTs the performance relies on the 1D FFT algorithms which applied to transform the data on one dimension.

### 2.4.1 Parallelisation of the 3D FFT

There are two transport-based FFTs available to the scientists, which can be used in order to compute the Fourier transform of 3D meshes. The first and the most common through the years, is the 1D decomposition. The other is the 2D decomposition which on higher number of processors it becomes recently for decompose over the 2 dimensions [11], [12], [13].

### 2.4.2 1D Decomposition

With a given problem of $N \times N \times N$ grid distributed over $N_p$ processors, it is needed to do one dimensional FFT in each of the three dimensions and the data should not be local to any of the processors.

Two approaches exist in order to solve the problem. The first approach is to implement a parallel version of 1D FFT where a number of processors take part and communicate when needed. The other approach, which requires less overall communication, is to make the data local for the dimensions to be transformed.

The main characteristic of both approaches is the communication overhead. The second approach is the most well known and the main idea is the data rearrangement.

On this algorithm each processor has several slabs ($N \times N \times \frac{N}{P_{processors}}$). A library such as the FFTW which implements 1D FFT the MPI_Alltoall() operation is used in order to transpose the mesh after the completion of the 2D FFT. After that, the data is transposed to localize the $3^{rd}$ dimension. Finally, the transform is completed by operating along the $3^{rd}$ dimension, on which a 1D FFT is applied.

### Scaling

The scaling performance of this kind of 1D decomposition strategy depends on the interconnection bisectional bandwidth. The major aspect is the limit factor $P \leq N$ (i.e.
<table>
<thead>
<tr>
<th>Stage</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$N \times N \times N$</td>
</tr>
<tr>
<td>2</td>
<td>$\frac{N}{P_1} \times N \times \frac{N}{P_2}$</td>
</tr>
<tr>
<td>3</td>
<td>$\frac{N}{P_1} \times \frac{N}{P_2} \times N$</td>
</tr>
</tbody>
</table>

Table 2.1: Stages During the 2D Decomposition

for a $256 \times 256 \times 256$ 3D mesh the maximum number of tasks is 256).

2.4.3 2D Decomposition

In 2D decomposition the 2D mesh is distributed over $P = P_{row} \times P_{col}$ tasks and the processors form columns instead of slabs. The dimensions of these columns is $N_x \times \frac{N_y}{P_{row}} \times \frac{N_z}{P_{col}}$.

The 2D decomposition of a 3D array $A_{L \times M \times N}$ illustrated in the figure 2.1 is discussed in detail on [13].

![Figure 2.1: 2D decomposition of 3D FFT.](image)

The stages of the parallelisation, presented in table 2.1 is discussed in great detail in [11] and [13], where the reader could be referred for more information.

Scaling

The scaling limit factor for the 2 dimensional decomposition strategy is $P \leq N^2$ (i.e. for a given $256 \times 256 \times 256$ the limit is $256^2 = 65536$ (maximum number of processors)).
This limit factor gives the opportunity to programs which use 2D decomposition to make use of the available thousands of cores of the modern supercomputers.

2.5 FFT Libraries

There are many FFT libraries provided by CPU & GPU vendors and the Open-Source community. Each one of these libraries has its own features and some of them have similarities. The vast majority of the fast Fourier transform libraries are usually described in high level programming languages such as C and Fortran.

Several universities and research centres have implemented different FFT libraries which support either 1D or 2D decomposition models. Some of the best known libraries are the FFTW library, the IBM’s ESSL library, the AMD’s ACML and the Intel’s IPP and MKL.

Moreover, many other libraries, such as the P3DFFT and the 2DECOMP&FFT, have been developed recently which will also be reported in this report as well as other libraries. The majority of them support real to complex and complex to real and some of them complex to complex data transforms.

In addition, the development of the general-purpose computation on graphics processing units (GPGPU), combined with the graphics processor unit’s power, has led to the implementation of libraries which reflect to the GPU programming models.

Most of these libraries are in early stages and the demanding communication of the FFTs does not allow them to achieve very good performance, but in the following years their efficiency is expected to improve. Such libraries are the CuFFT provided by Nvidia, the GPUFFTW, the NukadaFFT and the DiGPUFFT which is based on the P3DFFT library.

On this project, we examined the FFTW 3.3, the P3DFFT and the 2DECOMP&FFT libraries. The two last libraries support 2D decomposition and they are built on top of the FFTW 3.3 in order to have a fair comparison.

In the following sections, there is a detailed description of the libraries which are being used along with a description of their functionality. Finally, there is also a section which presents the features and the capabilities of those libraries.
2.5.1 Parallel FFT Libraries

Most FFT libraries provide implementations which allow execution on parallel environment (for example the FFTW). The parallelisation can be achieved by using the messaging passing interface or the multi-threaded implementations (or both).

2.5.2 FFTW 3.3

Fastest Fourier Transform in the West (FFTW) library is one of the most popular cross-platform FFT libraries written in C programming language and it is not tuned to a specific vendor machine. The flexibility is one of the key factors in FFTW’s success in addition to its performance.

The FFTW package was developed at MIT [25] by Matteo Frigo and Steven G. Johnson and its latest version is the FFTW 3.3. In fact, FFTW is probably the most flexible DFT library available, for real and complex fast Fourier transforms of any dimension. One unique aspect of the FFTW is the optional use of self-optimizing strategies (these optimisations done by the library at start only).

FFTW includes three parallel transform implementations:

- A second set of routines utilizes shared-memory threads for parallel multi-dimensional transforms. The 3.3 version of the library can perform FFTs in parallel through OpenMP.
- A multi-dimensional transforms implementation for parallel machines supporting MPI. The main feature of this part of the implementation is the support of distributed-memory transforms.
- An implementation written in Cilk [26] available for several SMP platforms.

FFTW3.3 design

FFTW 3.3 provides competitive performance comparing to the other FFT libraries as presented in the FFTW.org website [27] and in chapter 5 of this report. The flexibility of the FFTW in computing the DFT comes from the ability to adopt its functions in different hardware architectures by using the planner component.

Initially, the FFTW’s planner starts making calculations, investigating the fastest configuration for the DFT computation on a given hardware and at the end of this step it produces the problem’s plan which contains all the required details such as the input
array shape & size and the memory layout. Then, the plan is executed in order to transform the input data. A recomputation of the plan can be invoked at anytime during the computations [28].

Generally, the FFTW’s planner helps the scientists because they do not need to do hand tune on their programs. However, the trade-off is that sometimes the planner computation tends to be time consuming and as a result we have reduced performance. In addition, there is an functionality for the advanced FFTW users to manually tune the FFTW.

**Codelet Generator**

FFTW provides the *genfft* compiler (written in the Caml Light dialect of ML [29]) which automatically produces the codelets. The *genfft* codelet generator is a fast Fourier transform compiler and produces codelets (in C programming language), which implement an optimised combination [28] of the Cooley-Tukey algorithm and other discrete Fourier transform algorithms with similar formula.

The codelet generator operates in four phases: a) the creation b) the simplification c) the scheduling and d) the unparsing. In general, when the plan’s sub-problems become simple the code produced by the generator gives a direct solution. The paper by M.Frigo [30] describes the *genfft* and stages it operates in detail.

**Real Data Transforms**

On this project, all the test programs used as a 3D array storing the real data. It is important to briefly outline how the FFTW computes a real DFT, as it described in section VII of the [28]. The FFTW:

- Uses the Hermitian symmetry in order to improve the storage and the performance.
- For simple sub-programs of real data the *genfft* produces codelets which automatically solve the problem by using the complex algorithm [28].
- A fast mixed-radix Fourier algorithm variation [31] is used in order to reduce the computations of the Cooley-Tukey’s algorithm [32], [33], [34].
- For prime sizes, an adaption of the Rader’s algorithm [35] is used.
New SIMD Features on FFTW3.3

The codelets design gives the opportunity to the FFTW 3.x library to support the SIMD (Single-Instruction Multiple Data) instructions in new codelets. As a result, the latest FFTW version offers functionality for the latest SIMD hardware.

FFTW3.3 Scaling

As it was mentioned in section 2.4.2, where the 1D decomposition was described, the FFTW library implements 1D decomposition so that the maximum number of processors that can be used is specified by the limit factor of $P_{\text{processors}} \leq N_{\text{dimension}}$.

This upper bound is a limiting factor for the use of the FFTW. However, many libraries which use the 2D decomposition are built on the top of the FFTW in order to implement the computations.

2.5.3 P3DFFT

The Parallel Three-Dimensional Fast Fourier Transforms library (P3DFFT) is an open source library. The main author is Dmitry Pekurovsky and the library started as a project at University of California, San Diego [36].

P3DFFT library supports both 1D and 2D decomposition and it can be used in large-scale calculations because of the large number of tasks that can execute in parallel.

P3DFFT design

The latest version of the library is the 2.4 which can be obtained from the library’s web page in Google Code [37]. The library requires to be built on the top of an optimised 1D FFT library (IBM’s ESSL or FFTW) and supports 2D decomposition strategy in order to compute the 3D FFT. The library’s interface is written in Fortran 90 using the Message Passing Interface and it can be used by the programs, by calling the library’s Fortran 90 module.

The P3DFFT implements a 2D decomposition into a 2D processors grid $P_1 \times P_2$ and it can compute the forward and then the backward transform. When the $P_1$ is equal to 1, a 1D decomposition takes place.

The memory requirements of the library is three times the size of the input array. In particular, it is the sum of the input and the output arrays plus one for the buffer.
The P3DFFT library uses a 2D domain decomposition by performing 1D FFT in each dimension. Figure 2.2 illustrates the 2D decomposition which is implemented by the library. There are three stages of execution. In the first stage the Y and Z dimensions of a 3D mesh with dimensions X, Y and Z, split amongst the processors in rows and columns of the two dimensional physical processors grid.

On the second stage by using MPI_alltoall a transpose phase starts where the X dimension split amongst the row processors. Finally in the third stage the Y dimension split within columns, by implementing of another transpose on the Z dimension.

![2D decomposition of the P3DFFT.](image)

All of the above computations are very demanding, and because of the time complexity of the Cooley-Tukey algorithm caused by the butterfly pattern of memory access.

With the prospective of increasing efficiency and optimizing communications, the latest version of the library supports installation by using the MPI_alltoall instead of MPI_alltoallv. For an extended analysis of how kernel works and for timings on different MPI collective operations which are used by the kernel, the reader is referred to [38].
P3DFFT Scaling

The 2D decomposition allows the library to scale up to the limit factor of \( P \leq N^2 \). According to the author [39] the library seems to scale well on petascale platforms, especially in direct numerical simulations of turbulent flows. Moreover, this algorithm could scale even to \( 10^5\) – \( 10^6 \) CPUs when a powerful interconnection network exists.

2.5.4 2DECOMP&FFT

2DECOMP&FFT library is a software library written in Fortran. The latest version of this library on the time of the project is the 1.1.319 (contains an implementation for the FFTW 3.3 library) and it supports 1D and 2D decomposition. This functionality makes the library very useful for large scale simulations on the modern petascale supercomputers.

In particular, the 2DECOMP&FFT library:

- Follows the NAG Fortran library [40] standard, for easy parallelisation of the existing applications.
- Delivers excellent performance and scalability across distributed memory parallel computers, including SMP platforms.

2DECOMP&FFT Design

The 2DECOMP&FFT for the project purposes is built on the top of the optimised FFTW 3.3 and it implements:

- A 2D domain decomposition algorithm (also supports the 1D decomposition).
- 3D FFTs in parallel.
- Complex-to-complex, real-to-complex and complex-to-real data transforms support.

The main design goal of this library is to become easy to use by the scientists of different areas. In order to achieve this goal, the library implementation, does not reveal communication details to the end user. communication details.

2DECOMP&FFT 2D Decomposition

The 2DECOMP&FFT lets the scientist configure manually the \( P_{\text{row}} \times P_{\text{col}} \) 2D processors grid, so scientist can choose between the 1D and the 2D decomposition.
2DECOMP&FFT Scaling

2DECOMP&FFT could be used efficient on large-scale computations in modern supercomputers consist of thousands of cores. Compared to the FFTW which has the limit factor of $P_{\text{processors}} \leq N$, the 2DECOMP&FFT is using the 2 dimensional decomposition in order to achieve scaling up to a large number of processors with the limit factor of $P_{\text{processors}} \leq N_{\text{dimensionX}} \times N_{\text{dimensionY}}$.

2.5.5 Features Comparison

Table 2.2 summarises the most important libraries features the users interested. It is very useful to list these features in order to make a fair comparison of them and refer to the interfaces characteristics. In addition it is important to mention that the FFTW library is a complete & general API whereas the other two libraries require to built on the top of 1D FFT library. Furthermore, the FFTW is auto-tuned in different hardware architectures.

<table>
<thead>
<tr>
<th>FFTW</th>
<th>P3DFFT</th>
<th>2DECOMP&amp;FFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version 3.3</td>
<td>Version 2.4</td>
<td>Version 1.1.319</td>
</tr>
<tr>
<td>1D Decomposition</td>
<td>1D, 2D Decomposition</td>
<td>1D, 2D Decomposition</td>
</tr>
<tr>
<td>c2c, c2r, r2c</td>
<td>c2r, r2c</td>
<td>c2c, c2r, r2c</td>
</tr>
<tr>
<td>Single &amp; Double Precision</td>
<td>Single &amp; Double Precision</td>
<td>Single &amp; Double Precision</td>
</tr>
<tr>
<td>MPI, Threads</td>
<td>MPI</td>
<td>MPI</td>
</tr>
<tr>
<td>Open Source</td>
<td>Open Source</td>
<td>Open Source</td>
</tr>
<tr>
<td>$P_{\text{processors}} \leq N$</td>
<td>$P_{\text{processors}} \leq N^2$</td>
<td>$P_{\text{processors}} \leq N^2$</td>
</tr>
</tbody>
</table>

Table 2.2: FFTW, P3DFFT and 2DECOMP&FFT features.

Finally, it should be noted that the P3DFFT does not support complex-to-complex data transforms. In addition, both P3DFFT and 2DECOMP&FFT libraries do not provide function calls by using threads. Whereas, the FFTW 3.3 provides function calls by using OpenMP.
Chapter 3

Design and Implementation

This chapter is divided into two sections. The first section refers to the project’s design analysis carries out for the implementation of the FFT libraries benchmark.

The second section refers to the project implementation phases including the software and the hardware aspects of the project and the source code implementation.

3.1 Design

In order to benchmark the FFT libraries, it was necessary to investigate the existing FFT benchmarks and how they work. In addition, it was necessary to investigate for any related works which referring to FFT libraries comparison.

The FFT benchmarks methodology is very common and many papers and dissertation thesis are dealing with it. This happens because the parallel programming models are being developed and the hardware architecture upgrades as many new FFT libraries appear. Therefore, a wide variety of options exists on how to implement this kind of project and obtain correct results.

The procedure we chose in order to analyse the FFT benchmark, was the methodology which presented in the FFTW.org website at the benchFFT [41] section. According to this analysis the phases of an FFT benchmark are concerning the data format, the compilation, the time measure and the reporting.

Figure 3.1 illustrates a general FFT benchmarking methodology. In particular, the first phase includes the selection of the input data used during the procedure. These data
can be either complex or real numbers depending on the type of transforms supported by each library.

The second phase is describing the compilation process and is referring on the compilation of the program’s source code by using hand-picked compiler optimisation flags.

Regarding the libraries installation and configuration which are described in section 3.2.3, there isn’t any requirement to test which optimisation flags to use in order to build the libraries. Each of them, either implement routines in order to find their own optimisation flags (FFTW) or their documentation had suggestions about the compiler optimisation flags for certain hardware architectures (P3DFFT and 2DECOMP&FFT).

The third phase is referring to the timing procedure. This is the most critical step during the benchmarking because it is very important to measure the same routines for all the libraries.

The last phase is the reporting, where all the results are reported using graphs and tables. The time is measured in seconds and the bold numbers in the tables represent the minimum time values.

![Diagram](image)

Figure 3.1: FFT benchmarking methodology

### 3.2 Implementation

The implementation section follows the figure’s procedure. In the beginning, there is a description about the programming languages, which were used in order to implement the test programs. Then, there is a description about the test data which that were used by the test programs.
In addition, there is a source code structure analysis, a time measure methodology analysis and a discussion about the scripts were used to facilitate our work.

### 3.2.1 Programming Languages

The test programs were implemented in C and Fortran programming languages. As far as the FFTW 3.3 library is concerned the library is implemented in C and supports function calls both in C and Fortran. However, until the release of the beta version, only the C language interface provided by the FFTW.org. Therefore, the test program is written in C programming language. The performance of both C and Fortran are almost identical for this kind of problems, because the fast Fourier transform computation performed by the FFTW routines which is already implemented in C programming language.

The test programs using the P3DFFT and 2DECOMP&FFT library implemented in Fortran 90 language. The main reason was that both libraries are implemented in Fortran. Moreover, the P3DFFT does not require any particular flag during the installation concerning the arrays manipulation when the program is written in Fortran 90.

Furthermore, the 2DECOMP&FFT library does not provide a C interface. In contrary, for the P3DFFT both C and Fortran interfaces are available.

### 3.2.2 Compilation

The Portland Group compilers [42] were used for the compilation of the source code. The compiler is described in section 4.1.3 of this report.

The PGI latest compiler module was loaded on Hector in order to compile the source codes and build the libraries. We installed the libraries because the FFTW 3.3 was not installed in Hector (the library was built in our home directory). The same procedure followed for the other two libraries which was built in the top of FFTW 3.3.

### 3.2.3 FFT Libraries Installation

For the FFT libraries installation the compiler and the compilers optimisation flags used in order to built the libraries were the optimisations which supported and proposed from each library installation guide for the Cray [43] XE6 systems [44].
We present the installation procedure so the reader can have a clear view of the installation procedure and the options we chose.

Building the Libraries

**FFTW 3.3 Installation**

```
$ ./configure CC=cc CFLAGS=-fast F77=ftn --prefix=/home/d04/d04/s1013813/work/build/fftw3.3.build --enable-openmp --enable-mpi
$ make
$ make install
```

The Cray’s compiler wrapper scripts *cc* and *ftn* are used in order to facilitate the compilation. In addition, the module xtpe-mc12 loaded in order to achieve better performance. This suggestion is presented in the Hector’s website [8].

**P3DFFT 2.4 Installation**

```
$ ./configure --prefix=/home/d04/d04/s1013813/work/build/p3dfft-2.4/ --enable-pgi --enable-fftw --with-fftw=/home/d04/d04/s1013813/work/build/fftw3.3.build --enable-measure --enable-FC=ftn FCFLAGS=-fast CC=cc CFLAGS=-fast
$ make
$ make install
```

As for the 2DECOMP&FFT library installation, the *Makefile.inc.Cray_XE* was used. The *Makefile.inc.Cray_XE* included all the information and the optimisations which were needed to build the library on Hector phase 2b. In addition to the default options, the `DOUBLE PRECISION` and the `fftw3.3` option were selected in order to build the FFT libraries using the same capabilities.

In order to make a fair comparison of the FFT libraries which were used in this project, the following conditions were checked:

- For each library all the test programs compute the same kind of transform.
- All the test programs use the same input binary data file.
- All the libraries built by using double precision. In particular, the P3DFFT and the FFTW libraries build by default by using the double precision and the 2decomp requires the `–DDOUBLE_PREC` flag.
• Both P3DFFT and 2DECOMP FFT libraries linked to the FFTW 3.3 library.
• The argument FFTW_MEASURE used for planning on the FFTW. The P3DFFT requires the --enable-measure and the 2DECOMP&FFT uses by default MEASURE for planning.

Finally, the test programs compiled by using hand-picked optimisation flags and compared to the libraries optimisation flags used in their examples.

3.2.4 Test Data

All the test programs required an input file which keeps the 3D input array. A separate program was implemented in order to generate a binary file containing the input data.

During the experiments, the same binary files (with dimensions 128 × 128 × 128 and 256 × 256 × 256) were used as an input by all the test programs. In particular this program uses the rand() function and initialises the buffer with real numbers from 0 to 1.

Another program which read the data files (input or output) was implemented in order to examine the data of these files at anytime. The main purpose of this program was to observe the data during the debugging process and during the implementation of the project.

Finally, the input and output data files were checked for errors by using the compare program. This program reads both input and output files and performs an one by one comparison of each array’s data. The input and the output data stored separately in two buffers, and each element of the input data buffer is compared to the output data buffer. The errors and the total number of errors are printed on the standard output.

3.2.5 Test Programs

Source Code Structure

A test program written in C programming language by using the Message Passing Interface measured the performance of the FFTW 3.3 library. Two more similar programs implemented for the P3DFFT and 2DECOMP&FFT libraries, but in this case the source code of each program was written in Fortran 90 programming language. Both source codes followed the ordinary structure of a common 3D FFT library test programs.
In this project the input data were not the data of a scientific problem but random data. This is not a disadvantage because most of the real world problems described by complex or real numbers. For this reason most of the benchmarking tests use random data as an input.

All the libraries provide functions for planning and executing the forward and the backward plans. All these functions are built on the top of common MPI functions (except the OpenMP functions of the FFTW). On account of this, the computation part of these programs makes use each library’s functions.

As for the remaining tasks required by the programs, the MPI I/O and the other common MPI functions were used in order to handle the I/O binary data files and some other tasks (for example the processors number, the time measure etc.). All these computations were not included in the time measure procedure.

Timing

The most important part of a benchmarking procedure is the correct measure of the time required by each library for the computations. The FFT timing process we followed during this project reflects the performance of the FFT libraries as they use small 3D FFTs of the same 3D array size i.e. $128 \times 128 \times 128$ or $256 \times 256 \times 256$ on different number of nodes by using either one or more cores per node.

The FFT performance measured by the repeated computing of small 3D FFTs of sizes less equal to $256 \times 256 \times 256$. The same input file created by the create.c program, was used by all the test programs.

The time measuring part in each test program’s source code is measuring the execution time for the forward and backward plans. The forward and the backward plans computations is included inside a FOR LOOP and the number of iterations could be setted by the user. The time reported at the end of each test program is the average time of all iterations.

As for the FFTW timing test, the planning time was excluded (both P3DFFT and 2DECOMP plan once at the initialisation stage). The time in each program was measured by using the MPI_Wtime() function. The same methodology used in the OpenMP FFTW test program where we used the OpenMP omp_get_wtime function.
The code structure followed for the time measuring of the computations in all the test programs was:

**Computation Time Measure**

```plaintext
T=0
DO FOR n ITERATIONS
  T -= MPI_Wtime()
  CALL forward_transform_r2c (B, A)
  T += MPI_Wtime()
  ...
  T -= MPI_Wtime()
  CALL backward_transform_c2r (A, B)
  T += MPI_Wtime()
END DO
```

**Estimate versus Measure**

The planning functions could make either use of the `measure` or the `estimate` argument. As for the FFTW library, the `FFTW_MEASURE` argument requires the library to investigate the most efficient way to compute the fast Fourier transform for a specific hardware. This allows the test program to find the best way to compute the Fourier transform of size N.

In contrast to the `FFTW_MEASURE`, the `FFTW_ESTIMATE` argument does not run any particular calculation, and uses directly a plan which is not as efficient as the optimal plan. Therefore, the `FFTW_MEASURE` argument was used in order to have an optimal plan for the computations of the transforms.

### 3.2.6 Scripts

**Makefiles**

A makefile for each library was used in order to built each test program. These makefiles contained the path of each library, the compiler’s optimisation flags, the FFTW environmental flags and the compiler.

**Jobs Submission**

The data collection was the most time consuming procedure. Numerous tests had to execute on Hector’s nodes. Each test program returns on the standard output the
processors grid, the decomposition and the average time of the iterations, which could be initialised from the user.

In addition, the task scheduler used in Hector, prints in the standard output many other useful details such the total number of processors and the number of cores per node. All these details were very useful during the classification of the obtained results.

A bash script was implemented for the FFTW OpenMP and MPI test programs. This script automated the procedure of the tasks submissions. In addition, when the script finishes and all the output files are in the folder, another bash script is parsing the files and keeps the required data.

The other two libraries which supported the 2D decomposition, was required the test program to be compiled for every processors grid (except when the automated routine mode was selected in the 2DECOMP&FFT).
Chapter 4

Experimental Design

4.1 HPC Systems Architecture

The last two decades the HPC Systems are becoming a trend in the scientific world. Most of the universities and the research centres make use of machines which consist of a large number of cluster nodes. The vast majority of those supercomputer’s nodes are SMP machines consist of one or more multicore processors. The architecture complexity of these systems, complicates the programming and makes difficult the utilization of all the available computing resources.

Such machines can run parallel programs which require demanding computations and numerous tasks. The supported programs usually, are using the Message Passing Interface or other models of parallelisation, in order to be executed in these parallel environments. In recent years there is great interest to search for the ideal programming model of these supercomputers.

In the beginning of this section we present some specific information about the project’s hardware and software and then we analyse how we setup the experiment and which arguments we use for different total cores and cubes combinations for each decomposition.

4.1.1 Hardware

The performance of the supercomputers is measured and listed in the TOP500.org [45]. This test programs were executed on Hector Phase 2b which is one of the fastest supercomputers of the TOP500 list, and the fastest supercomputing facility in the United Kingdom. The architecture features of Hector phase 2b summarised on table 4.1.
Distributed Memory Systems with SMP nodes

The structure of the modern supercomputer is very complex. In particular, each SMP system node includes everything from a two one-core processor or multiple processors consisting of multiple cores by using separate or the same main memory.

Machines like Cray XE6 consist of clusters of SMP nodes. In Hector each node consists of two twelve-core processors. In particular, each one of the AMD’s Magny-Cours Opteron processors [46] has two dies each one is containing 6 cores. Inside the processor 12 MB of level 3 cache shared by both dies. Each one of these dies has 6MB level 3 cache and this capacity is shared by the six cores.

Each die in the processor has two memory channels and each one of these dies is a Non-Uniform Memory Access (NUMA) node. As a result each one of the nodes consists of four NUMA regions.

Programming Clusters of Multicore-Nodes

The mixed environment of distributed SMP systems sometimes do not allow the programs to scale above a large number of cores, therefore, the programming model which best suits on these hardware architectures is a very open-ended question.

Programming models such the Message Passing Interface are very common for the programs in order to run on parallel machines like supercomputers. In particular MPI allowed the programs to scale up to thousands of processors and most of the programs in the scientific world are implemented by using MPI.

Recent work [47] and [48] have shown that the mixed mode programming model by using the Message Passing Programming model for the inter-node programming and

<table>
<thead>
<tr>
<th>Machine</th>
<th>Cray XE6 system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Nodes</td>
<td>464 blades×4 nodes</td>
</tr>
<tr>
<td>Cores per Node</td>
<td>24 (2 × 12core)</td>
</tr>
<tr>
<td>Processor</td>
<td>AMD Opteron 64bit</td>
</tr>
<tr>
<td>Model</td>
<td>Magny Cours</td>
</tr>
<tr>
<td>Cores</td>
<td>12</td>
</tr>
<tr>
<td>Frequency</td>
<td>2.1GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>32Gb (2 × 16Gb)</td>
</tr>
<tr>
<td>Total Memory</td>
<td>59.4Tb</td>
</tr>
</tbody>
</table>

Table 4.1: Hector Hardware Features
a Shared Memory Programming model for the intra-node programming are offering interesting results.

4.1.2 Interconnection Network

As a CRAY XE6 machine Hector uses the Gemini Scalable interconnect [49]. The Gemini ASIC is designed to implement $10^7$ of MPI messages per second. The main advantage of the Gemini Interconnect is that offers direct hardware support to the PGAS languages (for example the UPC [50] and the Co-Array Fortran [51]). As expected, the programs which are written in one of these languages have better performance in systems with the Gemini Interconnect.

On a CRAY XE6 machine (for example Hector) each two nodes share a Gemini chip which has 10 links in order to implement a three dimensional torus of CPUs. On Hector the Gemini features summarised in the table 4.2.

<table>
<thead>
<tr>
<th>MPI point-to-point bandwidth</th>
<th>5GB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency between 2 nodes</td>
<td>1 − 1.5µs</td>
</tr>
</tbody>
</table>

Table 4.2: Gemini Interconnect on Hector Phase 2b

4.1.3 Compiler

The Portland Group compilers were used in order to compile the source codes. The PGI compiler were available in Hector between many other compilers & tools. In addition the Hector’s module environment used in order to manage the systems’s software components.

The PGI compiler includes all the latest compiler optimisations which can be used by calling the appropriate flag in the compilation. The main flags used in order to compile the source codes were the -O3 and the -Mpreprocess. Those flags were also used by the library’s examples.

Many other optimisation flags examined such the -fastsse, -Msmartalloc, -O2 and -Mipa=fast,inline without any performance improvement. Probably, that was the reason the 2DECOMP&FFT makefile for the Cray XE6 systems didn’t make use these flags.

In addition many information about the compilation taken from the PGI, the Hector’s and the FFT libraries websites.
4.2 Experiments Setup

4.2.1 Testing Different Decompositions

In this report the 1D decomposition sometimes called a "slab decomposition". In this approach the first dimension of the input array is divided to the rows and each task assigned a subset of the data. As discussed in section 2.4 the maximum number of the tasks is equal to the first dimension.

The 2D decomposition sometimes called a "rod decomposition" or "pencil decomposition" because the data are divided over the two dimensions and the maximum number of processors depends on the 3D array dimensions.

The three programs which were used for the project (one for each library) required the input binary data file, the parameters $P_{\text{row}}$, $P_{\text{col}}$ (processors grid), $\text{dimX}$, $\text{dimY}$, $\text{dimZ}$ (dimensions of the 3D array) and the name of the output file.

For the 1D decomposition each test program executed by using up to 16 cores per node on Hector for the test cases presented in table 4.3:

<table>
<thead>
<tr>
<th>Dimensions</th>
<th>$P_{\text{row}} \times P_{\text{col}}$ (Processors Grid)</th>
<th>$\text{dimX}$, $\text{dimY}$, $\text{dimZ}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$128 \times 128 \times 128$</td>
<td>${1} \times {1, 2, 4, 8, 16, 32, 64, 128}$</td>
<td>128, 128, 128</td>
</tr>
<tr>
<td>$256 \times 256 \times 256$</td>
<td>${1} \times {1, 2, 4, 8, 16, 32, 64, 128, 256}$</td>
<td>256, 256, 256</td>
</tr>
</tbody>
</table>

Table 4.3: 1D decomposition test parameters

For the 2D decomposition we implemented two phases of tests. The first phase was an experimental phase. In particular, we changed the processors grid of each library by using all the possible combinations, in order to find how much the processors grid affects the computational time required by the libraries. During this tests we used fixed array dimensions and fixed number of cores (i.e. for 512 cores and 3D array with dimensions $256 \times 256 \times 256$).

The results of these tests were analysed and used for the main testing procedure of the 2D decomposition, in order to used the optimal processors grid for each library and collect the best possible timings.

In the second experiment where the 2D decomposition was used, each test program executed by using 1 and 16 cores per node on Hector for the test cases presented in the table 4.4:
During all the experiments, the number of iterations was fixed to 20. Moreover, each test program executed on the back-end nodes 20 times. The jobs submitted up to 16 cores per Hector’s node (for the 1, 2, 4, 8, 16 total cores only one node used, except the experiments with one core per node).

The timing results we presented in this report are the minimum average timings of the 20 iterations which achieved from the 20 times of each test program execution.

The majority of the experiments implemented by using up to 16 cores per node in Hector. Each Hector’s node, as presented in table 4.1, consists of 24 cores. However, this number it is not convenient because it is not a power of 2. Therefore, it was difficult to distribute equally the tasks to the nodes. In addition, this is one of the reasons the mixed mode programming models such the OpenMP/MPI developed, in order to optimise the programs performance in such hardware architectures.

As a result, the maximum number of cores per node which belongs to the power of 2 was the 16, and all the tests implemented by using 1, 2, 4, 8 and 16 cores per node. In addition, a number of experiments implemented in 24 cores per node in order to examine the intra-node communications.

### 4.2.2 Reporting

All the test programs produce an output data file. This file was compared to the input file by using the program which implements the comparison, in order to find if there are any errors between the files. In addition, on the standard output we got the average time required for the computation.

### Plots and Tables

Time – Cores and Speedup – Cores graphs were used in order to compare the performance of the FFT libraries. The graphs present the average time in order to be readable. In addition, tables present each experiment details and the speedup of the programs were used in order the reader to have all the information required for the understanding of the analysis.

<table>
<thead>
<tr>
<th>Dimensions</th>
<th>$P_{row} \times P_{col}$ (Processors Grid)</th>
<th>dimX, dimY, dimZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>$128 \times 128 \times 128$</td>
<td><em>Optimal Processors Grid</em></td>
<td>128, 128, 128</td>
</tr>
<tr>
<td>$256 \times 256 \times 256$</td>
<td><em>Optimal Processors Grid</em></td>
<td>256, 256, 256</td>
</tr>
</tbody>
</table>

Table 4.4: 1D decomposition test parameters
The bold numbers in tables present the best time achieved across the libraries for the same number of total cores, cores per node and cubes dimensions.

**Speedup**

The speedups presented in the tables in this report calculated by using the common speedup formula: $S_p = \frac{T_1}{T_p}$, where the $p$ is the total number of processors, $T_1$ is the execution time required by the sequential program and $T_p$ is the execution time required by the parallel program using $p$ number of processors.

**Mflops**

The megaflops performance during the FFT is calculated using the following formula:

$$M F l o p s \approx 2.5 \times \frac{N \log_2(N)}{time_{10^{-3}sec}}, \quad N \text{ is the product of the FFT dimensions}$$

MFlops is a scaled version of the speed and it is defined by the above formula, which referred to the real transform.

The radix-2 Cooley-Tukey FFT algorithm asymptotically requires $5N \log_2(N)$ floating point operations and the FFT MFlop estimate is used in most of the FFT benchmark reports in order to compare the algorithm efficiency.
Chapter 5

Results and Performance Analysis

This chapter presents the results collected during the tests. Figures and tables were used in order to describe the behaviour of each library. All the timings presented in this chapter are referring to the performance of each library for the computation of the Fourier transform of two 3D meshes with dimensions $256 \times 256 \times 256$ and $128 \times 128 \times 128$.

5.1 Slabs and Pencils on Hector

In the first two parts of this section we presented the results of the 1D decomposition along with an analysis of the results. The other four parts are referring to the performance of the P3DFFT and 2DECOMP&FFT libraries by using the 2D decomposition (Rods or Pencils).

Finally, in the last section we present a comparison between the slab and the pencil decomposition after interpreting the obtained results from all the experiments in Hector.

5.1.1 1D Decomposition (Slab)

$256 \times 256 \times 256$ – 1 core per node

Table 5.1 shows the test programs timings by using only one core per node. From the observed timings when one core per node was used, the libraries performance as presented in the figure A.1 is almost identical. The timing needed from the FFTW on the sequential code was the minimum between the libraries.
The experiments using from 2 to 128 nodes (cores) presented in 5.1, the P3DFFT library performed better than the other two libraries. Furthermore, in the case of 256 nodes (cores) the FFTW 3.3 library outperform the other two libraries. The FFTW 3.3, P3DFFT and 2DECOMP&FFT speedups were 95.68, 99.28 and 97.49 respectively. Figure 5.1 illustrates the speedups concerning these tests.

<table>
<thead>
<tr>
<th>Cores</th>
<th>FFTW SPEEDUP</th>
<th>P3DFFT SPEEDUP</th>
<th>2DECOMP&amp;FFT SPEEDUP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.1928</td>
<td>1.2954</td>
<td>1.3716</td>
</tr>
<tr>
<td>2</td>
<td>0.8936</td>
<td>0.7061</td>
<td>0.7220</td>
</tr>
<tr>
<td>4</td>
<td>0.4054</td>
<td>0.3658</td>
<td>0.3709</td>
</tr>
<tr>
<td>8</td>
<td>0.2157</td>
<td>0.1864</td>
<td>0.1897</td>
</tr>
<tr>
<td>16</td>
<td>0.1132</td>
<td>0.0913</td>
<td>0.0979</td>
</tr>
<tr>
<td>32</td>
<td>0.0535</td>
<td>0.0446</td>
<td>0.0470</td>
</tr>
<tr>
<td>64</td>
<td>0.0292</td>
<td>0.0245</td>
<td>0.0263</td>
</tr>
<tr>
<td>128</td>
<td>0.0185</td>
<td>0.0157</td>
<td>0.0167</td>
</tr>
<tr>
<td>256</td>
<td>0.0125</td>
<td>0.0130</td>
<td>0.0141</td>
</tr>
</tbody>
</table>

Table 5.1: SLAB: 256³, 1 core per node, time and speedup

Figure 5.1: SLAB: Speedup using 256³ 3D FFT with 1 core per node

256 × 256 × 256 – 16 cores per node

Figure 5.2 presents the speedup of the libraries when up to 16 cores per node were used. In particular, from 1 to 16 cores, one node was used for the experiments from 16 to 256 cores, 16 cores per node were used (for example the 256 total cores distributed in
16 nodes). The FFTW library achieved the minimum sequential time and the P3DFFT achieved the minimum timings up to 256 total cores. In addition, the performance of the 2DECOMP&FFT library was almost identical to the performance of the P3DFFT. The FFTW, P3DFFT and 2DECOMP&FFT speedups as presented in Table 5.2 were 34.95, 55.50 and 56.02 respectively.

<table>
<thead>
<tr>
<th>Cores</th>
<th>FFTW</th>
<th>SPEEDUP</th>
<th>P3DFFT</th>
<th>SPEEDUP</th>
<th>2DECOMP&amp;FFT</th>
<th>SPEEDUP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.1928</td>
<td>1.00</td>
<td>1.2954</td>
<td>1.00</td>
<td>1.3716</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>1.0435</td>
<td>1.14</td>
<td>0.7670</td>
<td>1.69</td>
<td>0.7330</td>
<td>1.87</td>
</tr>
<tr>
<td>4</td>
<td>0.5595</td>
<td>2.13</td>
<td>0.4615</td>
<td>2.81</td>
<td>0.4649</td>
<td>2.95</td>
</tr>
<tr>
<td>8</td>
<td>0.3809</td>
<td>3.13</td>
<td>0.2771</td>
<td>4.67</td>
<td>0.2798</td>
<td>4.90</td>
</tr>
<tr>
<td>16</td>
<td>0.2089</td>
<td>5.71</td>
<td>0.1443</td>
<td>8.98</td>
<td>0.1481</td>
<td>9.26</td>
</tr>
<tr>
<td>32</td>
<td>0.1296</td>
<td>9.20</td>
<td><strong>0.0852</strong></td>
<td>15.21</td>
<td>0.0875</td>
<td>15.67</td>
</tr>
<tr>
<td>64</td>
<td>0.0847</td>
<td>14.09</td>
<td><strong>0.0479</strong></td>
<td>27.02</td>
<td>0.0489</td>
<td>28.03</td>
</tr>
<tr>
<td>128</td>
<td>0.0440</td>
<td>27.08</td>
<td><strong>0.0383</strong></td>
<td>33.82</td>
<td>0.0422</td>
<td>32.53</td>
</tr>
<tr>
<td>256</td>
<td>0.0341</td>
<td>34.95</td>
<td><strong>0.0233</strong></td>
<td>55.50</td>
<td>0.0245</td>
<td>56.02</td>
</tr>
</tbody>
</table>

Table 5.2: SLAB: 256³, 16 cores per node, time and speedup

This project focuses on small 3D FFTs. Therefore, it was necessary to experiment with even smaller cubes. The same experiments were performed for the case of a 3D mesh with dimensions 128 × 128 × 128. In the first experiment we used one core per

128 × 128 × 128

Figure 5.2: SLAB: Speedup using 256³ 3D FFT with 16 cores per node
node and then 16 processors per node (from 1 to 16 cores total number of cores only one node was used).

\[128 \times 128 \times 128 \text{ – 1 core per node}\]

Figure 5.3 shows the speedup during the experiment with \(128^3\) cubes using one core per node. When the test programs executed using only one core (sequential program), the FFTW completed the computations by using the minimum time compared to the sequential timings achieved by the other libraries.

From 2 to 64 cores, the performance of the P3DFFT and the 2DECOMP&FFT were almost identical, with the P3DFFT achieving better timings during all the tests. Figure A.3 and table 5.3 present the results in more detail.

Finally, when the maximum number of cores was used (128 because of the slab decomposition limit factor) the FFTW 3.3 test program outperform the other two libraries, and required less time to complete the Fourier transform computation.

<table>
<thead>
<tr>
<th>Cores</th>
<th>FFTW</th>
<th>SPEEDUP</th>
<th>P3DFFT</th>
<th>SPEEDUP</th>
<th>2DECOMP&amp;FFT</th>
<th>SPEEDUP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1324</td>
<td>1</td>
<td>0.1445</td>
<td>1</td>
<td>0.1554</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0.0845</td>
<td>1.56</td>
<td>0.0792</td>
<td>1.8234</td>
<td>0.0806</td>
<td>1.92</td>
</tr>
<tr>
<td>4</td>
<td>0.0477</td>
<td>2.77</td>
<td>0.0387</td>
<td>3.7281</td>
<td>0.0397</td>
<td>3.90</td>
</tr>
<tr>
<td>8</td>
<td>0.0226</td>
<td>5.84</td>
<td>0.0178</td>
<td>8.0818</td>
<td>0.0194</td>
<td>7.98</td>
</tr>
<tr>
<td>16</td>
<td>0.0117</td>
<td>11.31</td>
<td>0.0093</td>
<td>15.5040</td>
<td>0.0096</td>
<td>16.14</td>
</tr>
<tr>
<td>32</td>
<td>0.0070</td>
<td>18.77</td>
<td>0.0050</td>
<td>28.8104</td>
<td>0.0051</td>
<td>29.99</td>
</tr>
<tr>
<td>64</td>
<td>0.0048</td>
<td>27.32</td>
<td>0.0035</td>
<td>40.8118</td>
<td>0.0039</td>
<td>39.72</td>
</tr>
<tr>
<td>128</td>
<td>0.0035</td>
<td>37.77</td>
<td>0.0040</td>
<td>36.0052</td>
<td>0.0044</td>
<td>35.23</td>
</tr>
</tbody>
</table>

Table 5.3: SLAB: \(128^3\), 1 core per node, time and speedup

\[128 \times 128 \times 128 \text{ – 16 cores per node}\]

In the experiment where we computed the Fourier transform of a 3D mesh with dimensions \(128 \times 128 \times 128\) by using 16 cores per node the FFTW achieved the minimum timing by using 1 and 128 total cores. As we can see from the figure 5.4 in all the other tests the P3DFFT library requires the minimum time to complete the computations by using different numbers of cores between 1 and 128, compared to the other two libraries.

Once again, the 2DECOMP&FFT according to the timings presented in the table 5.4 had similar performance to the P3DFFT.
The test programs executed on Hector by using up to 16 cores per node for the computation of the FT of a 3D input array with dimensions \(256 \times 256 \times 256\), in order to analyse the performance of the libraries using MPI. In addition to these tests, another test was implemented by using the openmp FFTW functions.

FFTW test program using the OpenMP was executed on up to 16 threads and the results were compared to the MPI results from the other two libraries where up to 16 MPI tasks was used. In addition, the results compared to the FFTW MPI results. The obtained results were presented in Table 5.5 and show that the FFTW OpenMP version achieved the minimum timings for all the core/threads selection.
Finally, the figure A.4 shows that the OpenMP outperform all the other implementations by using only one node, including those implementations are using the 2D decomposition.

<table>
<thead>
<tr>
<th>Threads/Cores</th>
<th>OpenMP</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FFTW</td>
<td>SPEEDUP</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1.1617</td>
</tr>
<tr>
<td>2</td>
<td>0.6175</td>
<td>1.88</td>
</tr>
<tr>
<td>4</td>
<td>0.3303</td>
<td>3.52</td>
</tr>
<tr>
<td>8</td>
<td>0.1810</td>
<td>6.42</td>
</tr>
<tr>
<td>16</td>
<td>0.1056</td>
<td>10.99</td>
</tr>
</tbody>
</table>

Table 5.5: SLAB: 256³, FFTW OpenMP version versus MPI in one node

5.1.2 1D Decomposition Performance Analysis

In general from the 1D decomposition results we can make a conclusion that:

**SLAB: 256 × 256 × 256 – 1 core per node**  All the libraries had very good scaling.

**SLAB: 256 × 256 × 256 – 16 cores per node**  P3DFFT and 2DECOMP&FFT test programs had better speedup than the FFTW.

**SLAB: 128 × 128 × 128 – 1 core per node**  FFTW test program scales better than the other two libraries test programs.
SLAB: 128 × 128 × 128 – 16 core per node FFTW test program scales better than the other two libraries test programs.

256 × 256 × 256

According to the results presented in figures A.1 and A.2 all the libraries achieved to minimize the computational time of the Fourier transform of a 3D mesh with dimensions 256 × 256 × 256 (and 128 × 128 × 128).

In particular, the 3.3 version of the FFTW achieved the best performance by using 1 and 256 total cores (128 for the 128 × 128 × 128 array). However, in the experiment for the computation of the FFT on a 3D array with dimensions 256 × 256 × 256 presented by figure A.2 By using 16 cores per node the FFTW achieved worse timings compared to the other two libraries.

In general, as presented in the figures 5.1, 5.2, 5.3 and 5.4, the tests programs which were used the P3DFFT library achieved very good timings.

In particular, the library achieved the minimum timings, except the timings achieved during the tests by using one core and the maximum number of cores (depending on the experiment and the limit factor of the 1D decomposition). In addition, the P3DFFT achieved the minimum timings by using up to the maximum number of total cores, in
order to complete the Fourier transform computation of a cube with dimensions $256 \times 256 \times 256$.

Furthermore, during the test using an input with dimensions $256 \times 256 \times 256$, as the number of cores increased the time required for the computations decreased. This behaviour shows that the library scales well. The tables 5.1 and 5.2 present in detail the speedups and the timings in each case.

The test programs implemented by using the 2DECOMP&FFT library recorded almost identical timings to the P3DFFT. Moreover, in individual experiments the 2DECOMP&FFT tests required less time to complete the computations.

$128 \times 128 \times 128$

It should also be noted that when the dimensions of the 3D input array reduced from $256 \times 256 \times 256$ to $128 \times 128 \times 128$ the P3DFFT and the 2DECOMP&FFT, as presented in the tables 5.3 and 5.4 both libraries achieved their minimum timings for 64 cores. Whereas, by using 128 total cores the performance decreased regardless the number of cores per node.

On the contrary, during all the tests the FFTW test programs, minimise the computational time as the total number of cores increased up to the maximum number of cores allowed by the 1D decomposition.

The analysis concerning the 3D array with dimensions $128 \times 128 \times 128$ and the tables 5.3 and 5.4 show that the P3DFFT and 2DECOMP&FFT library test programs which compute small 3D FFTs such $128^3$ did not scale well when the number of cores exceeds a number of cores depending on the 1D decomposition limit factor for the total tasks. In particular, there is a performance decrease after a number of total cores, regardless the number of cores per node were used.

**Intra-Node 1D Decomposition Performance Analysis**

The FFTW OpenMP version program achieved better timings compared to the MPI test versions of the other libraries including the FFTW’s test program written in MPI. The speedup of the FFTW’s test program written in OpenMP, as presented in the table 5.5 significant compared to the speedups and timings of the FFTW MPI version as presented in table 5.2. The memory usage and the communications inside the node of the OpenMP function calls, are better than the MPI and let the algorithm to scale faster.
In particular, the FFTW MPI test program needs 16 cores to achieve the FFTW OpenMP test program’s performance using 8 threads (cores). A comparison between MPI and MPI/OpenMP for CRAY XE6 systems (reader can be referred to the [52]), concludes that the FFT computation programs using MPI/OpenMP scale better than the pure MPI implementations. In addition, the OpenMP model reduces the memory usage inside a node. To sum up, the MPI presented difficulty to scale further a number of cores, especially for thousands of cores presented in the tables A.4 and A.1. In particular, the performance depends on the 3D input array dimensions and the total number of cores.

1D Decomposition - Conclusion

Summarizing the above analysis, the FFTW requires the minimum time to compute the Fourier transform of small cubes by using the maximum number of cores. The other two libraries have very close behaviour and almost identical performance. From the figures A.1, A.2, A.1 and A.2 it is difficult to distinguish one of them, but in the majority of the tests the P3DFFT achieved slightly better timings.

Another interesting point is that the timings achieved by the libraries by using one core per node were better than the timings when up to 16 cores per node was used. This behaviour is observed because of the memory access between the intra-node cores and the demanding communications. In addition, the Gemini Interconnect minimises the communication overhead between the cluster nodes.

Finally, it is important to mention the timings achieved by the FFTW test program by using the FFTW’s OpenMP functions. These timings were the minimum timings achieved during the 1D decomposition experiments 5.5 by using up to 16 cores per node. The FFTW OpenMP functions are more efficient when applied on SMP hardware architectures. This behaviour is due to the lack of replicated data and the intra-node MPI overheads.

5.1.3 2D Decomposition (Rods or Pencils)

These sections present the performance of the P3DFFT and the 2DECOMP&FFT libraries using 2D decomposition. The FFTW does not support 2D decomposition as a result it was excluded from the experiment. However, for the benchmarking procedure, both libraries are built on the top of the FFTW 3.3.

This section is divided into two parts. In the beginning there is a description about the processors grid selection and there is also a benchmark concerning the performance of the libraries by using different processors grid (those applied on a fixed number of
cores by using a 3D array with dimensions $256 \times 256 \times 256$. Furthermore, we analysed the behaviour and the performance of each library.

In the second part we present the main benchmarking procedure were libraries using the 2D decomposition. The obtained results presented the performance of each one of the two libraries by using different numbers of total cores.

At the end of the section there is an extended analysis based on the overall libraries performance using the pencil decomposition.

### 5.1.4 Processors Grid Selection

The processors grid selection has an important role in the computation of the Fourier transform. As mentioned in the experimental analysis on section 4, many tests implemented in order to find the most efficient processors grid of each library by using a total number of physical cores.

The 2DECOMP&FFT library supports an automated routine, where the library undertakes the task to find an ideal processors grid using a number of total cores.

The 2DECOMP&FFT automated routine seems to work very well for different sizes of 3D FFTs. Furthermore, the choice it is very efficient when thousands of cores were used for the computations. In both libraries, the optimal 2D grid layout depends on the system’s software and hardware architecture and on each library’s architecture.

The correct choice of 2D processors grid can improve the communication efficiency and the decomposition balance. As described in the P3DFFT library guide [53], for a given grid of dimensions $K \times L \times M$, the decomposition sometimes it is not entirely even, as a result the elements are not divided equally to the tasks. In such cases, the load imbalance of the problem maximises and the program’s performance decreases. In this project the selected numbers were power of two, therefore, we didn’t have a such problem.

Figure [5.6](#) shows the possible options of the physical processors grid by using 1024 cores (16 cores per node) for the computation of a 3D mesh with dimensions $256 \times 256 \times 256$. 

39
From the timings indicated in figures 5.6 and A.9 (the same experiment with 512 total cores) it is observed that for the particular occasion of the $256^3$ dimensions 3D mesh for both libraries the ideal processors grid is about $P_{row} \approx P_{col}$. Especially when the $P_{row} \times P_{col}$ was 32x32 in figure 5.6 the timing achieved by each library was the minimum.

![Figure 5.6: Libraries performance using $256^3$ 3D array, 1024 total cores](image)

**Processors Grid Analysis**

As analysed in the 2decomp.org website [54] in order to achieve better performance a particular test in [54] shows that the $P_{row} \ll P_{col}$ is the better option to select the processors grid. In addition, in the "Processors Grid" page of the [54] noted that the $P_{row}$ value should be smaller than or equal to the number of the cores of each physical processor. This remark concerning the CRAY XT4 machines with quad core processors with SeaStar interconnection network.

In our project the test results presented in figures 5.6 and A.9 show that the Hector’s phase 2b interconnection network (Gemini) which used in CRAY XE6 supercomputers, optimise the inter-node communication performance (compared to the older interconnects) and achieved better performance when the $P_{row}$ is bigger than the physical processors cores. The same remark observed in the report [55].

In addition, the non-uniform memory access (NUMA) arrangement seems to be less efficient for the intra-node communications (because of the complexity of the AMD’s
Magni-Cours architecture \[46\]). However, when thousands of cores were used in order to compute large 3D FFTs the \( P_{\text{row}} \ll P_{\text{col}} \) it is preferred as reported from the timings on \[54\]. The computations seems to be more balanced by using this ratio.

The processors grid selection had similar affect on the performance of the P3DFFT library. According to the presented results in \[56\] the performance increased when the \( P_{\text{row}} \ll P_{\text{col}} \), and as presented in that project’s benchmarks, while the number of the processors increased the ideal processors grid converges to \( P_{\text{row}} \approx P_{\text{col}} \) (for the particular 3D FFTs used in this project). Figures 5.6 and A.9 present a similar response because while the number of the cores increased from 512 to 1024 the best processors grid changed from \( P_{\text{row}} \ll P_{\text{col}} \) (512 cores) to \( P_{\text{row}} \approx P_{\text{col}} \) (1024 cores).

Finally, for each library the optimal processors grid depends mainly on the total number of cores. For both libraries the \( P_{\text{row}} \) should be less or equal to the \( P_{\text{col}} \) depending on the total cores, the system hardware and the dimensions of the array. In the majority of the occasions the optimal processor grid is when the \( P_{\text{row}} \) is less than or equal to the \( P_{\text{col}} \).

### 5.1.5 2D Decomposition Performance

#### 256 × 256 × 256 – 1 core per node

The timings for each experiment presented in table A.1 and figure 5.9 are concerning the test programs which compute the Fourier transform of a 3D mesh with dimensions 256 × 256 × 256. Especially, both libraries achieved their best timings by using 512 total cores. In addition, the libraries achieved the timings presented in table A.2 by using the same processors grid.

During the tests when using from 1 to 8 total cores, the P3DFFT had better performance compared to the 2DECOMP&FFT. When using from 16 to 512 cores the 2DECOMP&FFT required less time to complete the computation. In addition, the 2DECOMP&FFT requires less time to perform the computations compared to the P3DFFT.

#### 256 × 256 × 256 – 16 cores per node

In the experiment where up to 16 cores per node were used, the 2DECOMP&FFT achieved the minimum timing by using 8192 total cores in order to execute the forward and the backward plans (3D mesh with dimensions 256 × 256 × 256). In particular, as the total number of cores increased up to 8192 total cores the performance of the test program was increased, as shown in the table A.1.
The P3DFFT library had an almost identical performance to the 2DECOMP&FFT’s by using up to 4096 total cores. However, in this test case using 8192 cores the average time required by the library for the computations increased compared to the time required when it used 4096 cores. During this experiment, the processors grid used from both libraries was the same.

128 × 128 × 128 – 1 core per Node

The performance of the libraries according to the figure 5.8 and the results presented on table A.3 were similar. In particular, the P3DFFT test program achieved best results when 1 and 512 cores were used. The computational time required by the test program decreased as the number of the cores increased up to 512 cores in Hector.

The performance of the 2DECOMP&FFT library test program was similar to the P3DFFT’s performance. The test program scales well up to 256 cores when it achieved the minimum required time for the computations. However, the performance decreases when using 512 total cores. The result is presented in the table A.3.

The processors grid were used by each one of the test programs corresponding to the minimum average timings achieved by them are presented in table A.4.
128 × 128 × 128 – 16 cores per Node

Figure A.6 presents the performance of the test programs by using the P3DFFT and the 2DECOMP&FFT in order to compute the Fourier computation of a 3D mesh with dimensions 128 × 128 × 128.

In this case, when 16 processors per node were used, the P3DFFT library had the best performance when using 4096 cores and the 64 × 64 processors grid. After that, the performance decreased when using 8192 total cores.

In particular, using the P3DFFT library we observed that, as the number of processors increased the time required for the computations decreased. However, that happened up to the 4096 cores. For the number of 8192 total cores the time required for the computations increased.

The 2DECOMP&FFT library achieved its minimum computational timing as presented in table A.4 by using 2048 total cores (32 × 64 processor grid). Finally, the processors grid used from both libraries confirms the results reported in the [56] and [54] regarding the optimal processors grid for each library.

![Figure 5.8: PENCIL: Speedup using 128³ 3D FFT 1 and 16 cores per Node](image)
5.1.6 2D Decomposition Performance Analysis

The obtained results presented in tables A.3, A.4, A.2 and A.1 show the performance of the libraries for the computation of the Fourier transform of a 3D input array using the 2D decomposition (Pencil). Generally, the results shows that the test programs achieved to decrease the computational time of the Fourier transform as the number of the cores increased by using up to 16 cores per node.

\[256 \times 256 \times 256\]

The figure 5.7 shows that during the experiment with a 3D mesh larger than \(128 \times 128 \times 128\) \((256 \times 256 \times 256)\) the performance of the libraries had better scaling.

In contrast to the \(128 \times 128 \times 128\) test case, the 2DECOMP&FFT requires the minimum time to complete the computations using 8192 cores (16 cores per node). This timing is the minimum recorded for the computation of the 3D array with dimensions \(256 \times 256 \times 256\). As the number of cores increased (by using 1 and 16 cores per node), the time required by the library decreased.

2DECOMP&FFT for both tests achieved better timings by using 1 and 16 cores per node, compared to the P3DFFT timings. However, the communication overhead of the FFT computation affected the performance and the time needed by the 512 cores (1 core per node) is better compared to the time required by the library when 4096 cores were used (16 cores per node).

The P3DFFT library when using 1 core per node had an almost identical performance with 2DECOMP&FFT. However, when using 16 cores per node it appeared to have similar performance with 2DECOMP&FFT when up to 4096 cores were used (it required less time for the computations using 4096 cores compared to the 2DECOMP&FFT). Whereas, the communication overhead affected the performance and all the achieved timings by using 8192 total cores (16cores per node) were close to 0.0082s which performance is similar to the time measured by using 512 total cores (16 cores per node).

\[128 \times 128 \times 128\]

The minimum timing achieved was the time required for the computations by the P3DFFT library by using 512 cores (1 core in node). The library required more time for the computations when it uses 4096 cores (16 cores per node) compared to the experiment with the 512 cores (1 core per node).
The 2DECOMP&FFT required more time for the computations using one core per node, but it achieved a similar timing to the minimum P3DFFT timing, by using 256 cores instead of the 512 cores required by the P3DFFT.

Tables A.4 and A.3 show that the performance of the 2DECOMP&FFT starts to decreases after the 2048 cores by using 16 cores per node (after the 256 cores by using one core per node). The P3DFFT which had better performance in both tests seems to be more efficient because its performance decreased after the 4096 cores when 16 cores per node were used, whereas up to the 512 total cores (1 core per node) there is not such decrease.

5.2 Slabs versus Pencils on Hector

In this section we summarised the analysis about the performance of the libraries by using either 1D or 2D decomposition. We wanted to make a comparison regarding which decomposition and which library required the minimum time for the Fourier transform computation of the 3D mesh with dimensions $256 \times 256 \times 256$ and $128 \times 128 \times 128$.

$256 \times 256 \times 256$

Regarding the Fourier transform computations of a 3D array with dimensions $256 \times 256 \times 256$ the 2DECOMP&FFT achieved the minimum overall timing ($0.0045$ sec) by using 8192 cores and the 2D decomposition. The same library achieved the second minimum overall timing by using 512 core (1 core per node) and the 2D decomposition.

Slightly worse was the performance of the P3DFFT library. The library’s test programs achieved their minimum timings by using 4096 total cores (16 cores per node) and 512 cores (1 core per node). Both timings achieved by using 2D decomposition and are presented in table 5.9.

In general, the 2D decomposition implementations achieved better timings compared to the 1D decomposition. For the same number of nodes the FFTW timing was $0.0124$s (best timing in 1D decomposition), whereas the P3DFFT timing was $0.0060$s. The FFTW by using 1D decomposition requires $51.62\%$ more time to complete the computation. In addition, the performance of the FFTW when using 256 nodes it is not the most cost efficient because it makes use of 1 core per node instead of 16 cores per node.
Table 5.6: 256\textsuperscript{3}: 1D & 2D decomposition results

<table>
<thead>
<tr>
<th>Library</th>
<th>Time</th>
<th>Cores</th>
<th>Cores/Node</th>
<th>Decomposition</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2DECOMP&amp;FFT</td>
<td>0.0045</td>
<td>8192</td>
<td>16</td>
<td>2D</td>
<td>512</td>
</tr>
<tr>
<td>2DECOMP&amp;FFT</td>
<td>0.0057</td>
<td>512</td>
<td>1</td>
<td>2D</td>
<td>512</td>
</tr>
<tr>
<td>P3DFFT</td>
<td>0.0060</td>
<td>4096</td>
<td>16</td>
<td>2D</td>
<td>256</td>
</tr>
<tr>
<td>P3DFFT</td>
<td>0.0063</td>
<td>512</td>
<td>1</td>
<td>2D</td>
<td>512</td>
</tr>
<tr>
<td>FFTW</td>
<td>0.0124</td>
<td>256\textsuperscript{*}</td>
<td>1</td>
<td>1D</td>
<td>256</td>
</tr>
<tr>
<td>P3DFFT</td>
<td>0.0130</td>
<td>256</td>
<td>1</td>
<td>1D</td>
<td>256</td>
</tr>
<tr>
<td>2DECOMP&amp;FFT</td>
<td>0.0140</td>
<td>256</td>
<td>1</td>
<td>1D</td>
<td>256</td>
</tr>
<tr>
<td>P3DFFT</td>
<td>0.0233</td>
<td>256</td>
<td>16</td>
<td>1D</td>
<td>16</td>
</tr>
<tr>
<td>2DECOMP&amp;FFT</td>
<td>0.0244</td>
<td>256</td>
<td>16</td>
<td>1D</td>
<td>16</td>
</tr>
<tr>
<td>FFTW</td>
<td>0.0341</td>
<td>256</td>
<td>16</td>
<td>1D</td>
<td>16</td>
</tr>
</tbody>
</table>

\^1 Libraries sorted by time.
\^2 Total number of cores
\^* Maximum number of total cores (1D decomposition).

Figure 5.9 shows, another interesting point. The 2DECOMP&FFT is more efficient on the 2D decomposition when 16 cores per node were used. In contrast, the P3DFFT does not have very good scaling because after the 4096 total cores has a decrease in performance.

Table 5.6 shows the minimum timings measured for each library during the tests, for both 1D and 2D decomposition.

For small 3D FFTs with dimensions $128 \times 128 \times 128$, the speedup of each library is illustrated in figure 5.10. In addition, table 5.7 presents the best timing achieved by each library either using 1D or 2D decomposition.

The results obtained during this experiment are very valuable because they present clearly each one of the behaviour of the libraries and how they are affected by the communication overhead.

The libraries achieved their minimum timings by using the 2D decomposition. The minimum time achieved by using the 2D decomposition was the P3DFFT library using 512 cores (1 core per node). The timing achieved by the FFTW 3.3 by using the 1D decomposition and 128 cores (1 core per node).
Another interesting result is that the 2DECOMP&FFT achieved a timing of 0.0015s by using 2D decomposition, while the FFTW achieved the timing 0.0035s by using 128 cores (1 core per node). This result shows that the 2D decomposition achieved to over- come the 1D decomposition’s limit factor of 128 cores and reduce the computational time by 57.14% when using 256 cores (1 core per node).

In addition, the 2D decomposition outperforms the 1D decomposition by using a fixed number of nodes. In particular, the minimum time achieved by the 1D decompo- sition was 0.0035s by using 128 cores (FFTW), whereas the 2D decomposition achieved a time around 0.0020s which is reduced is around 42.85%.

Further analysis involves the cost efficiency of the decompositions. The 0.0020s required from the 2DECOMP&FFT library by using 2048 cores and 128 nodes (16 cores per node) is not so important compared to the same library’s performance by using only 4 nodes and the 1D decomposition. The table 5.7 shows that the timing using 4 nodes is similar for the P3DFFT’s. Such difference in timings is not so important in real world simulations of this kind of 3D FFTs because the cost difference between 4 and 128 nodes is disproportionate to the benefit of the decreased time.

Finally, comparing the performance of the decompositions as the cube’s dimensions decrease it is obvious that the communication overhead affects significantly the time
Table 5.7: 128³: 1D & 2D decomposition results

<table>
<thead>
<tr>
<th>Library</th>
<th>Time¹</th>
<th>Cores²</th>
<th>Cores/Node</th>
<th>Decomposition</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3DFFT</td>
<td>0.0013</td>
<td>512</td>
<td>1</td>
<td>2D</td>
<td>512</td>
</tr>
<tr>
<td>2DECOMP&amp;FFT</td>
<td>0.0015</td>
<td>256</td>
<td>1</td>
<td>2D</td>
<td>256</td>
</tr>
<tr>
<td>P3DFFT</td>
<td>0.0017</td>
<td>4096</td>
<td>16</td>
<td>2D</td>
<td>256</td>
</tr>
<tr>
<td>2DECOMP&amp;FFT</td>
<td>0.0020</td>
<td>2048</td>
<td>16</td>
<td>2D</td>
<td>128</td>
</tr>
<tr>
<td>FFTW</td>
<td>0.0035</td>
<td>128</td>
<td>1</td>
<td>1D</td>
<td>128</td>
</tr>
<tr>
<td>P3DFFT</td>
<td>0.0035</td>
<td>64</td>
<td>1</td>
<td>1D</td>
<td>64</td>
</tr>
<tr>
<td>2DECOMP&amp;FFT</td>
<td>0.0039</td>
<td>64</td>
<td>1</td>
<td>1D</td>
<td>64</td>
</tr>
<tr>
<td>FFTW</td>
<td>0.0058</td>
<td>128*</td>
<td>16</td>
<td>1D</td>
<td>8</td>
</tr>
<tr>
<td>P3DFFT</td>
<td>0.0065</td>
<td>64</td>
<td>16</td>
<td>1D</td>
<td>4</td>
</tr>
<tr>
<td>2DECOMP&amp;FFT</td>
<td>0.0068</td>
<td>64</td>
<td>16</td>
<td>1D</td>
<td>4</td>
</tr>
</tbody>
</table>

¹ Libraries sorted by time.
² Total number of cores
* Maximum number of total cores (1D decomposition).

Table 5.8: 128³: 2D decomposition results for 512 Hector’s nodes.

required for the computations. However, as the dimensions increases the 2D decomposition takes advantage of the available cores and minimises the computations time.

**Demanding Communications and Interconnection Network Performance**

In order to investigate the affect of the Gemini Interconnect on the libraries performance we implemented another set of experiments. In the beginning of these experiments the total number of nodes was fixed to 512. The decomposition chosen for the experiment was the 2D and the libraries were the P3DFFT and the 2DECOMP&FFT, because they support 2D decomposition. The 3D array dimensions were $128 \times 128 \times 128$ in order to understand the interconnect affect by using small 3D FFTs.

In table [5.2] the level which the Gemini Interconnect affected the results is clear. There is a decrease in the performance of the P3DFFT as the number of cores per node increased up to 16. The performance decreased by 84.61%. The second library had a similar behaviour. As the cores per node increased up to 16 the performance of the 2DECOMP&FFT decreased by 73.68%.
By comparing the results which presented in figure 5.10, table 5.2 and table A.4, we observe that after a certain computation time between 0.0013 and 0.0017 both libraries could not achieve lower timings. Therefore, the interconnection network places a limit in computation time because of the communication overhead. Thus, even the number of nodes or cores is increased the computation time can not reduced. Therefore, in the test case presented in table A.4 there is a performance decrease for both libraries.

Finally, the project [55] includes a performance comparison between the SeaStar and the Gemini Interconnect and present the ability of the test program which make use the Gemini Interconnect to scale almost ideal up to 10000 cores.
Chapter 6

Conclusions & Discussions

The results obtained during the project were very useful and let us make a number of useful observations. The final thoughts about these observations are brought in this chapter which presents the conclusions we made after the results interpretation.

Finally, we discuss the goals of this project and we suggest some thoughts in order to improve further the benchmarks.

6.1 Conclusions

The obtained results show that by using the slab decomposition, the FFTW 3.3 achieved very good scaling both using small and medium size cubes (the $128 \times 128 \times 128$ and the $256 \times 256 \times 256$). The other two libraries achieved similar performance to the FFTW and both had an almost identical performance.

As for the pencil decomposition, both P3DFFT and 2DECOMP&FFT had similar performance. However, the communication overhead didn’t let the libraries to scale very well after a certain number of cores depending on the cube size and the number of cores per node.

Comparing the slab and the pencil decomposition, the pencil decomposition seems to be more efficient and achieved to minimise the computation time because of the large total number of cores. However, as we reduced the size of the cubes the P3DFFT and the 2DECOMP&FFT didn’t take advantage of the increased limit of tasks allowed by the 2D decomposition, and they didn’t scale normally. In addition, from the results the minimum timings by using either 1D or 2D were too close by using cubes with dimensions $128 \times 128 \times 128$. 

50
This means that probably for very small 3D FFTs $32^3$ or $64^3$, the performance of the libraries will be similar either using the 1D or the 2D decomposition.

During the investigation for the best processors grid for the 2D decomposition we concluded that the 2DECOMP&FFT library’s routine which selects the processors grid works very well, especially for large number of cores. The same processors grid affected in the same way the performance of the P3DFFT library.

The experiment with the FFTW’s OpenMP functions, showed that the shared memory programming can outperform the MPI for the intra-node communications. Especially, this makes very interesting the mixed-mode programming model performance on the 3D FFTs computations.

The flexibility of the P3DFFT and 2DECOMP&FFT libraries by using either 1D or 2D decomposition make them very efficient for the real world simulations which usually, were described by large 3D FFTs.

In our project, when using the 2D decomposition, after a total number of cores the communication overhead didn’t allowed them to reduce further the computation time. On the other hand they outperform the 1D decomposition implementations. The difference in time was increased as the dimensions increased. Therefore, for larger cubes dimensions (i.e. $2048 \times 2048 \times 2048$ etc.) the computational time using the 2D decomposition compared to the 1D decomposition computation time will probably be perceivable to the end user.

To sum up, the Gemini Interconnect allowed the pencil decomposition to outperform the slab decomposition for the certain 3D mesh dimensions we used. However, the network set up a limit because the communication overhead didn’t allowed the libraries to scale after a number of cores. In addition, this limit was increased when the cube’s dimensions increased.

### 6.2 Discussions

Obviously, the interconnection network performance is an upper bound to the modern supercomputer’s performance. The Gemini Interconnect achieved to set the upper bound higher according to the experimental results. By analysing the OpenMP test program speedup, we conclude that the network interconnect was not the only factor which affects the performance of the FFT computation. Therefore, the message passing programming approach lacks of performance in the SMP systems compared to the OpenMP.
Unfortunately, in this project we didn’t succeed to implement the test programs by using the hybrid OpenMP/MPI model. The main reason was that the FFTW 3.3 released on the 26th of July, and before the final version there wasn’t any documentation related to the new API.

In addition, the 3.3 Alpha version we used in the beginning of the project, was implemented by using the 3.2 MPI code. Therefore, the timings we had from the 3.3 alpha have changed when we used the 3.3 final version. As a result, we did the performance comparison from the beginning because all the other libraries were built on the top of the 3.3 alpha.

Another problem which delayed the project was the fact that the obtained results didn’t represent the performance of the libraries. This happened because on the FFTW test program when we measured the time we included the time required for planning. In addition, we built the P3DFFT library without the --enable-measure flag which enables the MEASURE, in order to have a better plan according to our hardware. However, we found these problems and the final results represent the correct libraries performance for our experiments.

Finally, in the project plans was to implement the same benchmarking procedure in ECDF Cluster "Eddie" [57]. However, problems during the installation and the difficulty to run jobs in the InfiniBand [58] didn’t allowed us to test our programs so many times and the obtained data were not reliable. In addition, the results obtained by using the Ethernet network in Eddie were not so interesting for our research. In particular, the number of cores using the InfiniBand (816 total cores - 68 nodes) was not comparable to the 8192 total cores we used for the 2D decomposition in Hector.

### 6.3 Future Work

In this project, the benchmarks was implemented using real data input, because the P3DFFT 2.4 does not support complex-to-complex transforms. Another benchmark can be implement concerning complex number using the PFFT [59] instead of the P3DFFT. The PFFT library is developed from the Chemnitz University of Technology [60] and it is a higher-level library built on top of the FFTW 3.3. The main idea of these benchmarks is to use libraries which are built on the top of the new FFTW 3.3 library, in order to examine the new APIs performance.

In addition, our project can be expanded by using the hybrid OpenMP/MPI programming model in order to compare the libraries and to compare the mixed mode results to the MPI results. An interesting point is that the P3DFFT and the 2DECOMP&FFT does
not support an automated way to run tests using this programming model (the FFTW 3.3 supports OpenMP). The results will be very interesting as presented in the [52]. However, it will be a demanding project, and the final results and source codes can be a contribution to the libraries source codes.

Many supercomputer vendors have started to build clusters consist of powerful graphic processor units and many research centres such the National Supercomputing Center in Tianjin [61] (China) runs the Tianhe-1A supercomputer [62] which consists of GPUs. These developments distracted of the High Performance Computing scientific world and many researchers and industries have started to investigate the GPGPU programming approach further.

In particular, it will be very interesting as a further work to implement a benchmark by using the most popular available GPGPU FFT libraries. Such an experiment will be very interesting because the GPUs performance in raw numbers outperform the CPUs performance. However, the programming models which should be used are in very early stages and especially the demanding communication of the FFTs will affect the scalability.
Appendix A

Results

A.1 1D and 2D decomposition results

Figure A.1: SLAB: Libraries performance using $256^3$ 3D FFT with 1 core per node
Figure A.2: SLAB: Libraries performance using $256^3$ 3D FFT, 16 cores per node

Figure A.3: SLAB: Libraries performance using $128^3$ 3D FFT, 1 core per node
Figure A.4: OpenMP performance versus the MPI implementations up to 16 cores - 256³ 3D FFT

<table>
<thead>
<tr>
<th>Cores</th>
<th>P3DFFT</th>
<th>SPEEDUP</th>
<th>2DECOMP&amp;FFT</th>
<th>SPEEDUP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.2954</td>
<td>1.00</td>
<td>1.3716</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.7670</td>
<td>1.69</td>
<td><strong>0.7330</strong></td>
<td>1.87</td>
</tr>
<tr>
<td>4</td>
<td><strong>0.6191</strong></td>
<td>2.09</td>
<td>0.6238</td>
<td>2.20</td>
</tr>
<tr>
<td>8</td>
<td><strong>0.4006</strong></td>
<td>3.23</td>
<td>0.4049</td>
<td>3.39</td>
</tr>
<tr>
<td>16</td>
<td><strong>0.2171</strong></td>
<td>5.97</td>
<td>0.2197</td>
<td>6.24</td>
</tr>
<tr>
<td>32</td>
<td><strong>0.1177</strong></td>
<td>11.01</td>
<td>0.1190</td>
<td>11.53</td>
</tr>
<tr>
<td>64</td>
<td><strong>0.0683</strong></td>
<td>18.97</td>
<td>0.0697</td>
<td>19.69</td>
</tr>
<tr>
<td>128</td>
<td>0.0396</td>
<td>32.72</td>
<td><strong>0.0366</strong></td>
<td>37.48</td>
</tr>
<tr>
<td>256</td>
<td><strong>0.0196</strong></td>
<td>66.05</td>
<td><strong>0.0196</strong></td>
<td>70.08</td>
</tr>
<tr>
<td>512</td>
<td><strong>0.0096</strong></td>
<td>134.34</td>
<td>0.0106</td>
<td>129.38</td>
</tr>
<tr>
<td>1024</td>
<td>0.0081</td>
<td>159.36</td>
<td><strong>0.0079</strong></td>
<td>173.66</td>
</tr>
<tr>
<td>2048</td>
<td>0.0074</td>
<td>174.68</td>
<td><strong>0.0070</strong></td>
<td>197.24</td>
</tr>
<tr>
<td>4096</td>
<td><strong>0.0060</strong></td>
<td>215.17</td>
<td>0.0065</td>
<td>210.13</td>
</tr>
<tr>
<td>8192</td>
<td>0.0082</td>
<td>141.32</td>
<td><strong>0.0046</strong></td>
<td>299.30</td>
</tr>
</tbody>
</table>

Table A.1: PENCIL: 256³, 16 cores per node, time and speedup
<table>
<thead>
<tr>
<th>Cores</th>
<th>P3DFFT</th>
<th>SPEEDUP</th>
<th>2DECOMP&amp;FFT</th>
<th>SPEEDUP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.2954</td>
<td>1.00</td>
<td>1.3716</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.7061</td>
<td>1.83</td>
<td>0.7220</td>
<td>1.90</td>
</tr>
<tr>
<td>4</td>
<td>0.4453</td>
<td>2.91</td>
<td>0.4466</td>
<td>3.07</td>
</tr>
<tr>
<td>8</td>
<td>0.2291</td>
<td>5.66</td>
<td>0.2262</td>
<td>6.06</td>
</tr>
<tr>
<td>16</td>
<td>0.1202</td>
<td>10.78</td>
<td>0.1182</td>
<td>11.61</td>
</tr>
<tr>
<td>32</td>
<td>0.0595</td>
<td>21.79</td>
<td>0.0602</td>
<td>22.77</td>
</tr>
<tr>
<td>64</td>
<td>0.0298</td>
<td>43.52</td>
<td>0.0310</td>
<td>44.17</td>
</tr>
<tr>
<td>128</td>
<td>0.0161</td>
<td>80.63</td>
<td>0.0163</td>
<td>84.37</td>
</tr>
<tr>
<td>256</td>
<td>0.0090</td>
<td>144.62</td>
<td>0.0086</td>
<td>159.78</td>
</tr>
<tr>
<td>512</td>
<td>0.0063</td>
<td>205.21</td>
<td>0.0058</td>
<td>238.46</td>
</tr>
</tbody>
</table>

Table A.2: PENCIL: $256^3$, 1 core per node, time and speedup

<table>
<thead>
<tr>
<th>Cores</th>
<th>P3DFFT</th>
<th>SPEEDUP</th>
<th>2DECOMP&amp;FFT</th>
<th>SPEEDUP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1445</td>
<td>1.00</td>
<td>0.1555</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.0793</td>
<td>1.82</td>
<td>0.0807</td>
<td>1.93</td>
</tr>
<tr>
<td>4</td>
<td>0.0502</td>
<td>2.88</td>
<td>0.0511</td>
<td>3.04</td>
</tr>
<tr>
<td>8</td>
<td>0.0249</td>
<td>5.80</td>
<td>0.0247</td>
<td>6.29</td>
</tr>
<tr>
<td>16</td>
<td>0.0124</td>
<td>11.64</td>
<td>0.0125</td>
<td>12.41</td>
</tr>
<tr>
<td>32</td>
<td>0.0066</td>
<td>22.01</td>
<td>0.0073</td>
<td>21.22</td>
</tr>
<tr>
<td>64</td>
<td>0.0034</td>
<td>42.56</td>
<td>0.0034</td>
<td>46.38</td>
</tr>
<tr>
<td>128</td>
<td>0.0026</td>
<td>55.41</td>
<td>0.0024</td>
<td>65.24</td>
</tr>
<tr>
<td>256</td>
<td>0.0023</td>
<td>64.00</td>
<td>0.0015</td>
<td>102.69</td>
</tr>
<tr>
<td>512</td>
<td>0.0013</td>
<td>108.89</td>
<td>0.0019</td>
<td>80.78</td>
</tr>
</tbody>
</table>

Table A.3: PENCIL: $128^3$, 1 core per node, time and speedup
<table>
<thead>
<tr>
<th>Cores</th>
<th>Processors Grid</th>
<th>Time (sec)</th>
<th>Speedup</th>
<th>Processors Grid</th>
<th>Time (sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1x1</td>
<td>0.1445</td>
<td>1.00</td>
<td>1x1</td>
<td>0.1555</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>1x2</td>
<td>0.0851</td>
<td>1.70</td>
<td>1x2</td>
<td>0.0871</td>
<td>1.79</td>
</tr>
<tr>
<td>4</td>
<td>2x2</td>
<td>0.0785</td>
<td>1.84</td>
<td>2x2</td>
<td>0.0765</td>
<td>2.03</td>
</tr>
<tr>
<td>8</td>
<td>2x4</td>
<td>0.0528</td>
<td>2.74</td>
<td>2x4</td>
<td>0.0506</td>
<td>3.07</td>
</tr>
<tr>
<td>16</td>
<td>4x4</td>
<td>0.0247</td>
<td>5.86</td>
<td>4x4</td>
<td>0.0237</td>
<td>6.57</td>
</tr>
<tr>
<td>32</td>
<td>4x8</td>
<td>0.0124</td>
<td>11.68</td>
<td>2x16</td>
<td>0.0136</td>
<td>11.41</td>
</tr>
<tr>
<td>64</td>
<td>8x8</td>
<td>0.0067</td>
<td>21.70</td>
<td>8x8</td>
<td>0.0074</td>
<td>21.00</td>
</tr>
<tr>
<td>128</td>
<td>4x32</td>
<td>0.0042</td>
<td>34.30</td>
<td>4x32</td>
<td>0.0040</td>
<td>38.41</td>
</tr>
<tr>
<td>256</td>
<td>8x32</td>
<td>0.0039</td>
<td>37.44</td>
<td>8x32</td>
<td>0.0030</td>
<td>52.41</td>
</tr>
<tr>
<td>512</td>
<td>8x64</td>
<td>0.0026</td>
<td>54.62</td>
<td>8x64</td>
<td>0.0027</td>
<td>56.76</td>
</tr>
<tr>
<td>1024</td>
<td>16x64</td>
<td>0.0026</td>
<td>56.02</td>
<td>16x64</td>
<td>0.0025</td>
<td>61.98</td>
</tr>
<tr>
<td>2048</td>
<td>32x64</td>
<td>0.0022</td>
<td>64.64</td>
<td>32x64</td>
<td>0.0020</td>
<td>76.68</td>
</tr>
<tr>
<td>4096</td>
<td>64x64</td>
<td>0.0017</td>
<td>84.33</td>
<td>64x64</td>
<td>0.0031</td>
<td>50.20</td>
</tr>
<tr>
<td>8192</td>
<td>64x128</td>
<td>0.0024</td>
<td>60.63</td>
<td>64x128</td>
<td>0.0033</td>
<td>47.01</td>
</tr>
</tbody>
</table>

Table A.4: PENCIL: $128^3$, 16 cores per node, time and speedup

Figure A.5: PENCIL: Libraries performance using $128^3$ 3D FFT, 1 core per node
Figure A.6: PENCIL: Libraries performance using $128^3$ 3D FFT, 16 cores per node

Figure A.7: PENCIL: Libraries performance using $256^3$ 3D FFT, 1 core per node
Figure A.8: PENCIL: Libraries performance using $256^3$ 3D FFT, 16 cores per node

Figure A.9: Libraries performance using $256^3$ 3D FFT, 512 total cores
Bibliography


[60] [Online]. Available: http://www-user.tu-chemnitz.de/
